Software Engineering Principles Notes 2016-2017

Version: 0.65

Glenn L. Jenkins

Print Date: March 30, 2017

©2017 Dr. Glenn L. Jenkins

Contents

1 An Introduction to Software Engineering 12

1.1 Introduction . . . 12
1.2 Software Engineering Defined . . . 12
1.2.1 From the NATO conference of Software Engineering . . 12
1.2.2 IEEE Definition . . . 12
1.2.3 Definition from Schach . . . 13
1.2.4 Definition from Sommerville . . . 13
1.2.5 Other definitions of Engineering . . . 14
1.2.6 Common Aspects . . . 14
1.3 What makes software difficult to create? . . . 15
1.3.1 Spectacular Early Failures . . . 16
1.4 Evolution of Software Engineering . . . 17
1.4.1 A Brief History . . . 17
1.4.2 Where are we now? . . . 21
1.5 Software Process . . . 23
1.5.1 Development Methodologies? . . . 24
1.5.2 Example: Waterfall . . . 24
1.5.3 Cost of Finding Flaws? . . . 25
1.5.4 So what is Software Engineering? . . . 26
1.6 Why do YOU need to know? . . . 29
1.6.1 Roadmap for the Module . . . 30
1.7 Tutorial - Definitions and Rationales . . . 31
1.8 Summary . . . 31
1.9 Preparation for Next Week . . . 32

2 Models of Software Development 33

2.1 Introduction . . . 33
2.2 Terminology . . . 33
2.3 Software Processes . . . 35
2.4 Process Activities . . . 36

2.4.1 Software Specification 36
2.4.2 Software Design and Implementation 39
2.4.3 Software Validation 42
2.4.4 Software Maintenance & Evolution 44
2.5 Software Process Models 44
2.5.1 Origin of Process Models 45
2.5.2 Coping with Change 46
2.5.3 Evolution of Process Models 47
2.6 Planned and Agile 47
2.6.1 Agile Methods 50
2.6.2 Home Grounds 53
2.7 Plan for this year 53
2.8 Tutorial - Planned Or Agile 55
2.9 Summary 56
2.10 Preparation for Next Week 56

3 Ballencing Agility and Discipine 57
3.1 Introduction 57
3.2 Agile and Planned 57
3.3 Examples - Plan Driven Approaches 58
3.3.1 Waterfall 58
3.3.2 Spiral 62
3.3.3 Code and Fix 65
3.3.4 Brief History of Agile 66
3.3.5 Key Characteristics of Agile Methods 68
3.4 Almost Agile? 69
3.4.1 Protoyping 69
3.4.2 RAD 70
3.4.3 DSDM 70
3.5 Examples - Agile Methods 71
3.5.1 Extreme Programming 71
3.6 Additional Reading 74
3.7 Recap - Home Grounds 74
3.8 Balanced Approaches 74
3.8.1 UP and Agile Modelling 74
3.8.2 ROPES 82
3.8.3 Do we need to design? 82
3.9 Plan for this year 83
3.10 Tutorial - Evaluating Development Models 83
3.11 Summary 84
3.12 Preparation for Next Week 84

4 Agile UP: Inception (Part 1) 85
4.1 Introduction . . . 85
4.2 Inception . . . 86
4.2.1 The Development Case . . . 87
4.3 Requirements . . . 88
4.3.1 Requirements Engineering . . . 88
4.3.2 Functional and Non-Functional . . . 90
4.3.3 Requirements Engineering . . . 92
4.4 Evolutionary Requirements . . . 98
4.4.1 Skilful Means to Find Requirements . . . 98
4.4.2 FURPS+ . . . 99
4.5 Where are the Requirements in Agile UP . . . 99
4.6 Tutorial . . . 100
4.6.1 Case Study . . . 101
4.7 Summary . . . 102
4.8 Preparation for Next Week . . . 102
4.8.1 Additional Reading . . . 102
4.8.2 Essential Reading . . . 103
4.8.3 Background Reading . . . 103

5 Agile UP: Inception (Part 2) 105
5.1 Introduction . . . 105
5.2 Inception - Recap . . . 105
5.3 Use Case . . . 106
5.3.1 Key Concepts . . . 107
5.3.2 Formal Definition of Use Cases . . . 108
5.3.3 The Use-Case Model . . . 109
5.3.4 Rational for Using Use-Case . . . 110
5.3.5 Use-Case Notation . . . 110
5.4 Use-Case Diagrams . . . 111
5.4.1 Basic Use-Case Syntax . . . 112
5.4.2 Intermediate Use-Case Syntax . . . 112
5.5 Guidelines for Creating Use Case . . . 114
5.5.1 Guideline: Essential Style . . . 114
5.5.2 Guideline: Terse Use Cases . . . 114
5.5.3 Guideline: Black-Box Use Case . . . 114
5.5.4 Guideline: Take an Actor and Actor-Goal Perspective . 115
5.5.5 Creation: Steps to Identify Use Cases . . . 115
5.5.6 The System Boundary . . . 116

5.5.7 Checking: Tests to ensure we've found 'useful' use case 118
5.6 Use-Case in UP . . . 118
5.6.1 Use-Case - In Inception . . . 119
5.6.2 Use-Case - In Elaboration . . . 119
5.7 Tutorial . . . 120
5.7.1 Case Study . . . 120
5.8 Summary . . . 122
5.8.1 Text or Diagram . . . 122
5.9 Preparation for Next Week . . . 123
5.9.1 Essential Reading . . . 123
5.9.2 Background Reading . . . 123

6 Agile UP: Inception (Part 3) 137
6.1 Introduction . . . 137
6.2 Inception - Recap . . . 137
6.3 Where are the Requirements in Agile UP . . . 138
6.4 Supplementary Specification . . . 139
6.4.1 FURPS+ . . . 139
6.5 Vision . . . 141
6.5.1 Important Sections . . . 142
6.6 Glossary . . . 143
6.7 Business or Domain Rules . . . 144
6.8 Use-case or Vision first? . . . 144
6.9 Evolutionary Requirements and UP . . . 145
6.10 Tutorial . . . 145
6.10.1 Case Study . . . 146
6.11 Summary . . . 147
6.12 Preparation for Next Week . . . 147
6.12.1 Essential Reading . . . 147
6.12.2 Background Reading . . . 148

7 Project and Team Management 158
7.1 Introduction . . . 158
7.2 Leadership and Management . . . 158
7.2.1 Theories of Leadership . . . 159
7.2.2 Management . . . 167
7.2.3 Leadership in Groups . . . 169
7.2.4 Power . . . 171
7.2.5 Conflict . . . 172
7.3 Motivation . . . 174
7.3.1 The Motivation Process . . . 174

7.3.2 Needs and Expectations . . . 174
7.3.3 Barriers . . . 175
7.3.4 Motivation Theory . . . 176
7.4 Organisations . . . 180
7.4.1 Organisational Culture . . . 180
7.4.2 Organisational Structure . . . 182
7.5 Summary . . . 184
7.6 Preparation for Next Week . . . 184
7.6.1 Essential Reading . . . 184
7.6.2 Background Reading . . . 184

8 Agile UP: Elaboration - Object Model (Part 1) 186
8.1 Introduction . . . 186
8.2 Inception - What just happened? . . . 186
8.3 Elaboration . . . 187
8.3.1 Planning Iterations . . . 188
8.3.2 UP Disciplines (Workflows) . . . 190
8.4 Domain Models . . . 192
8.4.1 Domain Model as a Visual Dictionary . . . 193
8.4.2 Domain Models and Decomposition . . . 193
8.4.3 Conceptual Classes . . . 193
8.4.4 Examples . . . 194
8.4.5 Lowering the Representational Gap . . . 194
8.5 Domain Diagram Creation . . . 195
8.5.1 How to find Conceptual Classes . . . 195
8.5.2 Noun Phrase Identification . . . 195
8.5.3 Class Diagrams at Different Levels . . . 199
8.6 Domain Diagram Syntax . . . 199
8.6.1 Basic Class Diagrams . . . 199
8.7 Domain Modelling Guidelines . . . 204
8.7.1 Think Like a Map Maker . . . 205
8.7.2 Concept or Attribute . . . 205
8.7.3 Description or Specification Conceptual Class . . . 206
8.8 Tutorial . . . 207
8.8.1 Case Study 1 . . . 207
8.9 Summary . . . 208
8.10 Preparation for Next Week . . . 209
8.10.1 Essential Reading . . . 209
8.10.2 Background Reading . . . 210

9 Agile UP: Elaboration - Object Model (Part 2) 211
9.1 Introduction . . . 211
9.2 Elaboration . . . 211
9.3 Domain Models - Recap . . . 212
9.4 Orientation - from last week . . . 213
9.5 Associations . . . 213
9.5.1 Identifying Associations . . . 217
9.6 Attributes . . . 219
9.6.1 What attributes should be shown? . . . 220
9.6.2 Identifying Attributes . . . 223
9.7 Is the domain diagram correct? . . . 223
9.7.1 The Role of Domain Models in UP . . . 224
9.8 Tutorial . . . 224
9.8.1 Case Study 1 . . . 224
9.9 Summary . . . 226
9.10 Preparation for Next Week . . . 226
9.10.1 Essential Reading . . . 226
9.10.2 Background Reading . . . 227

10 Agile UP: Elaboration - System Sequence Diagrams 228
10.1 Introduction . . . 228
10.2 Elaboration - Recap . . . 228
10.3 System Sequence Diagrams . . . 229
10.3.1 Key Motivation for SSD . . . 230
10.4 Sequence Diagram Syntax - Part 1 . . . 231
10.4.1 Participants and Messages . . . 231
10.4.2 Frame Operators . . . 238
10.5 System Sequence Diagram Creation . . . 242
10.5.1 Guidelines . . . 243
10.5.2 Should we sketch SSDs for everything? . . . 244
10.5.3 SSDs within UP . . . 244
10.6 Tutorial . . . 244
10.6.1 Case Study 1 . . . 245
10.6.2 Use Case . . . 246
10.7 Summary . . . 249
10.8 Preparation for Next Week . . . 249
10.8.1 Essential Reading . . . 249
10.8.2 Background Reading . . . 250

11 Agile UP: Elaboration - Moving to Design: Architectural Design 251
11.1 Introduction . . . 251
11.2 Elaboration - Recap . . . 252
11.2.1 Iterations, Evolution and Change . . . 252
11.3 Moving to Design - A change of Emphasis . . . 253
11.3.1 Stages of Design . . . 253
11.4 Architectural Design . . . 255
11.4.1 Example Logical Architecture . . . 256
11.4.2 Layered Architectures . . . 257
11.4.3 Aside: Layers, Tiers and Partitions . . . 258
11.5 Packages Diagrams . . . 259
11.5.1 Reverse Engineering Package Diagrams . . . 262
11.6 Guidelines for Architectural Design . . . 262
11.6.1 Design with Layers . . . 262
11.6.2 Maintain Separation of Concerns . . . 264
11.6.3 Name Packages Carefully . . . 265
11.6.4 Do now show Physical Resources below the Foundation/Technical Services Layer . . . 265
11.7 Application Logic in the Domain Layer . . . 265
11.7.1 Domain Layer and Domain Object Model . . . 266
11.8 The Model View Separation Principle . . . 267
11.8.1 Layers and SSDs . . . 269
11.9 Tutorial . . . 270
11.9.1 Case Study 1 . . . 271
11.9.2 Use Case . . . 272
11.9.3 Domain Object Model . . . 275
11.10Summary . . . 275
11.11Preparation for Next Week . . . 276
11.11.1Essential Reading . . . 276
11.11.2Background Reading . . . 276

12 Agile UP: Elaboration - Designing with Patterns/Principles: Mechanistic Design - Part 1 277
12.1 Introduction . . . 277
12.2 Elaboration - Recap . . . 278
12.2.1 Iterations, Evolution and Change . . . 278
12.3 Towards Object Design . . . 279
12.3.1 Agile Modelling and Lightweight UML Drawing . . . 280
12.3.2 Stages of Design . . . 281
12.4 Mechanistic Design . . . 284

12.5 Models in Mechanistic Design 284
12.6 Use Case Realisation 285
12.6.1 Orientation: What's next? 285
12.7 Responsibilities and Responsibility Driven Design 286
12.7.1 Aside: Class Responsibility Communication Cards . . . 286
12.7.2 RRD 287
12.7.3 GRASP - Design Principles (mini-patterns) 289
12.7.4 Example - Solving A Simple Design Problem using a GRASP Pattern 290
12.8 More Class Diagrams 295
12.8.1 Associations Between Classes 296
12.8.2 Specialised Associations 302
12.8.3 UML Recap - Sequence Diagrams 303
12.8.4 Participants and Messages 303
12.8.5 Frame Operators 310
12.9 Tutorial 314
12.9.1 Case Study 1 315
12.9.2 Case Study 2 319
12.10Summary 320
12.11Preparation for Next Week 321
12.11.1Essential Reading 321
12.11.2Background Reading 321

13 Agile UP: Elaboration - Designing with Patterns/Principles: Mechanistic Design - Part 2 322
13.1 Introduction 322
13.2 Recap: Use Case Realisation 322
13.2.1 Orientation: What's next? 322
13.3 Recap: Responsibilities and Responsibility Driven Design (RRD) 323
13.4 Responsibilities and Responsibility Driven Design (RRD) . . . 324
13.4.1 Recap: What is RRD? 324
13.4.2 Solving A Simple Design Problems using a GRASP Patterns - Continued 325
13.5 More Class Diagrams 345
13.5.1 Dependency 345
13.6 More Sequence Diagrams 346
13.6.1 Collections and Return Variables 346
13.6.2 Messages to Self 347
13.7 Tutorial 348
13.7.1 Case Study 1 348
13.7.2 Case Study 2 354

13.7.3 Domain Object Diagram for Case Study 2 355
13.7.4 System Sequence Diagram for Case Study 2 355
13.8 Summary 356
13.9 Preparation for Next Week 357
13.9.1 Essential Reading 357
13.9.2 Background Reading 357

14 Agile UP: Elaboration - Designing with Patterns/Principles: Mechanistic Design - Part 3 358
14.1 Introduction 358
14.2 Use Case Realisation 358
14.2.1 Orientation: What's next? 359
14.3 Responsibilities and Responsibility Driven Design (RRD) . . . 359
14.3.1 Recap: What is RRD? 359
14.3.2 Recap: GRASP - Design Principles (mini-patterns) . . 360
14.3.3 Solving A Simple Design Problems using a GRASP Patterns - Continued 360
14.3.4 The Command-Query Separation Principle 375
14.4 More Class Diagrams 376
14.4.1 Generalisation 376
14.5 More Sequence Diagrams 377
14.5.1 Interactions with Polymorphic Instances 377
14.5.2 Relating Interaction Diagrams 378
14.6 Tutorial 380
14.6.1 Case Study 1 381
14.6.2 Case Study 2 386
14.6.3 Domain Object Diagram for Case Study 2 386
14.6.4 System Sequence Diagram for Case Study 2 387
14.7 Summary 388
14.8 Preparation for Next Week 389
14.8.1 Essential Reading 389
14.8.2 Background Reading 389

15 Agile UP: Elaboration - Designing with Patterns/Principles: Mechanistic Design - Part 4 390
15.1 Introduction 390
15.2 Use Case Realisation 390
15.2.1 Orientation: What's next? 391
15.3 Applying GRASP to Realise a Scenario/Use Case 391
15.3.1 Monopoly Case Study: Reminder 391
15.3.2 The Design of playGame() 392

15.3.3 Initialisation and the Start-up Use Case 400
15.4 Additional UML 401
15.5 Tutorial 401
15.5.1 Case Study 1 401
15.5.2 Case Study 2 406
15.5.3 Domain Object Diagram for Case Study 2 407
15.5.4 System Sequence Diagram for Case Study 2 407
15.6 Summary 408
15.7 Preparation for Next Week 409
15.7.1 Essential Reading 409
15.7.2 Background Reading 409

16 Agile UP: Elaboration - Building the Implementation Model 410
16.1 Introduction 410
16.2 Where are we? 410
16.3 Programming and OOD 410
16.3.1 Creativity and Change 411
16.4 Mapping Designs to Code 412
16.4.1 Creating Class Definitions from DCDs 412
16.4.2 Creating Methods from Interaction Diagrams 412
16.4.3 Misc. Guidance 412
16.5 Test Driven Development 413
16.6 Team Based Development 414
16.6.1 Source Code Management 414
16.6.2 Git 414
16.6.3 Git Basics - A crash course in using git 415
16.7 Tutorial 415
16.8 Summary 416
16.9 Preparation for Next Week 416
16.9.1 Essential Reading 416
16.9.2 Background Reading 416

17 Agile UP: Elaboration - More Version Control 417
17.1 Introduction 417
17.1.1 Recal: Source Code Management 417
17.1.2 Recap: Git Basics 417
17.1.3 Branching and Merging 418
17.2 Tutorial 418
17.3 Summary 419
17.4 Preparation for Next Week 419
17.4.1 Essential Reading 419

17.4.2 Background Reading 419

18 Agile UP: Elaboration - Testing - part 1 420
18.1 Introduction 420
18.2 What is Testing? 420
18.3 Psychological and Economic Considerations 420
18.3.1 Attitude to Testing 420
18.3.2 Economic Factors 421
18.3.3 White-Box Testing 422
18.4 Myers Principles of Testing 423
18.5 Designing Test Cases 424
18.5.1 Black Box Testing Methodologies 425
18.5.2 White Box Testing Methodologies 431
18.5.3 Testing Strategy 436
18.6 Phases of Testing 437
18.6.1 Unit Testing 437
18.6.2 Integration Testing 437
18.6.3 Higher-Order Testing 442
18.7 Tutorial 450
18.8 Summary 451
18.9 Preparation for Next Week 451
18.9.1 Essential Reading 451
18.9.2 Background Reading 451

19 Google Test: Introduction and worksheet 453
19.1 Introduction 453
19.1.1 xUnit 454
19.2 Using Google Test 454
19.2.1 Basic Concepts 454
19.2.2 Assertions 454
19.2.3 Simple Tests 457
19.2.4 Test Fixtures - multiples tests with the same data . . . 457
19.3 Setup for Google Test 461
19.3.1 Windows 10 / Visual Studio 2015 461
19.4 Exercises 461
References 463

An Introduction to Software Engineering

1.1 Introduction

The modern world is dependent on software and software systems are essential for everything from our National infrastructures and utilities, to manufacturing and consumer products, to entertainment systems and media production [110]. This dependency is part of the reason you are studying this course, either you seek to contributed to one of the industries dependent on Software or simply to be employed. This introductory class will attempt to provide a overview of the topic and a rational for the topics selected for study in this introductory module.

1.2 Software Engineering Defined

There are a number of definitions of Software Engineering, considering a handful of these we'll then think about some more general definitions of engineering and consider the common points.

1.2.1 From the NATO conference of Software Engineering

[87]
Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

1.2.2 IEEE Definition

[24]

1. The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

2. The study of approaches as in 1.

1.2.3 Definition from Schach

[102]
Software Engineering is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the client's needs.

1.2.4 Definition from Sommerville

[110]
Software engineering is an engineering discipline that is concerned with all aspects of software production. Software is not just a program or programs but also includes documentation. Essential software product attributes are maintainability, dependability, security, efficiency, and acceptability.

Product characteristics	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.

Figure 1.2 Essential attributes of good software

Figure 1.1: from [110].

1.2.5 Other definitions of Engineering

Definitions of software engineering are similar to broader definitions e.g. that of engineering design:

"Engineering design is the process of devising a system, component, or process to meet desired needs. It is a decision-making process (often iterative), in which the basic science and mathematics and engineering sciences are applied to convert resources optimally to meet a stated objective." ABET Definition of Engineering Design.

1.2.6 Common Aspects

There is clearly a dual emphasis on product and process, i.e. the properties of the product and the process which creates it. Essentially the software produced needs to do what the customer/user wants it to do. The process which produces it must be a reliable one, we need to know how long it will take, what it will cost and that it will succeed!
An interesting side note here is that the Engineering Design process describes the application of related (or possibly underpinning,) knowledge in the development of engineering problems. In the same way Software Engineering is underpinned by Computer Science (the theoretical basis of computing,) and Systems Engineering (the development and evolution of complex systems) [110].

Aside:
The comparison with traditional engineering is a limiting one, while we've borrowed techniques in terms of planned phased delivery, audit, construction of a complete and clear design etc., there are marked differences [117]:

1. We regularly embark on software development projects beyond our current expertise. We use new technology, enter new domains tackle new kinds of problem.

2. The cost of constructing bridges comes at the end, when we build the bridge. The cost of copying/downloading/installing software is negligible all the cost comes with its design and development.

3. Software is logical rather than physical, physical products wear out over time. Problems with software are caused by errors which have

always existed but have only now manifested themselves or changes in specification. *Vilet goes on to argue that software may even wear out because it is maintained. It is my experience (at that of several members of staff,) the software experiences a kind of rot caused by many little changes over time all though maintenance of the software and its connected components (e.g. the libraries it uses or OS it runs on).*

4. Software is invisible. We can't see progress as we can when constructing a bridge. This leads to a number of problems when estimating the time require to undertake or complete a project.

5. Software is not continuous. Where as most physical systems are continuous and some change in their specification leads to a small change in the product this is not true of software. Small changes in the specification lead to large changes have considerable effect. The same is true of errors in the specification (a typing error crashed Mariner Venus probe, while a misunderstanding the specification of units caused the lost of the Mars Climate Orbiter).

1.3 What makes software difficult to create?

Unlike hardware software is abstract and intangible (we can't see/touch software, only the media on which it is stored,) as a result software is not constrained by the properties of materials, physics or limitation of manufacturing [110]. Computers are through software "logically malleable" and can undertake any task that we can design, train or evolve them to do [76][35], this makes them both flexible and powerful. In practice this means that software systems can quickly become complex, difficult to understand and expensive to change/maintain [110]. The lack of physicality is a major part of the problem here its hard to inspect or test something which in its final form is intangible. A bridge for example once constructed can be inspected, tested etc., code once compiled can only be executed (tested).

Software is abstract and intangible - but powerful.

The resulting range of software which exists further complicates the problem, there are many different types of software from embedded systems which themselves are found in everything from power stations to washing machines.

To organisational information systems which again can be found anywhere from a multinational bank, to a local health authority, to a small business or large university. Neither of which have much in common with a graphics/-physics intensive video game or a sensor driven GPS enabled mobile application. Sommerville [110] correctly states that the notion of a single method, notation, technique etc. for the creation of such *any* software is pointless. While all need software engineering techniques, they do not need the same techniques!

All software needs Software Engineering, but not all software needs the same Software Engineering techniques.

1.3.1 Spectacular Early Failures

Getting software right is very important, there are many examples of software itself and software projects failing, I've selected a handful of early ones as these are perhaps the best documented.

On the 9th of November 1979 US Strategic Air Command began deploying its forces in response to a report from the Worldwide Military Command and Control Systems (WWMCCS) suggested that the Soviet Union had launched missiles at the US [102]. What actually happened was that test date was interpreted as the real thing, why the system or its users could not distinguish between real and simulated events is unknown [102]. This later inspired a film called WarGames.

In a heavily loaded computer system, a steady stream of high-priority processes can prevent a low-priority process from ever getting resources, this is known as priority inversion. The normal course of events would be that the process would run when the system was quite (e.g. early on a Sunday morning,) however if the system crashed such processes could be lost. Rumour has it that, when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run [107].

The Maiden flight of the Ariane 5 launcher (June 4th, 1996,) lasted around 40 seconds. The $500M loss was caused by an overflow in the conversion from a 64-bit float to a 16-bit int. The bug had been present in an earlier version (from which the component and code was re-used,) but the Ariane 4 (specified some 10 years earlier,) never flew fast enough to trigger this bug.

A full analysis is provided in [118].

The Therac-25 Medical Linear Accelerator (X-Ray Machine,) a software fault known as 'Malfunction 54' essentially the software didn't check often enough that new instructions has been entered. Essentially if the operator changes two settings with the 8 seconds it takes to configure the system then no second check is made on the sanity of the parameters given. One patient received **10 times** the safe radiation dose and died 5 months later. A full analysis is provided in [118].

Most books on Software Engineering have examples of such failures and there are many websites:
Software Horror Stories - Perhaps one of the best referenced.

The financial losses here and the loss of life (potential or actual,) are significant. So what has been done to try and prevent this, the answer is - Software Engineering!

1.4 Evolution of Software Engineering

This section gives an brief (perhaps too brief introduction to Software Engineering, it writing this section I found some interesting things myself. Notably a lot of the 'new' ideas which I though were developed in the mid and late 90's (such as iterative development,) were actually substantially earlier an had been reinvented! In the words of George Santayana "Those who cannot remember the past are condemned to repeat it."

1.4.1 A Brief History

The term 'Software Engineering' was first proposed at a 1968 NATO conference which considered the 'software crisis' [87]. This conference was organised after it became clear that existing approaches to software development did not scale to larger more complex systems and the systems being produced were unreliable, over-budget and delivered late [110]. The original report edited by Naur and Randell (referenced above,) makes interesting reading and some of the problems faced by the industry and the solutions suggested reveal that while we've moved on significantly in the field some practices are much older than we might think!

This quote is from the introduction of [87] and gives the origin of the phrase "Software Engineering": "The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering." The aim was to try and make the creation of software more like other engineering disciplines, at the time these include the addition of design, verification, prototyping and systematic testing. There are also implications for the management of the production process.

Sommerville [109] summarises developments between 1968 and 2000 as follows (I've tried to add references where possible so you can read the original sources if you wish,) and augmented some of the later sections based on other sources.

- **1950s**
  - Evolutionary, itteritive and incremental (IID) development used on the Mercuray space project [49]
- **Late 1960s**
  - 1968 Nato conference aimed at tackling the 'software crisis', introduction of Software Engineering as a term. Digitised version of the report.
  - Seminal note (a kind of short paper in a journal,) by Dijkstra on the dangers of the goto statement in programming.
  - Simula67 developed for simulation, early object orientated language.
  - Trident submarine project uses an IID development method [49].
  - Paper proposes an iterative development method rather than waterfall (published 1968 at the IBM T.J. Watson Research Centre) [49].
- **Early 1970s**
  - Development of the big ideas of structured programming (i.e. sequence, selection and iteration). See
  - Parna's paper on the benefits of information hiding.
  - Development of the Pascal programming language.

- Smalltalk language introduces notions of Object Orientated development.

- **Late 1970s**
  - Early use of software design methods such as Yourdon and Constantine's structured design.
  - Development of first programming environments (dBase 2, OS/VS1, ICCF, ISPF/PDF etc.,) several tools emerge which combined editor with compiler/debugger around the same time.
  - By the end of the 1970s most organisations were using some variation on the waterfall model (see later notes) [102].
  - Some software written with IID methods including USA Space Shuttle flight controls (built in 17 four week iterations) [49].
  - Harlen Mills (a prominent thinker in Software Engineering,) wrote about the failure of waterfall and the need for IID [49].
  - USA Department of Defence adopts the waterfall (DoD-2167) [49].

- **Early 1980s**
  - Development of the Ada programming language, includes notions of structured programming and information hiding.
  - Development of algorithmic approaches to software costing and estimation. More info. - See section 2.
  - CASE (Computer Aided Software Engineering) tools introduced to support design methods. More info.
  - First edition of course text (Software Engineering by Sommerville,) published.

- **Late 1980s**
  - Increased use of OO though languages such as C++ and Objective-C.
  - Introduction of OO design methods.
  - Extensive use of CASE tools.
  - DoD estimates that 50% of its software projects are cancelled or unusable.
  - IDD changes its mind about waterfall and moves to more IDD methods (staring in 1987), however legacy waterfall thinking still confuses some DoD projects [49].

  - Fred Brooks (another big thinker in SE,) publishes *The Mythical Man-Month* and wrote about the shortcomings of waterfall and the need for IID methods.

- **Early 1990s**
  - Object-oriented development becomes mainstream development technique.
  - Commercial tools to support requirements engineering.
  - IID widely accepted recognised as the successor to the waterfall various new methods UP, DSDM, Scrum, XP etc. [49].

- **Late 1990s**
  - Java developed and released (mid 1990s). See very commercial history here.
  - Increasing focus on software architecture, proliferation of client server applications.
  - Notation of component-based software engineering is proposed.
  - UML (Unified Modelling Language) brings together several notations for representing aspects of object-orientated systems.

- **Early 2000s**
  - Use of integrated development tools becomes more common.
  - Use of stand-alone CASE tools declines.
  - Use of UML becomes widespread.
  - Increasing use of scripting languages such as Python and PERL for software development.
  - C# released as a Java competitor.
  - Evolution of the web

- **Mid 2000s** [110, 4]
  - Birth of 'Web 2.0' the read-write web.
  - Shift from dedicated applications to web based applications
  - Introduction of Web Services followed by 'Software as a Service' and cloud computing

- Software more distributed, constructed from re-usable components this in turn demands incremental delivery.

- **Late 2000s** [108]
  - Increased focus on user experience (UX) across a range of devices and the introduction of standards.

1.4.2 Where are we now?

This will be a short section for now, we're obviously going to study what is current though out the course. The big question is have we found the one true way of producing software?

Moving Target Problem

In 1986 Fred Brooks suggested that there was No Silver Bullet, no one technical or managerial development which would keep ordinary projects from becoming monsters [19] (The article was widely re-published the reference is to a later publication in IEEE Computer). His article (which is easily obtained on-line,) makes interesting reading, he notes a large number of "Silver Bullet" candidates some of which had at that point not delivered and others which have not done so since.

More Varied Targets

It is now well established that due to the range of systems being developed that no one size fits all approach is possible, Sommerville [110] divides software into Generic Products and Customised (or Bespoke) Products. The former are usually stand alone and sold on the open market, these include for example PC software such as word processors and computer games. We may also include mobile applications along with so called vertical applications e.g. generic library systems, dental practice software, shop systems, bar systems etc. These are designed for specific purpose but not for a single customer [110]. In such systems the organisation who create the software control the specification [110] working (one hopes,) to fill a gap in the market or to meet needs expressed by a focus group or similar.

Bespoke products are those which are created/commissioned by a customer and usually developed under contract [110]. Control systems for factories and production lines are usually this kind of system along with air traffic

control and any critical system. Here the specification is controlled by the company who ordered the system.

There is some blurring of this distinction more and more systems are based on a generic base adapted for an individual company [110]. Sommerville [110] suggests Enterprise Resource Planning systems are the best example (e.g. SAP,) however a more tangible example for you may be the Student Information System. Another example perhaps is the creation of software under licence, examples include the development of computer games by games companies on behalf of major games producers.

General Movement of the Targets

Software Engineering is essentially a moving target problem we keep improving what we do and how we do it (as detailed in part above,) but in reality the systems we're trying to create are not only different, but changing.

Sommerville 110 suggests that there are general issues which affect all types of software:

1. *Heterogeneity* More varied devices for us perhaps the most tangible examples are mobile, PC, games platforms but realistically in industry we often have to integrate with legacy systems written in old languages. E.g. A student of mine working for a major corporation informed me that they had three current systems the legacy one (written in COBOL,) still held policy data and was actively maintained. Such was the cost of moving the data to a new system management had decided against it hoping that the volume of data (and cost of storage,) would decrease enough that by the time the COBOL programmers retired the cost of moving the data would be less painful.

2. *Business and Social Change* Business and society are changing and very quickly, new markets, emerging technologies etc. who could have predicted the Smart Watch (I did actually, I used it as an example of something unknown my students could face in the future when I started teaching this module). Traditional systems were one big program for one big problem, they were for high risk, hight investment projects - they were slow and document heavy. As we've moved to smaller less expensive to produce systems and large numbers of interconnected programs more flexible methods have become popular.

3. *Security and Trust* As software becomes more ubiquitous its is increasingly important that it can be trusted. This is especially true given trends towards remote systems accessed via the a web based interface. However it is equally true for critical systems, where the cost of failure could be in terms of lives.

A Range of Weapons

Boehem and Turner [16] suggest that development methods exist on a continuum from adaptive to predictive, or agile to plan driven. Both approaches have their homeground [15], i.e. an environment in which they are usually the superior method, but none provides us with Brooke's "Silver Bullet", we'll come back to some of these ideas when we discuss development methodologies in a few weeks time.

My advice is to be pragmatic, there is no one method which you can learn which works for every project, in fact the ones you find in books are probably idealised versions of what really happens anyway. Each project is different and the best approach is dependent on the people (both users/sponsors and development team,) and the project (known vs unknown, stability of requirements etc.).

In summary you need more than one weapon in your armoury, fortunately the key processes remain virtually the same. We always: Find out what is required, plan how we're going to do it, write the code, make sure we followed the plan and that we now have what was required!

Fortunately there are some key process in software development which are common to almost all methods, once you know these applying them (a lot, or a little,) in different methodologies is straight forward. The contrast between the methods and their 'homegrounds' to borrow a term from Boehem and Turner adds to our understanding of the topic.

1.5 Software Process

The systematic approach used in software development is sometimes known as a software process - a sequence of activities which leads to the production of software [110]. Sommerville defines these as [110]:

Software Specification	Where customers and engineers define the software to be produced and the constraints on its operation.
Software Development	Where the system is designed, (the software to be produced is planned out,) and programmed.
Software Evolution	The system is modified to reflected changing customer requirements.

1.5.1 Development Methodologies?

All software development methodologies (which in this context means "a way of developing a software product"[102],) are some combination of the above some organised in different ways and undertaken at different levels of detail [110]. A software process then is a simplified view of the activities in a methodology. We consider these in more detail in future classes (2), however a simple example is instructive at this stage.

1.5.2 Example: Waterfall

Waterfall is the simplest of all software methods, essentially take each of the key processes themselves borrowed from traditional engineering and put them in order. We must note that while this is the simplest its now more of an example used for teaching (like Bubble Sort,) and considered too limited for that vast majority of current real world software problems.

Sahach [102], suggest that until the end of the 1970s most organisations were using as their life-cycle module what is now termed the waterfall model. This has the following six phases [102]:

Requirements phase	Explore and refine the concept, work out what the customer needs/wants.
Analysis phase	Analyse the above, document this in terms which are as explicit as possible to avoid mistakes. Plan the rest of the process.
Design phase	Design the system, this usually entails some notion of Architectural Design (what are the big bits of the system,) and Detailed (or modular) Design (the bits inside the big bits).
Implementation phase	Code and Test (these phases are sometimes separated with the small bits tested during the coding and the whole system tested at the end.
Post delivery maintenance	Maintain (fix bugs,) adapt to change - e.g. decimalisation in the 1970s, the millennium bug etc.
Retirement	Replace the whole system with a new one and retire the old one.

The similarity between this and the common Software Processes outlined by Sommerville in the last section should be obvious also perhaps the similarity to engineering or manufacturing. Schach [102] notes that these names are not standard and the phases were often renamed by individual organisations. He also notes that the Implementation and Maintenance phases have only a temporal distinction, if the client wanted a change before the product was delivered its development the day after delivery its perfective maintenance.

While Schach describes the process very well, he doesn't provide the obligitory diagram of the waterfall model which perhaps gives rise to its name see 1.2.
Given that most projects take many years there is an obvious (I hope,) flaw with this method, i.e. if something changes while the process is on going (there is a change in requirements), we have to go back this takes time and costs money!!

1.5.3 Cost of Finding Flaws?

The cost of finding software flaws rises dramatically with the stage at which they a found [102]. See 1.3.

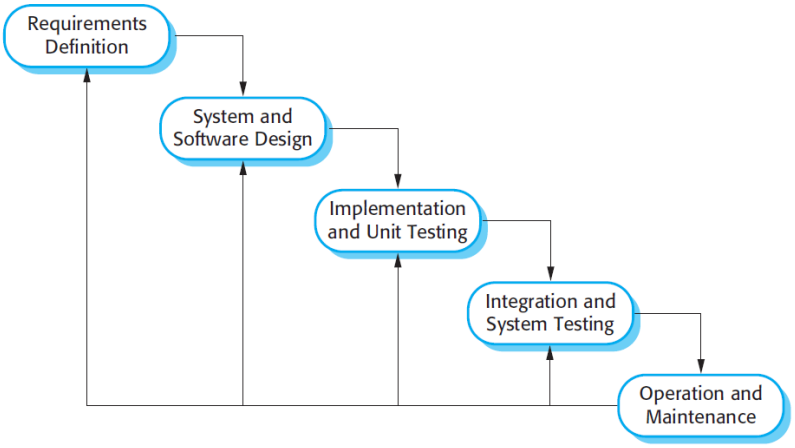


Figure 2.1 The waterfall model

Figure 1.2: from Sommerville [111]: The Waterfall Model

Data from traditional modules shows that this is at best a linear increase in cost, however later data suggests that as projects have become more complex this begins to look more like an exponential increase [102]. See 1.4

1.5.4 So what is Software Engineering?

Having looked at the definitions of software engineering, a littel of its history and some of the background lets re-consider our definition of Software Engineering. [117] suggests that the following characteristics are always present, in Software Engineering.

- **Software Engineering concerns the development of large programs** DeRemer and Kron made the distinction between programming-in-the-small and programming-in-the-large in 1976 [25]. The latter is used to programs written by one person in a short period of time, while the latter refers to multi-person jobs spanning more than six months. Traditional tools and methods were aimed at programming-in-the-small (think flowcharts and structured programming,) these cannot be directly transferred to larger developments.

- **Mastering Complexity** In general the problems we need to solve are complex. This is usually not in terms of what needs to be done, i.e. not all problems involve the solution of complex equations or low level optimisation, but the number of details which need to be considered.

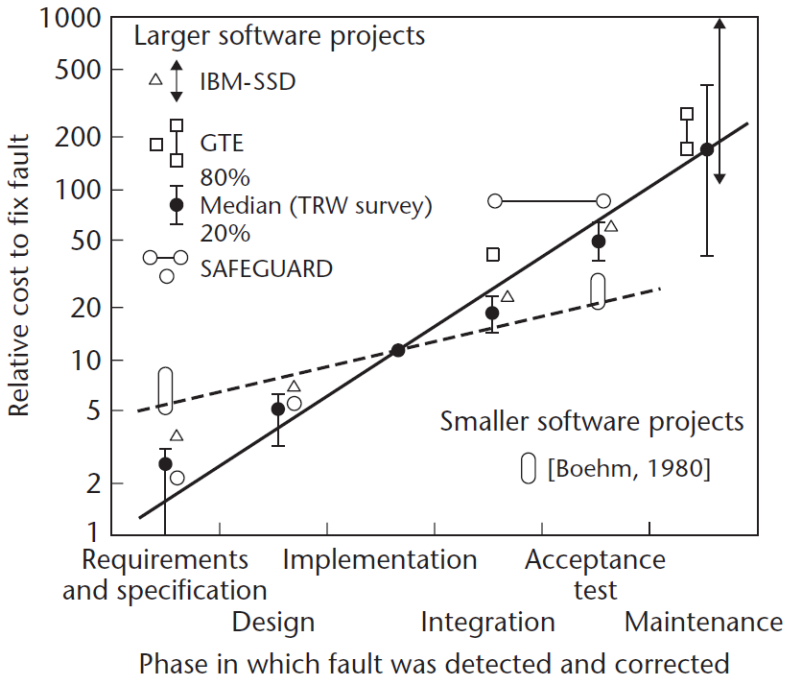


Figure 1.4
The relative cost of fixing a fault at each phase of the software life cycle. The solid line is the best fit for the data relating to the larger software projects, and the dashed line is the best fit for the smaller software projects.
Source: Barry Boehm, Software Engineering Economics, © 1981, p. 40. Adapted by permission of Prentice Hall, Inc., Englewood Cliffs, NJ.

Figure 1.3: from [102].

We seek to master this complexity by dividing it into smaller communicating parts, this contextualises these details and allows parts of the problem to be grasped. An analogy could be the constituent parts of a portable radio, here an amplifier, receiver, tuner, speaker etc., are connected by thin wires. Each component can be designed, constructed and tested in isolation before assembly.

- **Software Evolution** Most software models an element of the real world, e.g. bank transactions, hotel reservations etc., this reality changes over time and if systems are to avoid becoming obsolete they must also evolve. Other software such stand alone applications and games must also change, consider the frequent updates, expansion packs and the increased use of on-line services and on-line play in these applications.

- **Efficiency of Development** Software engineering both as a discipline and a profession constantly searches for new ways to create high quality software quickly. As programmable devices become ever more common demand for software grows, however time-to-market demands also increase as does and the cost and time required to develop software of ever increasing complexity.

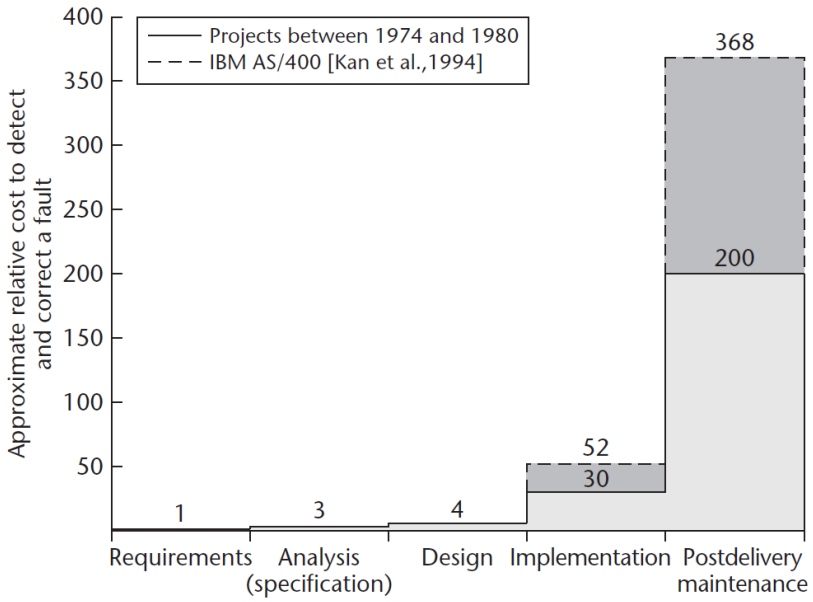


Figure 1.4: from [102].

- **Regular Cooperation** As the problems are large, many people have to work on them concurrently. Teams, often in different geographic locations must work together. The distribution of work, methods of communication, responsibility etc., must be arranged and enforced with standards and procedures. This requires discipline from those involved.

- **Software must be 'fit for purpose'** In the context of bespoke systems, this means ensuring that the system we produce is what the users want. The system must make what they have to do easier/faster etc., or they'll work around it, avoid using it, etc. This may include training users to use the system and ensuring it works in its intended environment. Where software is safety critical (e.g. nuclear power stations, air traffic control etc.,) the software we produce may be responsible for the lives of others. In the vast majority of companies ensuring user/-customer/player satisfaction is vital for the commercial continuation of the product and the company.

- **Software Engineering is a field in which members of one culture create artifacts on behalf of members of another culture** This relates closely to the last two points, software engineering in gen-

eral involves you (an expert in programming, software development etc.,) building a system for a major bank, newsagent, plane etc. It is often the case that software engineers lack the factual knowledge of the domain, but also knowledge of the organisational culture. These may or may not match with what is documented in procedures but remains critical to success. This also makes teaching Software Engineering more difficult, as case studies and coursework will require us to gain a little domain knowledge of a possibly fictional domain to apply that knowledge as we would in a real project.

- **Software is a compromise** In most cases it is futile to assume that the requirements outlined at the start of the project are the only criterion for success. In fact most requirements are negotiable in light of business, technical and political constraints which influence the project. The selection of a particular library of language may by taken based on developer experience (to ensure speed and success of the project,) rather than features. In extreme cases the availability of software components may determine the functionality of the software produced (though arguably this is the tail wagging the dog).

In his introduction to software engineering Vilet [117] states in summary that: Software Engineering is not the same as programming, but programming is certainly part of Software Engineering. Sound engineering practices are required to get useful products. Psychological and sociological aspects play a role in the communication between, human and machine, organisation and machine and between humans. That the development process must be controlled, which is a management issue. Vilet suggests that mathematical aspects play a role in terms of the correctness of software, though this is only really applicable to certain types of software.

1.6 Why do YOU need to know?

Lots of people write programs and it helps to make a distinction between the sort of development that goes on [25] make this distinction in terms of programming in the small and programming in the large. They and others [102] concede that this is not a sharp boundary, but there is certainly a difference between programs written by one person in a short time and those written by many hundreds over a year or so. Schach [102] suggests that traditional tools (such as flowcharts,) and methods such as structured programming (sequence, selection and iteration,) are no longer sufficient for programming in the large. In fact, the term program itself is probably not

adequate to describe the present day software systems which are often a number of (interconnected) programs or components.

You as students spent last year very much programming in the small (think of your introductory programming module - and shudder). The programs are not going to get much bigger this year but we're creeping ever closer to the tipping point where a more professional and considered approach is required.

Sommerville [110] goes a little further suggesting that professionally developed software is that which is intended for use by someone other than the original developer and is maintained throughout its life. Therefore the program is only part of a greater whole, we must take into account the documentation of the software (in terms of its construction and use,) how it will be maintained etc. along with associated configuration data.

While I broadly agree with Sommerville, I would argue that any software you want to be of use to anyone (including yourself,) over a long term needs to be professionally produced. Including the creation of appropriate documentation, having re-written software on several occasions due to maintenance issues this is a painful lesson you can avoid. Next year you'll need to undertake a project and demonstrate you can develop that software professionally, this will hopefully be something which impresses your future employer.

Warning: This section exposes a key problem in teaching programming in the large to students at university, there is no coursework/project that can be set which is guaranteed to cross the line between small and large. In this module you will apply the techniques for a large project to a small project and this will seem like overkill, because it is. **Remember the focus is on learning the techniques for the future not creating software x for the assignment!**

1.6.1 Roadmap for the Module

The (perhaps ambitious) roadmap for the module is as follows:

- Consider the common software processes (analysis, design, implementation, testing)
- Look at the combination of these into Software Development Methodologies (and their underlying general process models).

- Take an agile (iterative) approach to teaching the 'methods' common to a number of methodologies:

First iteration	Just enough to get simple application created essentially run though a simplified software process.
Second iteration	Adding detail, looking at alternative and complementary techniques etc.

- Consider professional issues concerned with software development e.g. project management, ethics etc.

1.7 Tutorial - Definitions and Rationales

Prime Examination Question!
You're task is to split into pairs (or very small groups,) and search for good definitions of Software Engineering and good reasons it continues to develop. You'll then complete based on the following scoring system.

1. **Two** quotes and **two** rational elements, 2 points for each. Each example must be sufficiently documented, i.e. you'll need to correctly quote and reference the definition and reference the rational.

2. For each example over your two - 1 point per example.

3. References for the above:

    - Journals, Conference papers, Books, Published Reports, PhD and MSc Thesis - 2 points for each.
    - Product, university, company web pages - 1 point for each.
    - Blog, personal website, bloke in the pub - 0 points for each.

4. Bonus Points

    - If you are the only group with a given example - 5 points for each.
    - The group with the most recent example - 5 points

The results will be written up and added to this section of the notes.

1.8 Summary

In summary Software Engineering is programming-in-the-large. But even the term program (as in a single software artefact executed by a user) is way out

of date. Software Engineering is an area of study/practice concerned with the managed development of software systems. It is continually evolving to keep pace with changes in technology, software use, software architecture, team and project management, programming etc. but remains focused on the need to develop high quality software, which is fit for a given purpose within a specified time/budget.

Adhoc methods don't scale to large projects, we need 'a way' some method which we follow or work within. This doesn't removed the creative element of programming (us developing a solution to a problem,) but it looks to provide a framework in which the creative efforts of multiple individuals (or the same individual over a long period of time,) can be organised.

1.9 Preparation for Next Week

A nice easy one this week to get you into the spirit of things in year two. I want you to read this chapter:

Sommerville, I. (2011a). *Software Engineering*, chapter Introduction, pages 3–27. Addison Wesley, 9th edition

YOU will need to do the preparatory work ahead of next weeks class!

Models of Software Development

2.1 Introduction

This lesson starts with more key concepts and some important disambiguation of terms, we'll then go on to look at software processes (Sommerville's [111] term for a key activity in software creation,) and in general terms these can be combined. We'll then look at the evolution of software methodologies and where we are now in terms of the methods of choice in industry.

2.2 Terminology

One of the key issues faced when reading about software development is the use (and in some cases misuse) [102] of key terms. This section will introduce these terms with a working definition and some explanation of its use in the field.

- **Program and System** While these terms are often used in modern texts interchangeably a program was previously distinguished as being an autonomous piece of code (at the time a deck of punch cards that could be executed) [102]. A system would consist of a number of such programs and involve mounting tapes for input, running a program, writing the result to another tape and repeating this process until some final program run on the n^th intermediate tape printed a set of reports. The modern definition is that a system is a combination of hardware and software although its origins as collections of software can be seen in such terms as operating system (OS) and management information system (MIS)[102].

- **Product and Artefact** Product is a term often given to a non-trivial piece of software [102]. The term artefact is used to describe any component of this software product e.g. a specification document, a code

module or a manual) [102].

- **Methodology** In Software Engineering the term methodology is used to mean "a way of developing a software product" however, the word actually means the "science of methods" [102]. The confusion here perhaps is that the collection of methods used for example in a scientific paper is often discussed in a section titled methodology.

- **Paradigm** In the 1980s 'paradigm' became a major buzzword in the business world and was adopted by the software industry giving us terms like the object-orientated paradigm and the classical (or structured) paradigm, meaning in this context "a style of software development" [102]. However paradigm actually means model or pattern.

- **Technique and Method** A technique is something applicable to a portion of the software engineering process (for example programming technique, documentation technique etc.,) [102]. The term method is used in a similar context.

- **Mistake, fault and error** When a mistake is made (at any stage in the process,) the result is a fault in the code [102]. When executed this results in failure (the incorrect behaviour of the product as a consequence), the amount of deviation is referred to as the error [102].

- **Software Process** A software process is a sequence of activities which leads to the production of software and is composed of activities [110]. This term is defined and used in Sommerville essentially he means the software development process.

- **Software Process Model** Software Processes can be simplified to a process model (the order in which activities are carried out without the detail,) there are a number of generic models (aka 'process paradigms') which can be used as frameworks to develop more specific software engineering processes [111].

2.3 Software Processes

To recap the previous class, a Software Process is a systematic approach used in software development - a sequence of activities which leads to the production of software [110]. Sommerville defines these activities as [111]:

Software Specification	Where customers and engineers define the software to be produced and the constraints on its operation.
Software Design and Implementation	Where the system is designed, (the software to be produced is planned out,) and implemented (programmed).
Software Validation	The software is validated - checked to ensure it does what the customer asked.
Software Evolution	The system is modified to reflected changing customer requirements.

While all software processes will include these fundamental activities in practice these activities may have a number of sub-activities, for example design usually includes some notion of architectural design (what are the big parts,) interface or mechanistic design (how to they communicate and work together,) and detailed design (the bits inside the big bits which are complex enough to require planning).
In addition to activities a software process may also include [111]:

1. Artefacts/Products produced as a result of a process, e.g. the architectural design phase may produce a number of models showing the architectural design of the system.

2. Roles, as in the things which must be done by people who are involved. For example, during the verification process the code may be inspected requiring a number of readers, a presenter, a chairman etc. Perhaps a more tangible example would include the hierarchy of a programming team, with junior, senior and lead programmer.

3. Pre- and Post-conditions, statements which must be true before and after a stage. For example in the traditional waterfall model all analysis documents had be be completed and signed off before the design could begin. In a more agile developed we may ensure that a customer has agreed and signed off on a set of user stories before beginning an iteration.

Sommerville's process models are an outline for a process and he acknowledges that no two organisations will have the same process, each will tailor their process to the system being developed and the organisation (i.e. the people involved at all levels) [111]. There is no ideal process, today critical systems are developed using very structured developed processes, with everything being checked and signed off in stages and masses of documentation. Business systems in a rapidly changing market, games aiming to offer something novel to players, mobile apps targeting a gap in the market are likely to require something less formal and more flexible.

Which methodology/software process do I need to know?

Those of you of a more pragmatic mindset may be thinking, "which one method do I need to know?". The answer is, nobody knows. Software processes (or methodologies,) like many other facets of computing go though phases and the biggest mistake is often to discard a valid potential solution simply as its out of fashion. The aim in this module is go get you familiar with the key activities, then whatever the process employed in your future work place you'll have a fair idea what's what.

2.4 Process Activities

This section was originally omitted, but I realised that the discussions which followed regarding the arrangement of these key activities in different process models makes no sense unless you understand the detail of these activities.

2.4.1 Software Specification

Software specification or requirements engineering is the process of identifying and documenting what the system should and should not do any any constraints on its operation or development [111]. This is a critical stage, misunderstanding what the customer wants or omitting some crucial detail will result in the development of the wrong system [111] (no amount of good

design/development) can save a project from this!

The requirements are often presented at two levels of detail **User Requirements** which document what the user wants the system / program to do and **System Requirements** which provide detail on what needs to be done to achieve this (the requirements of the requirements) [111].
There are four main activities (see 2.1) [111]:

1. **Feasibility Study** This produces a report which supports the decision to create the software. This may consider weather current software/hardware meets the users needs and the potential benefits in terms of increased revenue or savings. In the case of shrink wrapped software, games etc., this may take the form of a pitch of the initial idea along with some market analysis. There will also be cost estimation for budgeting.

2. **Requirements elicitation and analysis** This is the process of working out what the users/customers actually want. These requirements will then be documented in some way. The techniques used are dependent on the methodology, team and project. They may include observation, interviews, focus groups and the discussion of typical usage scenarios, user stories/journeys/storyboards etc. Techniques for their documentation may be entirely text based, include diagrams or event physical (**WHAT WAS THIS CALLED - FIND IMAGE**). This process may be split over a number of releases/increments with an initial set of requirements being created at the start and continually refined.

3. **Requirements specification** This is the transcribing of the gathered requirements into a document (or other form). These may be divided into **user requirements** - abstract statements of what the system should provide from the users perspective and **system-requirement** - a more detailed description of the functionality.

4. **Requirements validation** Here the requirements are checked for;
   Realism: Can it be done?
   Consistency: Can these things be done together?
   Completeness: Things which are missing, e.g. what should be done if X fails?

These are by no means sequential (even if the methodology has a single requirements phase,) analysis while the process is ongoing and issues raised

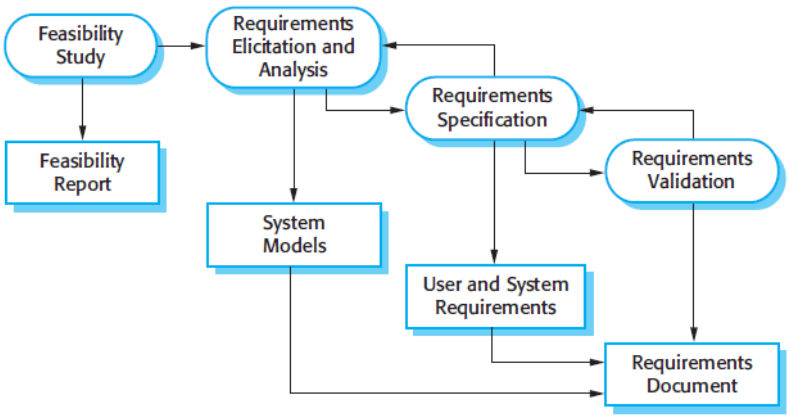


Figure 2.4 The requirements engineering process

Figure 2.1: from [111].

specification (missing details,) or validation (e.g. conflicting requirements,) will require stakeholders are re-consulted [111].
The techniques used vary widely in a traditional development using for example SSADM analysis may be conducted using formal interviews and documented using a standard layout for each user and system requirement. This is still the staple of many industries and may even be conducted up front before being agreed by the customer. RUP makes use of case diagrams (based on scenarios) and supplementary requirements to capture system behaviours (the latter is used to include what is unknown to the user). Scrum and other agile methods have moved to using user stories, which can be elaborated with sub-stories and success criterion to form a basis for development iterations.

A scenario in this context is defined as "a particular complete path though the options available when interacting with a system" [5]. There are many variations but the key remains the same they are a narrative (possibly supported by text, diagrams etc.,) this is more intuitive both customer and developer [5]. Not all authors agree that this is enough detail to create a system, and most approaches use additional documents/diagrams to capture essential details.

Aside: I wrote the above paragraphs after talking to Gordon, he had noted that recent movement in requirements elicitation and analysis had moved from developers writing down what the system would do for the user. Which has an air of technical snobbery to it, "we're experts we know what you want" sort of thing, to approaches which encouraged the developer to get into the shoes of the user (though not in a physical sense).

2.4.2 Software Design and Implementation

This (or perhaps these,) parts of the process involve translating between a specification (what the system should do,) and an executable system [111]. There is always some design and some programming, however if the development is incremental then there may be additional specification and possibly maintenance (re-factoring - changing,) of previously deployed software [111].

The design process is always an iterative one [111] and even where no formal design activities take place someone has a plan (design,) in their head for how the software their writing is going to work. Sommerville 111 suggests that all designers start with a rough outline which is refined with the addition of detail and extensions. He provides an outline of the design process acknowledging that the *Design Activities* are usually heavily interleaved see 2.2.

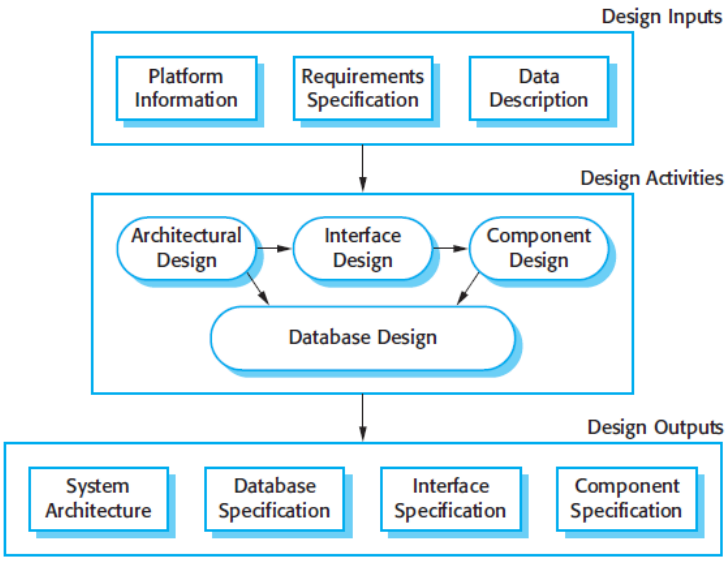


Figure 2.5 A general model of the design process

Figure 2.2: from [111].

This unfortunate needs significant explanation (some of the names are really poorly chosen!), the following explanations are based on the text which follows the diagram [111].

Design Inputs

Platform Information	The software platform consists of other software systems, the operating system, database (if there is one), middleware, frameworks libraries which the software uses. For example if you're writing an application in C# and .Net then the way that framework works is a constraint on your design. This would be different if you were working with the JFC (Java Foundation Classes). This is also true for OpenGL or any other library.
Requirements Specification	Mostly the functionality but possible non-functional demands on the system, e.g. performance, security etc.
Data Description	If the system processes (or stores) data a description of this data will be essential. Other data structures (if used to communicate with other systems,) may also need to be defined.

The design activities are really interesting, I first noticed the pattern highlighted here when I first started teaching different kinds of software engineering. I'll try and highlight how these ideas transfer across seemingly disparate design environments - if it makes not sense tell me I'll try and explain it better!

Design Activities

Architectural Design	The overall structure of the system, the big parts, principle components. These are known as modules (in assembler and old C programs,) sub-systems and components (in object orientated software), here we define them, their relationships and how they are distributed for example client/server.
Interface Design	Here we define the interface for each of the system components, at this stage you can think of this as a list of methods or functions which will be provided. This has to be unambiguous (and there are lots of design principles which attempt to ensure this,) but with the interfaces in place we can in theory develop the components in isolation. We'll look at this under the title Mechanistic Design (a title borrowed from the ROPES methodology).
Component Design	Taking each component and designing its operation, each component may consist of a large number of interoperating entities (functions and structures, objects, etc.). This may be a simple statement of functionality, a list of changes made to a re-usable component or a detailed design. We'll look at this for Object Orientated Systems under the heading Detailed Design (a term borrowed from ROPES).
Database Design	This is where data structures to be stored in a database are designed, this may be a relational database as you've studied in other modules but could equally be an XML file or binary file of some description used to store data.

As Sommerville 111 suggests design is the point at which methods diverge most in terms of what is done. The methods used to develop critical systems will ensure that every element of the design is complete, checked and possibly proved correct mathematically prior to programming. In agile methods the architectural design may exist only as a scruffy diagram on the wall, the interface design as sketches passed between developers, along with automated

tests and the detail design only within the code. There are even those who advocate no design at all, an interesting discussion is presented by Martin Folwer [34].

Sommerville 111 notes that the design and implementation is often interleaved, this is my experience you design, code then update the design when its not quite right. This is also the experience of many other developers I've spoken to. Recently I've spoken to advocates of test driven development who suggest that systems should be correct by test, rather than correct by design. While this gives a functional system there are questions to be asked as to how you evaluate the quality of the system designed?

Programming is a personal activity and there is no general process followed, some start with component they understand best and work towards those they understand least while others start with the most challenging components and leave the familiar components until last [111]. Some programmers will create prototypes of challenging components to test ideas/technologies these may form part of the solution or be discarded. Sommerville 111 notes that all programmers carry out some testing of their programs at this stage, this is not really testing more debugging. Some approaches suggest that test harnesses should be developed along side the components and their execution automated, this is a big part of XP. TDD (Test Driven Development,) goes a step further development actually starts with a test being written which will fail.

2.4.3 Software Validation

Software validation or more generally verification and validation (V & V) is intended to show software conforms to its specification and meets customer expectations. This quote from Boehm (a well respected author on Risk Management in software development,) is one I've found easy to remember:

"Verification : Are we building the product right?

Validation : Are we building the right product?"
[17].

Dynamic testing (where a system or sub-system,) is tested using simulated data is the principle validation technique [111]. However various other techniques exist, a requirements document may be inspected or reviewed as may design diagrams etc., this also applies to code. Static analysis is something

we'll look at later in the year, essentially rather than test the program we examine its source to try and find things which are going to cause problems. Tools to undertake this automatically are coming of age and many projects using automated testing also use automated static analysis - we'll come to this later in the year.

As there are common states of design (architectural, interface, component) which take different names in different approaches the same is true of testing. Sommerville 111 suggests that all but the smallest of programs should not be tested as a single monolithic unit. He advocates the testing of the components of the system as they are constructed, followed by the testing of their integration and finally the testing of the system as a whole. This process will be an iterative one, for example an error found during system testing requires a fix in a component, this then has to be tested by itself and with other components to ensure no new error has been introduced.

The stages outlined by Sommerville 111 are to my mind a little coarse, so I've modified them in line with ISTQB based definitions from here [45]:

1. **Component Testing** Also know as unit, module and program testing. This tests (checks functionality and searches for defects,) in elements of functioning software that a separately testable. This could be an object, more than one object (where they cannot exist alone, e.g. abstract class and a base class,) a group of related functions and structures or a module of assembler.

2. **Integration Testing** Tests the interfaces between components, either internally (between components of our system,) or with external systems e.g. database, operating system etc.

3. **System Testing** concerned with the behaviour of the system as a whole and its emergent properties (speed, stability etc.). This is usually high level testing against the original requirements for the system.

4. **Acceptance Testing** This is testing of the system by the user to establish confidence in the system. The focus here is not finding bugs but ensuring the system is fit for purpose. For contract and in-house software development this is often further divided into operational acceptance testing and compliance acceptance testing. Shrink wrapped software will have on side testing with potential users, alpha testing, and testing in the field, beta testing.

I've avoided going into how we do any of the above, we're going to look at in detail later.

2.4.4 Software Maintenance & Evolution

After software has been delivered to the customer or released it still contains errors (bugs,) which need to be fixed (or patched), unlike mechanical systems these are not caused by wear we are simply repairing hidden defects [117]. Software is more flexible than hardware and changes can be made at any time during or after development for far less cost than corresponding hardware [111]. Historically maintenance was seen as decoupled from development and to some extend a less stimulating task, though the cost and duration of maintenance often far exceeds that of development [111].
Van Vliet 117 identifies four kinds of maintenance activity:

1. **Corrective** maintenance is the process of fixing errors in the software.

2. **Adaptive** maintenance is the adaptation of software to changes in the environment, e.g. changes to the hardware, operating system, back-end database or dependent libraries.

3. **Perfective** maintenance is the adaptation of the software to new or changed user requirements such as extended functionality, improved performance or usability.

4. **Preventative** maintenance is undertaken to increase the systems maintainability, this may include updating the documentation, adding comments, restructuring the code (see re-factoring).

Sommerville 111 suggests that the distinction is becoming increasingly irrelevant, hardly any software systems (or products,) are completely new most replace or add to existing systems. It makes more sense then to think of software as an evolutionary process with its component parts undertaking their own independent life-cycles.

2.5 Software Process Models

The software process can be modelled in a number of ways, Sommerville highlights a number of 'very general process models' [111] these act as frameworks, giving only the ordering of activities without the detail. This provides

us with some means to consider the pros and cons of related families of process model. Van Vliet 119, uses the term generic model in much the same way.
General Software Process Models [111]:

1. **Waterfall Model** Each key activity required in a software process is made into its own phase, e.g. specification phase, development phase, testing phase etc.

2. **Incremental Development** The activities of specification, development and validation are interleaved. A number of versions (or perhaps more appropriately increments,) are produced with each adding functionality. This model has recently been popularised by the Agile software movement, which we'll discuss in more detail in future classes.

3. **Reuse-orientated Software Engineering** This approach requires the existence of a number of re-usable components, development focuses on the integration of these components rather than developing the whole system from scratch. The most recent incarnation and therefore most familiar is Component Based Software Engineering (CBSE,) [23]).

2.5.1 Origin of Process Models

Process models were first introduced after the software crisis the first published was the waterfall model (a plan driven model,) introduced by Royce in 1970 [111]. The original paper should be on Moodle for you, here's the reference:

Royce, W. W. (1970). Managing the development of large software systems. In *proceedings of IEEE WESCON*, volume 26, pages 328–338. Los Angeles.
When reading the documents relating to the NATO conference (1968) which is widely acknowledged as the beginning of software engineering its interesting to note that key ideas such as iterative development were there from the start [87]. Take the following quote (from page 31 of the original notes and page 21 of the transcribed pdf version,) attributed to Kinslow.

"The design process is an iterative one. I will tell you one thing which can go wrong with it if you are not in the laboratory. In my terms design consists of:

- Flowchart until you think you understand the problem.

- Write code until you realize that you don't.
- Go back and re-do the flowchart.
- Write some more code and iterate to what you feel is the correct solution."

Note that Kinslow is still talking about small programs which can be designed by flowchart, however a similar process is advocated for iterations of much larger processes in many (very modern,) agile methods. Interestingly the point he makes here is that this doesn't fit with traditional projects which have strict deadlines [87], a point we'll consider again when looking at iterative methods in general.

2.5.2 Coping with Change

[111] states change is inevitable in any large software development project, requirements change perhaps;

- as the business procuring the software responds to external pressure and managers priorities change
- new technologies become available and development possibilities emerge
- the release of other products changes the target market

Change inevitably leads to additional cost as work that has been completed has to be redone [111]. For example, the addition of a new user requirement will require analysis to identify related system requirements, checks for duplication, conflict etc. Any design which has been completed will need to be updated, any implemented code changed and re-tested.

There are two related approaches which attempt to mitigate rework and its resultant costs, these are change avoidance and change tolerance [111].

Change Avoidance

Here the Software Process includes activities which aim to anticipate changes before significant rework is required, for example prototyping [111]. Prototypes can be developed to demonstrate key features to users to ensure they are meet their exceptions, they can also be used to explore potential solutions (e.g. new technologies or libraries).

Change Tolerance

Here the Software Process is designed to accommodate changes at a relatively low cost, this usually involves an incremental development approach [111]. Proposed changes may be implemented in future increments of the system or where this is not possible as updates to the previously deployed increments. This leads into agile concepts such as refactoring (improving the structure and organisation of a program) which also supports change tolerance.

2.5.3 Evolution of Process Models

Software process models have evolved since they were first introduced after the software crisis. Without going into the detail of individual modules (we'll leave this for future lessons,) we can at least consider two ends extremes of what Boehm and Turner 15 term the continuum of software development methodologies.

2.6 Planned and Agile

This is a brief introduction we'll be looking at agile in more detail next week. Traditional phased development models are to a large extent 'document driven' with milestones attached to documents produced during the project [119]. To elaborate, during the specification phases (often as in this case decomposed into analysis and design phases,) after the requirements engineering there is a requirements specification document. Likewise the design phases results in a technical specification and depending on the methods used design diagrams. The implementation phase results in a set of programs and finally testing results in a test report. Such methods are often called **Plan-driven** because of their large upfront plan of the whole process and **Heavyweight** because of the emphasis placed on the process [119].

"Armour (2001) compares traditional software development with shooting down a Zeppelin, and agile approaches with shooting down a supersonic plane. To shoot down a Zeppelin, we collect information on altitude, distance, velocity and the like, relay this information to the gun, aim, and shoot. This approach does not work for supersonic planes. We do not know where the intercept will be, and the missile will have to change direction while in the air." - Van Vliet 117 describing the analogy of Armour 10.

In many software project change is essential and the requirements of the

project (what the customer wants,) may not be known in advance and so a number of lightweight, or agile methods have been developed [119]. These are designed to avoid expensive planning of the whole system and focus on delivering high value increments of the system as quickly as possible [119]. Feedback from the user can then be used to guide the selection of the next high value increment to be undertaken [119].

Plan Driven Models

The NATO conference broadly concluded that Software Engineering should use the philosophies and paradigms of established engineering to solve what was termed the software crisis [102].

Structure methods for design were developed in the 70s and 80s and advocated graphical models, examples include SSADM these are a precursor to the UML and Object Orientated design [111]. The graphical models produced have various levels of abstraction and in some cases can be used to generate a skeleton for the application.

Though the 1980s and 1990s planned software development, with formalised quality assurance and the use of analysis and design supported by CASE tools along with a rigorous and managed development process were the best way to produce software [112]. This came from a SE community developing large, long lived software systems such as aerospace and government systems. The software was developed by large teams, working over a number of years and often over a wide geographical area [112].

Sommerville 112 is careful to highlight that the overheads associated with such approaches is justified when:

- The work of multiple teams has to be coordinated
- The system is a critical system
- Many different people will be involved in maintaining the project over its lifetime.

However when these approaches (planned, heavyweight) are applied to small and medium sized businesses the overhead is such that it dominates the process [112]. More time is spent in completing, then updating the documentation than producing (and testing,) the software; worse still any change

in the requirements results not only in a re-work of the code but theoretically an update to all the documentation [112].

In my experience, and that of many others I've worked with and taught. These updates hardly every happen except in the largest companies and most critical projects. This is part of the reason that maintaining software becomes so expensive, while the documentation can tell you how it worked when it was developed successive fixes render it useless and days are spend reading pages of code.

An interesting side note is provided by Larman 49, who suggests that while large organisations such as the USA Department of Defence adopted a waterfall standard in the late 1970s and early 1980s (DoD-2167); by the late 1980s it was dropped (due to a large percentage of failed projects,) in favour of an more iterative and incremental approaches. While some of these modules were still very much plan driven they had started to look at incremental (developed in chunks) and iterative approaches (repeat steps to create each chunk,) to overcome the problem of shifting requirements.

During 2004 less than a third of software projects were successful (see 2.3), Schach [102] draws on a number of research papers which show that more than half of projects displayed one or more symptoms of the software crisis.

Figure 1.1
The outcomes of over 9000 development projects completed in 2004.
Source: [Hayes, 2004].



Figure 2.3: from [102].

The failure of planned methods to provide a 'one for all' solution coupled with their failure for so many projects lead to dissatisfaction amongst developers and though the 1990s lightweight alternatives were proposed, eventually giving birth to the 'agile' methods [112].

One observation here is that many of the techniques which planned approaches deploy consist of change avoidance 2.5.2, we aim to avoid having to change the code by being as sure as we can be that we're building the right think before we start.

Change in the market?

While the above sections implies that planned are being replaced by agile they may not have failed, they simply became less successful for the projects they were being used for. But these projects were not what they were created for, that is, the increase in the development of software for small and medium size businesses (an area the heavyweight/planned methods were not intended and had not been recommended for) may also be a contributing factor. The nature of software also changes as more applications are developed which are not replacing established paper based commercial systems. Sommerville 111 hints at this when he considers the need for change in current software development projects.

In short the approaches used though the 70s, 80s and into the 90s were probably the right ones for the kind of software we were developing at the time, however as the nature of software projects has changed.

2.6.1 Agile Methods

Iterative and Incremental Development (IID) approaches were first introduced in the 1950s and are much older than people believe [49]. These can perhaps be viewed as the forefathers of Agile, unlike many planned approaches the software was developed in small increments with each iteration adding to the system and the software growing or evolving to meet a changing specification [49]. Larman 49 notes several large projects which used such approaches including the mercury Space Project in the late 1950s and the Trident submarine process in the early 1960s. By the end of the 1980s the USA Department of Defence had also moved to recommending an iterative and incremental approach.

Agile approaches started to gain traction in the 1990s with the development of approaches such as DSDM, Scrum and eXtreme Programming [112], while the approaches vary the key characteristics are:

1. Specification design and implementation are interleaved (i.e. the process model is an incremental one). There is no detailed system specification and design documentation is minimal or generated from the code. The user requirements document only the most important characteristics of the system.

2. The system is developed in a series of versions (increments). These are specified and evaluated by end-users and other interested parties, they propose changes and new requirements for later versions.

3. System user interfaces are often developed using interactive developed systems which allow the user interface to be produced via 'drag and drop', alternatively this may be created by a designer in some mark up language (QML, XAML, etc.) and the domain of specialist UX designers.

The story of Agile, both as a name and a concept, is a rather strange one, seventeen software developers (later dubbed the Agile Alliance) met at a Utah ski resort for two days in Feburary 2001 and produced the *Manifesto for Agile Software Development* [103]. Key participants included Beck (creator of Extreme programming,) Cockburn (Crystal) and Schwaber (Scrum) [103]. Rather than outline a specific software process they produced a set of underlying principles which were common to their individual approaches [103]. The manifesto states:
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

That is, while there is value in the items on the right, we value the items on the left more. (from Sommerville 112).
Individual methods vary quite widely, while all follow the above common principle each has different processes to achieve this [112]. The above is principles are quite cleverly worded and in each case something is valued

over something else, but it is simply considered of more value. The items on the right are still valued.

Agile methods are recommended [102] smaller projects, with where rapid change is likely and the number of developers is small (around 50 people). Plan-plan driven approaches are well suited to very large projects and those where the requirements are well known in advance and not subject to change [104]. This may seem far too simpler distinction and the reality most projects employ a mix of both [15].

Here we observer the use of incremental development to tolerate change (see 2.5.2) unlike the attempt to avoid change by ensuring everything was known before hand. Van Vliet 119 suggest that such models take into account that much of what is called maintenance is really evolution and therefore anticipated change which should be accounted for in the model.

IMPORTANT: Note that there is no 'one process to rule them all' each approach is best suited to a particular subsection of projects.

A note on maturity

Technology developed in research laboritories takes some time to become commonly used, this holds for physical innovations (e.g. transistors,) but also tools and techniques [102]. Examples (again from [102]:

- **UNIX** Original developed in 1971, only widely used since the 1980s.
- **Programming** Dijkstra's seminal article on programming appeared in 1965, however the first programming textbooks weren't introduced until the late 1970s.
- **Software Engineering** The term arrived in 1968, but courses (programs of study at universities/colleges) didn't arrive until the 1980s.
- **Software cost estimation** Methods for estimating software cost were first studied in the 1960s but were re-discovered in the 1980s.

In short there is a maturation process, typically from these examples between 10 and 15 years over which technologies become widely accepted. This is probably just coming to an end for Agile methods as a rule.

2.6.2 Home Grounds

Boehm and Turner 15 introduce a concept called home grounds, types of project where either Agile or Plan-drive methods are most likely to succeed. They also suggest that software processes (they use the term 'methods') exist on a continuum between the most plan-driven (e.g. Waterfall) and the most Agile (e.g. Extreme Programming). Their paper [15] shows the five critical decision factors they suggest for identifying which projects require planned and which require agile methods.

Table 1. in 2.4 is for the most part fairly self explanatory, note that the staffing of the project suggests that projects using Agile methods require a higher proportion of senior developers. This is something I've encountered when discussing projects with former students, many companies are actually buying in 'mentors' while they develop their own.

Another key observation (and one we'll come back to in the next class,) is that projects which one or another are very few, the majority of projects are somewhere in-between and require some mix of agile and planned approaches [15]. The most tangible example perhaps is where a very large software product is being created, some planning may be required to identify the scope of the system and to ensure that major parts (possibly developed by large teams,) have their boundaries identified and interfaces (along the the process of changing them,) agreed.

Before looking at Balanced Methods [15], we're going to consider some examples of plan drive approaches in more detail. Then next week some Agile approaches in detail, then come back to Balanced Methods.

2.7 Plan for this year

My plan for this year is to introduce key elements of both approaches, the structure is going to be an iterative one. We'll start by looking at some basic techniques for the development of a system including basic techniques for each for the following.

- **Analysis** Capturing the requirements of the system/iteration, traditionally undertaken using requirements analysis. In the initial iteration

Table 1. Agile and plan-driven method home grounds

Characteristics	Agile	Plan-driven
Application		
Primary Goals	Rapid value; responding to change	Predictability, stability, high assurance
Size	Smaller teams and projects	Larger teams and projects
Environment	Turbulent; high change; project-focused	Stable; low - change; project/organization focused
Management		
Customer Relations	Dedicated on-site customers; focused on prioritized increments	As-needed customer interactions; focused on contract provisions
Planning and Control	Internalized plans; qualitative control	Documented plans, quantitative control
Communications	Tacit interpersonal knowledge	Explicit documented knowledge
Technical		
Requirements	Prioritized informal stories and test cases; undergoing unforseeable change	Formalized project, capability, interface, quality, forseeable evolution requirements
Development	Simple design; short increments; refactoring assumed inexpensive	Extensive design; longer increments; refactoring assumed expensive
Test	Executable test cases define requirements, testing	Documented test plans and procedures
Personnel		
Customers	Dedicated, collocated CRACK* performers	CRACK* performers, not always collocated
Developers	At least 30 percent full-time Cockburn Level 2 and 3 experts; no Level 1B or -1 personnel**	50 percent Cockburn Level 3s early; 10 percent throughout; 30 percent Level 1Bs workable; no Level -1s**
Culture	Comfort and empowerment via many degrees of freedom (thriving on chaos)	Comfort and empowerment via framework of policies and procedures (thriving on order)

* Collaborative, Representative, Authorized, Committed, Knowledgable
** See Table 2. These numbers will particularly vary with the complexity of the application

Table 2. Levels of software method understanding and use (after Cockburn)

Level	Characteristics
3	Able to revise a method (break its rules) to fit an unprecedented new situation
2	Able to tailor a method to fit a precedented new situation
1A	With training, able to perform discretionary method steps (e.g., sizing stories to fit increments, composing patterns, compound refactoring, complex COTS integration). With experience can become Level 2.
1B	With training, able to perform procedural method steps (e.g. coding a simple method, simple refactoring, following coding standards and CM procedures, running tests). With experience can master some Level 1A skills.
-1	May have technical skills, but unable or unwilling to collaborate or follow shared methods.

Drawing on the three levels of understanding in Aikido (Shu-Ha-Ri), Alistair Cockburn has identified three levels of software method understanding that can help sort out what various levels of people can be expected to do within a given method framework [2]. We have taken the liberty of splitting his Level 1 to address some distinctions between agile and disciplined methods, and adding an additional level to address the problem of method-disrupters.

Level -1 people should be rapidly identified and found work to do other than performing on either agile or disciplined teams.

Level 1B people roughly correspond to the "1975-average" developer profile. They can function well in performing straightforward software development in a stable situation. But they are likely to slow down an agile team trying to cope with rapid change, particularly if they form a majority of the team. They can form a well-performing majority of a stable, well-structured disciplined team.

Level 1A people can function well on agile or disciplined teams if there are enough Level 2 people to guide them. When agilists refer to being able to succeed on agile teams with ratios of 5 Level 1 people per Level 2 person, they are generally referring to Level 1A people.

Level 2 people can function well in managing a small, precedented agile or disciplined project but need the guidance of Level 3 people on a large or unprecedented project. Some Level 2s have the capability to become Level 3s with experience. Some do not.

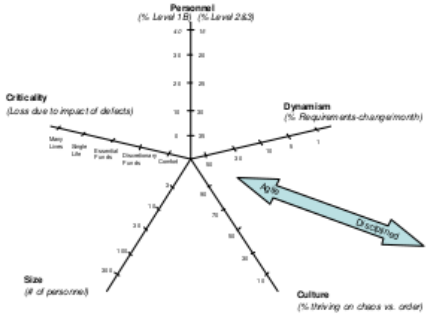


Figure 1. Dimensions affecting method selection

Figure 2.4: A collection of figures/tables from [15], showing the home grounds of agile and planned methods.

we'll look at this and requirements modelling techniques. Later iterations will hopefully add user stories and other contemporary techniques.

- **Design** We'll look at design in enough detail that you'll be able to read the diagrams and use them to sketch out a plan of a solution before implementing. Even this will be enough exposure to show how difficult

a complete up front design of a system is to create. I'd like to look at patterns here, but this may have to wait for a later iteration.

- **Implementation & Testing** The techniques here will be mostly those concerned with multi-developer projects e.g. managing source code and for iterative development e.g. automated testing (important in both planned and agile methods in different ways).
- **Professional Issues** There are management, team, ethical and professional issues we must be aware of. To qualify for the BCS accreditation you need to have studied these. They are an essential (if rather dry,) part of the education of all software engineers.

2.8 Tutorial - Planned Or Agile

You're task is to split into pairs (or very small groups,) and search for development methodologies. I'll give you two to get you started, Waterfall and eXtreme Programming.

You'll then complete based on the following scoring system.

1. Correctly classification - 1 point per example.
2. Rational, i.e. arguments for your selection - 1 point each (up to a max of 3).
3. Additional methods over those provided - 2 points per example.
4. Rational - 2 points each (up to a max of 6).
5. References for the above:
    - Journals, Conference papers, Books, Published Reports, PhD and MSc Thesis - 2 points for each.
    - Product, university, company web pages - 1 point for each.
    - Blog, personal website, bloke in the pub - 0 points for each.
6. Bonus Points
    - If you are the only group with a given example - 5 points for each.

2.9 Summary

In summary there are a large number of development methodologies which range between planned and agile, both have their strengths and weaknesses for different kinds of project. The activities/practices within the life-cycle models are in some instances the same and in others very similar.

This module will look at as many of these activities and practices as possible in the time in the context of an agile like methodology (based on UP-/ROPES,) which I've devised to expose you to as many of the activities involved in creating software within our limited time.

2.10 Preparation for Next Week

Essential reading for next week:

Sommerville, I. (2011b). *Software Engineering*, chapter Software Processes, pages 27–55. Addison Wesley, 9th edition.

Van Vliet, H. (2007c). *Software engineering: principles and practice*, chapter The Software Life Cycle Revisited, pages 45–77. Wiley New York.

Optional reading:

Sections 1.3.1 and 1.3.2 from:
Schach, S. R. (2007a). *Object Orientated Software Engineering*, chapter The Scope of Object-Orientated Software Engineering, pages 3–31. McGraw-Hill, New York, USA.

YOU will need to do the preparatory work ahead of next weeks class!

Ballencing Agility and Discipine

3.1 Introduction

3.2 Agile and Planned

See also: 2.6.1 & 2.6

Traditional phased development models are to a large extent 'document driven' with milestones attached to documents produced during the project [119]. Such methods are often called **Plan-driven** because of their large up-front plan of the whole process and **Heavyweight** because of the emphasis placed on the process [119].

"Armour (2001) compares traditional software development with shooting down a Zeppelin, and agile approaches with shooting down a supersonic plane. To shoot down a Zeppelin, we collect information on altitude, distance, velocity and the like, relay this information to the gun, aim, and shoot. This approach does not work for supersonic planes. We do not know where the intercept will be, and the missile will have to change direction while in the air." - Van Vliet 117 describing the analogy of Armour 10.

Change is essential in most modern software projects and the requirements of the project (what the customer wants,) may not be known in advance and so a number of lightweight, or agile methods have been developed [119]. These are designed to avoid expensive planning of the whole system and focus on delivering high value increments of the system as quickly as possible [119]. Feedback from the user can then be used to guide the selection of the next high value increment to be undertaken [119].

3.3 Examples - Plan Driven Approaches

3.3.1 Waterfall

This is a summary of Sommerville 111 (section 2.1.1) and [103] (section 2.9.2). I recommend you read both these sections.
The waterfall model of software development (aka Software life cycle) was derived from more general system engineering processes and gets its name from the cascade of artefacts and effort from phase to phase [111]. This is a plan driven process with each phase being planned before execution [111] and no phase being complete until the associated documentation is complete and approved (by the Software Quality Assurance (SQA) group) [102]. Testing is inherent in each phase (and in theory,) continuous [103], this testing is often described in other terms e.g. 'V & V' verification and validation. For example during the requirements gathering phase the requirements may be checked for duplication, conflict etc., during design the communication between classes may be checked via role play (but we don't call it that!)

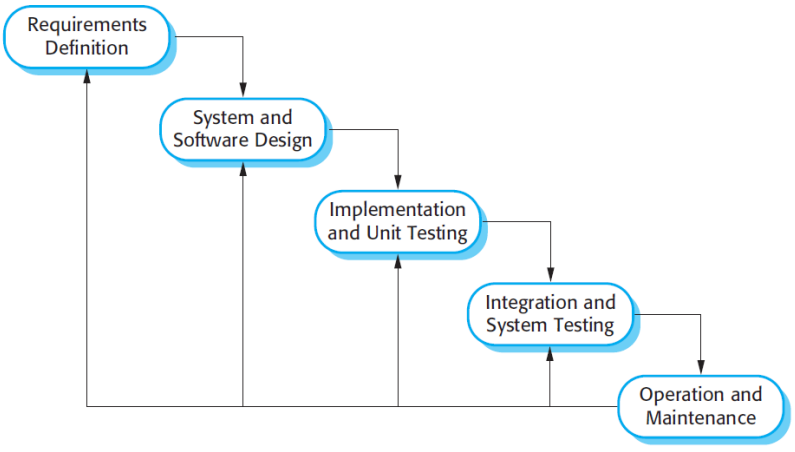


Figure 2.1 The waterfall model

Figure 3.1: from Sommerville [111]: The Waterfall Model

1.2 shows the principle stages, these directly reflect the principle development activities:

1. "*Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

2. *System and software design* The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. *Implementation and unit testing* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. *Operation and maintenance* Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered."

[111, p.30]
Conceptually each phase is checked and signed off (see 3.3.1,) but in practice there is some feeding overlap between stages [111]. The design process exposes omissions or conflicts in the analysis document, or during development a error is found in the design and this needs correcting. The cost of such iterations is significant and as a result earlier stages are often frozen and worked around [111]. During operation and maintenance the system continues to develop (errors and omissions are corrected and the system evolves to remain useful,) this requires more iterations [111].

Advantages

1. Progress is visible (from the production of documents) and can be compared to the plan [111].

2. If hardware etc. is being developed alongside using a traditional engineering approach waterfall allows for a common management model [111].

3. Documentation may be useful for long lived systems which will be maintained over very long periods.

Disadvantages

1. Partitioning into stages leads to inflexibility, commitments (key decisions,) must be made at an early stage making it difficult to respond to changing requirements [111].

2. Repeating stages is costly [111].

3. Documentation must be written and kept up to date, this requires a significant time commitment.

Larman 49 suggests that while there is no one reason so many waterfall based projects failed evidence from a study by Boehm and Papaccio showed that typically 25% of requirements changed. He also highlights a second study which suggests that this figures could actually be between 35-50% for larger projects (and increase linearly with code base size).

Aside: Formal Systems Development

Formal Systems Development or Formal Methods are a branch of software development where mathematically based tools are used for the synthesis (i.e. development,) and analysis of software systems [48]. Formal development processes are most often used where there is a stringent safety, reliability or security requirement and is often mandated by customers or regulators [111].

Rational for Formal Methods

The use of Formal Methods is motivated was initially motivated by the expectation that the success of mathematical modelling in other engineering disciplines could be transferred to software [48]. Though Formal Methods, like so many other things in Software Engineering, can be traced to the Software Crisis [13]. There is a suggestion that while methods for managing software project and the development of sound processes continued to develop, mathematical modelling was shunned [13]. Banach 13 postulates that this may have been the result of an influx of developers without mathematical backgrounds or as a result of perceived complexity (arguably CASE tools such as UML and automated development tools have surpassed this). He goes on to suggest that there was also over selling of formal methods, which were often unused resulting in bad press.

Many of the most serious problems related to safety critical systems relate to deficiencies in their specification [116]. Specifications should be complete, unambiguous, consistent and correct which is difficult to achieve with written

documents [116]. Demonstrating that the result is either complete, correct or even consistent is also prone to error as it relies on another person understanding both the requirement and the specification [116]. Storey 116 states that CASE tools and graphic specification languages (e.g. UML) are often referred to as formalised methods (rather than formal,) which aim to provide a systematic approach to system specification. He believes that these provide a partial solution but are limited by a reliance on engineers to correctly apply these techniques (e.g. the right arrows in UML) and interpret their output (e.g. implementing the diagrams, or checking the use case against the requirements). Rushby 101 goes further stating that the precision is illusory if there are no semantics giving strict meaning to the diagram. Consider UML for example with its various different kinds of arrow, each with a precise meaning.

Formal methods are defined using Formal Specification Languages, these have very precise rules which allow the language to be interpreted unambiguously they can also be checked automatically (using tools,) for certain omissions and inconsistencies [116]. This is one of the greatest advantages to the use of formal methods, however it is also possible to use software tools to check for functional equivalence between descriptions of the system [116]. As the requirements, specification and design (at various levels) are all descriptions of the system it is possible to prove that these are equivalent (i.e. verification between stages) [116].

Haxthausen 48 notes that the use of Formal Methods does not guarantee correctness, but increase the level of correctness and improve flaw/fault detention in the specification and design phases. You may recall from your other Software Engineering modules that this is a key factor in reducing costs, as the further along in the development process problems are found the more they cost to fix [122]. This is echoed by Hall 47 who suggests that the use of Formal Methods for error reduction in the early stages offers the most tangible benefits to their use. Leveson 73 notes that neither testing or formal specification will make a system safe (only safer). Testing is limited as exhaustive testing is impossible. Formal verification is becoming more possible but the development of the mathematical models required is as difficult and error prone as writing the code itself. Finally she notes that failure is more often caused by problems with the requirements/specification and not the code developed.

3.3.2 Spiral

This is a summary of Sommerville 111 (section 2.3.3), I suggest you read this and possibly the original paper by Bohem 18.

Bohem 18 proposed the spiral model in 1988 , this risk driven which each loop of the spiral representing a phase of the software process [111], this is show in 3.2. The focus of the loops progresses from inside to out as follows, (innermost) concerned with system feasiblity, the next requirements definition, then system design and so on [111]. This combines elements of changes avoidance with change tolerance and assumes changes are a result of project risks and that risk management with each spiral can reduce these [111].

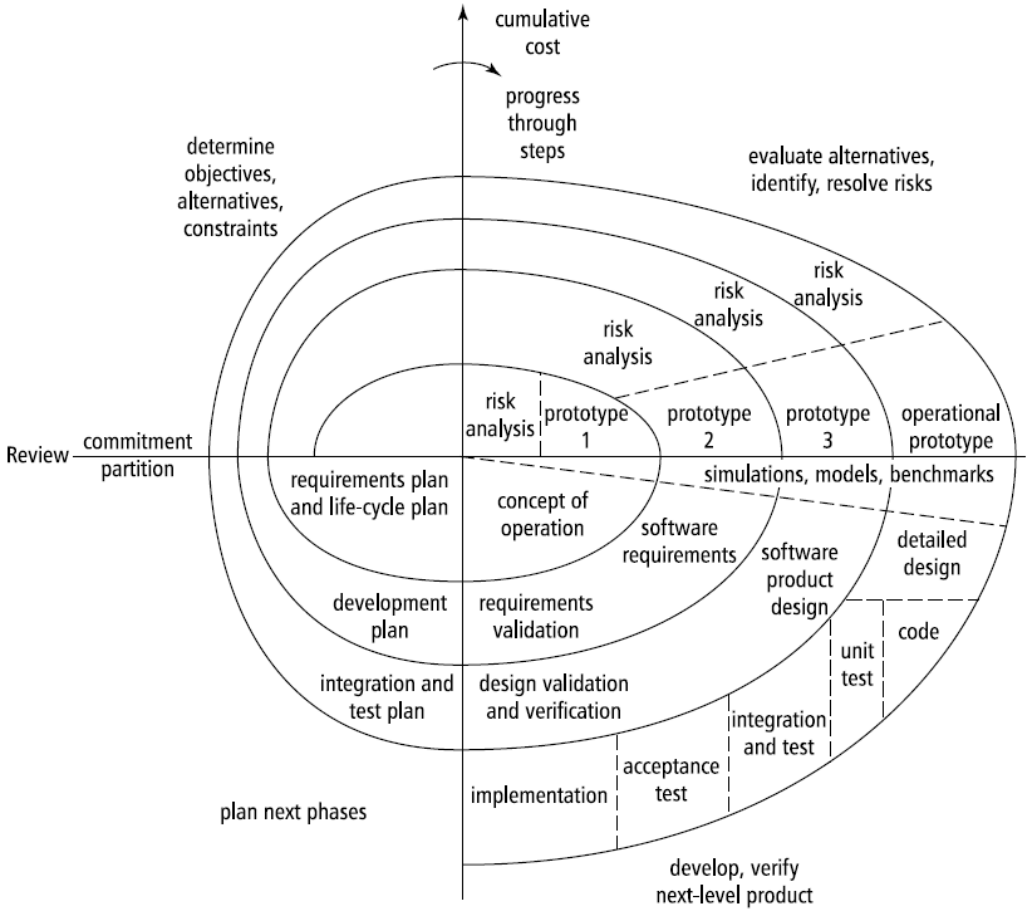


Figure 3.2: from Sommerville [111] The Spiral Model

The sectors of the spiral are outlined by Sommerville 111:

1. *Objective Setting* Set objectives for the phase, plans for phase management, constraints, risks and possible alternatives.

2. *Risk Assessment and Reduction* For the risks identified (from the last sector,) a detailed analysis is carried out and steps taken to mitigate the risk, e.g. if the requirements are poorly defined or have conflict a prototype may be developed.

3. *Development and Validation* Based on the risk evaluation a development model is chosen, this may be throw away prototyping if the UI/UX is a key risk, or if there are technological concerns. However it is not possible to mitigate the risks in all areas using a prototype, e.g. assessing the competence of the team [103]. Formal methods may be employed if safety is the driving factor and a waterfall like design (i.e. a big up front design, starting with the architecture,) if subsystem integration is the main risk.

4. *Planning* The project is reviewed and a decision made regarding the continuation of the project and planning for the next phase.

Schach 103 argues that Boehm original model is really little more than a modified waterfall which allows for iteration and has a significant risk analysis at the start of each stage. This is illustrated in 3.3
Van Vliet 119 observes that the spiral model can be simplified to the following activities:

- identify the sub-problem which poses the greatest risk to the project.
- find a solution

It can therefore be coupled with other models [119]:

1. If the need to obtain accurate user requirements is seen as the greatest risk, then the spiral can be followed multiple times to solve this problem. This may include developing prototypes to be shown to users or to test possible development solutions.

2. If a full requirements specification is available the spiral can be followed in once using the corresponding milestones (phase endpoints) as intermediate steps.

3. If incremental development is employed the spiral can be used per increment (iteration).

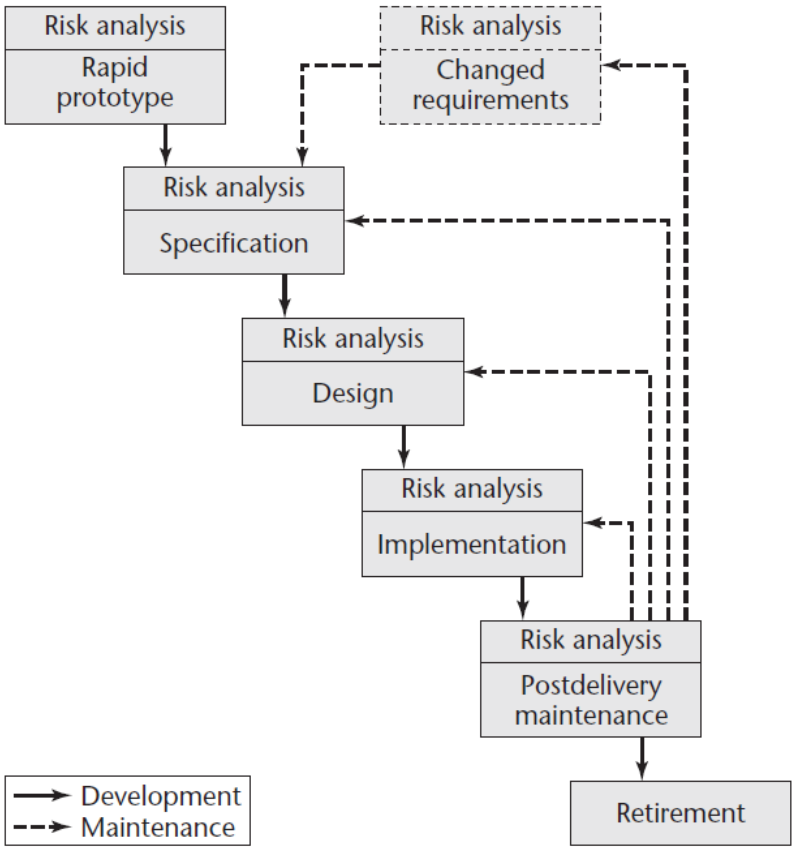


Figure 2.12
A simplified version of the spiral life-cycle model.

Figure 3.3: from Schach [103]: Spiral is modified waterfall

4. A spiral can be followed for each error reported after the delivery of the project (traditional maintenance) ensuring the repair is developed to the same standard as the rest of the code.

The spiral pattern has been borrowed in other approaches, take for example ROPES 3.8.2.

Advantages

1. High risk parts of the project can be overcome early on (or the project abandoned) [103].

2. Intended for large projects and has been used in a large number of projects and produced and shown to increase productivity [103].

3. Maintained becomes another cycle there is no distinction between this and development, this prevents it becoming a second class activity

[103].

Disadvantages

1. Not all risks can be mitigated, e.g. team performance, hardware promises etc. [104].
2. While in-house projects can be easily cancelled if the risks become to great, this may not be possible in contract based software development [103].
3. Risk analysis is costly and requires skilled personnel [103].

Larman 49 identifies the introduction of the risk driven spiral as a critical phase in software development. He suggests that the spiral facilitates iterative and incremental delivery.

3.3.3 Code and Fix

The code and fix life-cycle is perhaps the most often deployed model in a university context and is used in industry [103]. The product is implemented without any specified requirements and with no design, instead code is thrown together and re-written as required to satisfy the client.

Advantages

- Speed.

Disadvantages

- Any change is very expensive to implement as the code will need to be changed [103].
- Maintainance is difficult and error prone [103].
- The above are complicated by the lack of design documentation [103].
-

While this approach can be safely used with small projects (100-200 lines), it is unsatisfactory for projects of any size [103]. One of the key difficulties with studying Software Engineering is that few coursework really require more than a few hundred lines of code and while many approach the limit of

build and fix none but the major project really exceed it.

This model is unfortunately popular especially in companies who measure progress by the number of lines of code written as developers are encouraged to start writing code on day one of the project [103].

3.3.4 Brief History of Agile

Agile approaches started to gain traction in the 1990s with the development of approaches such as DSDM, Scrum and eXtreme programming [112], while the approaches vary the key characteristics are:

1. Specification design and implementation are interleaved (i.e. the process model is an incremental one). There is no detailed system specification and design documentation is minimal or generated from the code. The user requirements document only the most important characteristics of the system.

2. The system is developed in a series of versions. These are specified and evaluated by end-users and other interested parties, they propose changes and new requirements for later versions.

3. System user interfaces are often developed using interactive developed systems which allow the user interface to be produced via 'drag and drop', alternatively this may be created by a designer in some mark up language (QML, XAML, etc.) and the domain of specialist UX designers.

The story of Agile methods is a rather strange one, seventeen software developers (later dubbed the Agile Alliance) met at a Utah ski resort for two days in Feburary 2001 and produced the *Manifesto for Agile Software Development* [103]. Key participants included Beck (creator of Extreme programming,) Cockburn (Crystal) and Schwaber (Scrum) [103]. Rather than outline a specific software process they produced a set of underlying principles which were common to their individual approaches [103].

The manifesto states:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions over processes and tools*

- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

That is, while there is value in the items on the right, we value the items on the left more. (from Sommerville 112).
Individual methods vary quite widely, while all share the above values each has different practices/methods to achieve this [112]. These are summarised in 3.4 from [112].

Figure 3.1 The principles of agile methods

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Figure 3.4: from [112].

Agile methods are recommended [102] smaller projects, with where rapid change is likely and the number of developers is small (around 50 people). Plan-plan driven approaches are well suited to very large projects and those where the requirements are well known in advance and not subject to change [104].

Agile or Not?

Bob Martin, (also known as Uncle Bob and the author of some excellent books,) suggests that the enthusiastic adoption of new practices by one group creates tension in those who do not adopt [74]. They may feel that their good work before the new practice is somehow devalued. He also comments on the "Corruption of Agile", where practice and culture are not only divorced but eroded. Martin 74 argues that agile culture is expressed though a set of practices and the erosion of these practices is has diluted the culture. Folwer 37 has noted similar issues when agile practices are adopted in part without the

key cultural aspects. In this case the lack of mandated technical practices in a chosen agile methodology has lead to a lack of such practices and the failure or projects. The missing cultural aspect is - Individuals and Interactions are more important than Process and Tools.

Folwer 36 uses the term Semantic Diffusion to describe a process where an idea / concept etc. loses its meaning over time. He suggests that more concrete entities such as tools and agile methods with firm practices such as XP are least likely to suffer, unlike more conceptual entities e.g. Agile and Web 2.0.

A key observation in Martin 74 is that while culture is expressed through practice the culture should not be lost. For example a developer writes a test for each method he writes, but only one and with no effort being made to really exercise the code in the method.

3.3.5 Key Characteristics of Agile Methods

Agile methods display the following characteristics [119]:

1. Users involved at every step.
2. Development cycles are small and incremental.
3. Detailed plans for the overall development are not created in advance. However after each development cycle the current situation is reviewed and the next cycle planned.
4. At the end of cycle there is a running system. This is not a prototype (which suggests something intermediate).
5. No extensive architectural, mechanistic or detailed design up front. Design is undertaken but only as far as the next step. As a result a separate activity *Re-factoring* is employed to improve the code base (often the design,) between cycles.
6. A strong emphasis on people, i.e collective intelligence, team spirit, team ownership etc. The teams are usually in a single big room (rather than offices/cubicles and the users on site. Communication cycles are also short (meet users a lot).
7. Not much time spent on documentation, diagrams are used as tools to understand and discuss the problem. No upfront designs which will

have to change, however the design may be documented (hopefully automatically) once it's developed for maintenance.

3.4 Almost Agile?

There are a number of approaches which have some but not all of these characteristics. Prototyping and Incremental development both recognised the volatility of the problem space and that a plan-drive approach doesn't always work [119]. Rapid Application Development and DSDM emphasize customer collaboration and the role of people in the process [119]. Finally XP is a 'pure' agile method.

3.4.1 Protoyping

Prototyping is not uncommon in other kinds of product, for example cars here however the cost of making one off is much less than setting up the production line to undertake a production run. However once the software is designed and built producing many copies is simple, the expensive and difficult work has been done [119]. Prototypes are often used to explore key features of the problem space or the technologies available [119].

Prototyping can work well with interactive systems, users can see how the system will work and feel involved development, prototyping here becomes a tool to elicit requirements [119]. in *throw away prototyping* the roughly constructed code is disposed of and a system is designed and coded [119]. In my experience such prototypes can also be used by developers to make technical judgements, e.g. which graphics library to use etc.

Throwing away 'working' if poorly written code is often difficult for developers under pressure. There are also models where the user starts working with the prototype and suggests improvements and features for development, this is termed *evolutionary prototyping* [119]. The issue here is one of design, the code has been quickly put together and the layers of changes often conflict. The result is a hard to maintain code base [119].

There are some positives however, the users and developers take a lot of ownership of the system and there is a lot of communication [119]. Agile methods solve this problem with re-factoring, essentially going back and improving each development cycle to ensure the design remains robust [119].

3.4.2 RAD

Rapid Application Development (RAD) like many other iterative methods emphasises customer involvement, prototyping, reuse, automated tools and small development teams [119] RAD introduced the notion of a *time box* a fixed amount of time in which to implement some requirements, if this was not realised then some functionality is sacrificed [119]. Another key feature of RAD is the Joint Requirements Planning workshop, joint in that this involves both users and developers [119]. This process is know as *triarge* (a process usually used acute care to sort injured people into those most in need of help) and often uses MoSCoW [119]:

- **Must** have requirements - definitely need.
- **Should** have requirements - important but not absolutely necessary for a usable system.
- **Could** have requirements - only if time allows.
- **Wont** have requirements - left for the next iteration.

There are usually two such workshops during the design phase, the first leads to a prototype which is evaluated in the second [119]. RAD then calls for a small focused SWAT (Skilled With Advanced Tools,) team to work on developing the system, for details see: [119, 120]

3.4.3 DSDM

DSDM (or Dynamic Systems Development Method,) is build on RAD and based on nine key principles, the framework itself not for profit and is maintained by the DSDM Consortium [119]. While the framework is well documented you have to join the Consortium to gain access to the complete set of DSDM practises. The DSDM has five phases outlined briefly below [119]:

- **Feasibility Study** - Dual focus, can we build the system and is DSDM a feasible approach. For DSDM to work we must be able to identify the users of the system, it can't be too large and requirements not known up front.
- **Business Study** - Generate high-level descriptions of the business processes using facilitated workshops along with high level architecture.
- **Functional Model Iteration** - Analysis models, prototypes and implementation of the major funcitonal components. Each iteration is

time-boxed for two to six weeks and consists of four activities: identify what to do, agree how to do it, do it and check its done.

- **Design and Build Iteration** - engineer system to a high standard. Emphasis should change here to developing a high quality, well engineered system rather than focusing on what to build and how. This difference is not always easy to make.

- **Implementation Phase** - System is deployed in customer environment and user training takes place.

Van Vliet 119 suggests that these methods do not display enough of the key characteristics of agile methods to be consider agile.

3.5 Examples - Agile Methods

3.5.1 Extreme Programming

Extreme Programming or XP for short is a pure agile method [119], it importance is such that I recommend you reading up on it in some detail. The relevant sections are:

- Van Vliet, H. (2007c). *Software engineering: principles and practice*, chapter The Software Life Cycle Revisited, pages 45–77. Wiley New York - Section 3.2.4

XP takes a number of long established best practices to their extreme (this is where it gets its name,) [119]. It is well established that code reviews and inspections are more effective than testing, XP incorporates this into all code development by mandating Pair Programming [119]. McDowell et al. 75 describe pair programming as two programmers of varying experience sitting side by side at a single computer developing a program together. One programmer identified as the "driver" will control the input devices (mouse and keyboard) and actively creates the code, the "non-driver" (or "navigator" [14],) checks the code produced identifying misspellings, logical and syntactical errors and ensuring that the code is in keeping with the design [75]. After a designated period the programmers exchange roles, possibly moving to work with others within the programming team [75].

A full list of XP practices is shown in 3.5.

XP practice	Description
The Planning Game	The scope of the next release is quickly determined. When necessary, the plan is updated
Small releases	First realize a simple system, then release next versions in short cycles
Metaphor	Use of a simple metaphor for the whole system
Simple design	Make sure the design is as simple as possible at any point in time. Remove complexity as soon as possible
Testing	Programmers continuously write unit tests, customers write acceptance tests
Refactoring	Restructure the system without changing its behaviour, to improve quality
Pair programming	All code is written by two programmers at one machine
Collective ownership	Anyone can change any code, anywhere, at any time
Continuous integration	The system is integrated and built many times a day
40-hour week	As a rule, work 40 hours a week. Working overtime should be the exception
On-site customer	Have a real user on the team, full-time
Coding standards	Establish coding standards to ease communication

Figure 3.8 XP Practices

Figure 3.5: from Vilet [119]: List of XP Practices

XP teams are described [119] as being not to big and typically occupying one room. Meetings are short and involve the immediate delivery of functionality to the customer, who is involved in the planning in terms of setting priorities, determining dates etc. [119]. The desirable features are described in stories on index cards and a number of these will be realised per release [119]. Design is minimal as the future is presumed to be unclear and so is limited to the current version, as each increment is completed the code base is checked to see how it can be improved [119]. These improvements include removing duplicate code, simplification, increasing flexibility or robustness etc.,) and the process is called re-factoring often adhering so some pre-agreed standards [119].

Note: A certain member of staff delighted in catching students out in the final year vivas with this one. Essentially when a student claims they used XP as their methodology he'd ask who they pair programmed with, similar trick questions can be thrown with other methods such as SCRUM.

As each story is implemented tests are written to ensure that it works, these tests must run successfully before the code is integrated into the system [119]. Once integrated the tests for all stories are run and if there is any failure the new code is removed to fix it, in this way there is always a running system [119].

XP is based on five key principles [119]:

1. **Rapid Feedback** - Feedback is quickly obtained usually in minuets or hours, testing each increment allows developers to learn with each iteration. Customer sees running system at each increment, provides expectation management aids selection of future features.

2. **Assume Simplicity** - Develop only what is needed for the task at hand, don't build in extra functionality you may need (e.g. for flexibility or reuse) leave such concerns for future iterations and re-factoring.

3. **Incremental Change** - In XP, things change in small increments. This includes the plan, the design, the team etc. everything a little change at a time.

4. **Embracing Change** - This is achieved by not designing, coding, planning more than is needed right now. Tackle the most pressing problem today and leave the rest for tomorrow.

5. **Quality Work** - Quality is a must, everything produced must be production quality.

Advantages

See tutorial.

Disadvantages

See tutorial.

3.6 Additional Reading

These notes summarise a number of key texts on agile development and other concepts, there are some which are however seminal and you really need to read for yourselves, I've listed these below and I expect you to read them. Folwer, M. The new methodology [online]. (2005) [cited 2016.08.23]. Available from: `http://martinfowler.com/articles/newMethodology.html` Folwer, M. Design dead [online]. (2004) [cited 2016.08.23]. Available from: `http://www.martinfowler.com/articles/designDead.html` Folwer, M. Semantic diffusion [online]. (2006) [cited 2016.08.23]. Available from: `http://martinfowler.com/bliki/SemanticDiffusion.html`

3.7 Recap - Home Grounds

In our previous class we looked at the work of Boehm and Turner 15 who suggested that all development approaches (or software development methodologies,) had their own home ground (2.6.2) that is situations where they were best suited. They also suggested that methods existed on a continuum between two extremes with the most planned such as waterfall at one end and the most agile such as XP at the other. They go on to consider balanced approaches, which have elements of both (see next section,) but we will also considered methods which are flexible and can accommodate both.

3.8 Balanced Approaches

Van Vliet 119 suggests that while the 'pure' agile methods (ones which demonstrate all the characteristics listed above,) have their proponents. Others believe that a mixture of plan drive and agile methods can be as productive.

3.8.1 UP and Agile Modelling

The Unified Process (or UP) has emerged as a popular iterative software development process for building object orientated systems [49]. The Rational Unified Process is derived from work on the UML (Unified Modelling Language) and is a good example of a hybrid process model 2.5 in that it brings together elements from a number of process models [111]. UP is both flexible and open an encourages the use of practices from other iterative methods,

such as XP, Scrum etc. For example Test Driven Development (TDD) continuous integration (CI) and re-factoring can fit within a UP project [49].

Larman 49 explains that in UP almost everything in terms of artefacts (what is produced) and practices (what you do,) is optional, though some are invariant (e.g. iterative and risk driven development, continuous verification of quality). The selection of what which practices and artefacts are used is provided in something called the Development Case.

From what I've read it seems that UP and its refinement RUP were intended to provide a framework for companies/individuals to develop their own methods. We'll be following that set out by Larman - as I like his book - but the fundamentals are very similar to those used in RUP. We'll look at the most salient activities and creating the some key artefacts..

The UP is a phased model but unlike other modules where the phases have associated process activities [111]. The phases of UP are more closely equated to business and the development of the right product [111].

A Slightly Confusing Diagram

Sommerville 111 suggests that this kind of diagram (only for RUP (3.6)) is a little confusing, as it attempts to show different kinds of work going on at the same time over iterations of the project.

Critical UP Practices / Concepts

Phases in UP

Up has four phases, each containing one or more iterations [49]:

1. **Inception** - Produce an approximate vision (a kind of statement of the overall goal in creating the software). Develop the business case (do we want this, will it bring any value) and some vague estimates of cost/time-scale.

2. **Elaboration** - This refines the vision and over a number of iterations produces the core architecture. High risk elements are resolved, the majority of the requirements are identified (but not analysed in detail,) along with the scope of the project. More realistic estimates are produced.

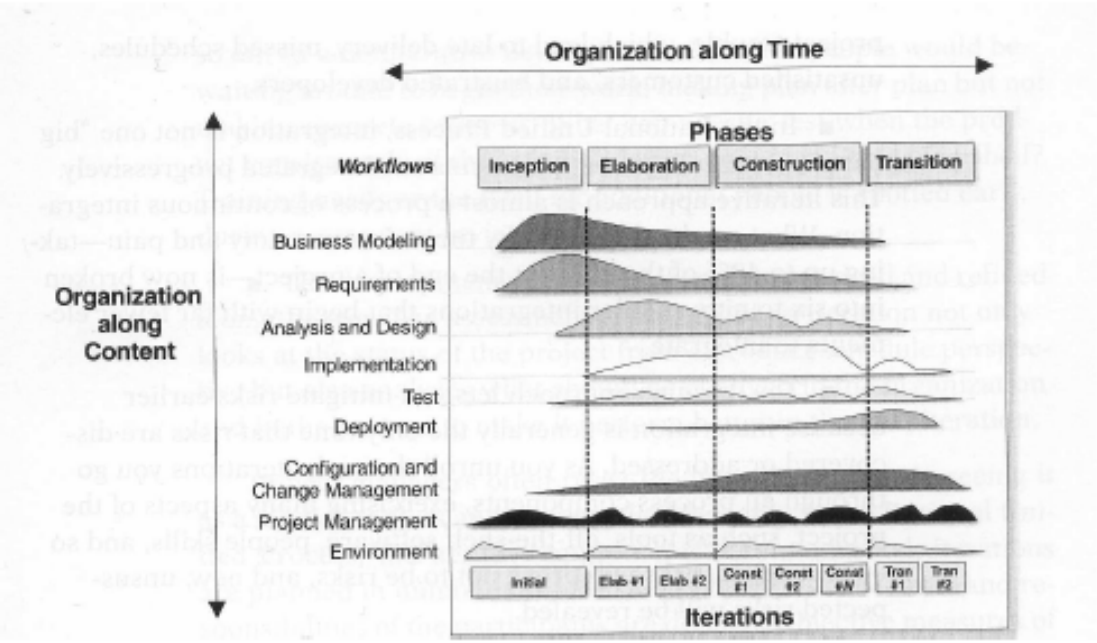


Figure 3.9 Process structure of RUP (*Source: P. Kruchten, The Rational Unified Process, An Introduction*, 2003, Addison-Wesley)

Figure 3.6: from Vilet [119]: RUP diagram

3. **Construction** - The remaining low risk/easier elements of the project are developed and prepared for deployment.

4. **Translation** - System goes into beta testing (we'll cover testing later,) and is deployed.

3.6 shows these going horizontally across over time, note at the bottom of the diagram the iterations, each phase may contain many iterations depending on the project size.

A key observation here is that neither inception or elaboration equate to the old requirements phase in the waterfall model [49]. Inception is a kind of feasibility phase where just enough is done to support the decision to continue or not and elaboration is the iterative implementation of the core and of risk (tackle the hard things) [49].

The UP Disciplines

The unified process defines work activities (such as drawing a particular diagram,) writing a some code within Disiplines (originally called Workflows) [49]. Workflows is actually a much better name (though it has a smell of serial about it). Essentially its a set of activities (some things you do,) which result in a number of artefacts (things e.g. code, text, database schema, models etc. etc.) [49].
This module (shamelessly,) follows Larman 49 and focuses on some of the processes and artefacts from the following:

- **Business Modelling** - Either just modelling the domain of a single application (in small systems,) or for larger systems modelling business processes across a much larger organisation.
- **Requirements** - Analysis of the requirements, writing use case documents and accompanying diagrams along with non-functional requirements.
- **Design** - All aspects of design, including architecture, object database etc.

Note: Design always seems to design of architecture (objects, functions/structures, modules in assembler etc.,) mechanistic design (how do the architectural bits do their job) also called interface design (as in how to bits talk) and finally detailed design (how to the complex little bits on the inside work).

Just to highlight some of the others Implementation means programming and building the system [49], I'm hoping to cover some of that. Environment is the setting up of the process (what to do etc.,) and the tool chain (compiler, linker, editor, version control, unit test suite etc.) [49] we'll do a tiny bit of that here and some in other modules. There are modules on both of these on the MComp., there just wasn't enough room in this module!

Sommerville 111 briefly describes all the workflows (see 3.7), EUP adds more. 3.6 shows these as workflows (their original name,) on the right hand side. It is sometimes the case these that are seen as distinct however this is not the case, for example in inception both business modelling and requirements go on together and project management is obviously continual.

Advantages

- Is an iterative and incremental approach.

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models, and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users, and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Figure 3.7: from Sommerville [111]: RUP workflows

- Can be customised to suit the project and project team.

Disadvantages

- Well documented, but a lot of documentation.

As suggested above there are a number of refinements of the original UP, RUP is heavily supported by IBM tools. Other methods include Scott Amblers Agile Unified Process and the Eclipse Foundations Open Unified Process. The official EUP website (Enterprise Unified Process,) provides the following timeline charting its evolution and that of some of the other refinements of UP [Ambler].

- **1988:** Objectory v1.0 is defined by Ivar Jacobson's Objectory AB company. The Objectory process defined the core process from which the RUP and later the EUP evolved.
- **1995:** Objectory v3.8 releases their first online version.

- **1995:** Rational Corporation purchases Objectory AB.
- **1996:** Rational Objectory Process (ROP) 4.0 is developed. Rational extends Objectory with internal Rational architecture concepts and introduces iterative concepts to it.
- **June 1998:** Rational Unified Process (RUP) 5.0 is released. This is the renamed ROP, extended with process material obtained from other tool companies purchased by Rational Corporation as well as material developed by the RUP group led by Phillipe Kruchten. This version describes how to apply the UML diagrams in the development of object-based systems.
- **February 1999:** The Unified Software Development Process is published. This book describes the Unified Process framework in detail.
- **June 1999:** RUP 5.5 is released. Major enhancements focus on real-time and web-based development.
- **October 1999:** An enhanced lifecycle for the RUP is first proposed by Scott Ambler, which extends the RUP with a production phase and a cross-system discipline called Infrastructure Management.
- **February 2000:** RUP 2000 is released. Major enhancements include the addition of business engineering techniques to the business modeling discipline and a more extensive approach to requirements.
- **2000-2002:** The Unified Process series of books, edited by Scott W. Ambler and Larry Constantine, is published. This series repositions the enhanced RUP lifecycle as the EUP. These books reprint a large collection of articles originally printed in Software Development that describe topics not yet covered by the RUP. Some topics, such as improved testing and agility, have since been added to the RUP.
- **March 2002:** Agile Modeling is published, showing how it is possible to take an agile approach to modeling on a RUP project.
- **August 2002:** IBM purchases PriceWaterhouseCooper (PWC), developers of the Summit methodology. Material from Summit is later adopted into the RUP.
- **December 2002:** IBM purchases Rational Corporation.

- **May 2003:** RUP 2003 is released. Some of the material from the Unified Process series, in particular an enhanced test discipline and some agile concepts, are now covered by the RUP.
- **Spring 2003.** The EUP site and the EUP mailing list go online.
- **January 2004:** EUP v2004 is released. Enhancements include the addition of the retirement phase and the expansion of the enterprise management discipline (formerly the infrastructure management discipline) into seven detailed disciplines.
- **February 2005:** Release of The Enterprise Unified Process: Extending the Rational Unified Process by Prentice Hall PTR.
- **September 2005:** The Agile Unified Process (AUP) product is announced at Software Development Best Practices in Boston.
- **October 2005:** The Eclipse Project Framework (EPF) and Basic Unified Process (BUP) announced by IBM.
- **November 2005:** The Rational Method Composer (RMC) is announced by IBM. For more information, read Peter Haumer's IBM Rational Method Composer: Key Concepts.
- **March 2006:** BUP is renamed OpenUP
- **September 2006:** EPF 1.0 and OpenUP 0.9 released
- **June 2012:** Disciplined Agile Delivery (DAD) book released

Agile Modelling

Ambler 7 proposed agile modelling, this is described by Larman 49 as using models primarily to understand rather than to document. The act of modelling should serve to explore the problem or solution, i.e. rather the a designer producing a large number of design diagrams which are handed over to the programmer (very waterfall,) diagrams are used to quickly explore the problem and solution [49].

Larman 49 summarises the principles and values of agile modelling as:

- Adopting agile does not mean no modelling - many agile methods have significant modelling sessions, even the XP founders endorsed agile modelling

- The purpose of the models is to support understanding and communication rather than for documentation
- Don't model all of the system, just the architecturally significant and technically complex parts
- Use the simplest tool possible, i.e. sketch on whiteboard or paper rather than use a UML tool. Though where you do need to produce diagrams for maintenance teams etc., formalising sketches is made easier by such tools. Scribbling use case documents on paper is fine in a meeting but you'll need to read them later so word process!
- Don't model alone, pairs or triads at a whiteboard is the way to go. Larman recommends taking turns much like pair programming.
- Create models in parallel - this will make more sense when we actually do it but its best to have the static diagram and dynamic diagram on the go a the same time.
- Use good enough notation when sketching, ideas are important not syntax. Obviously for the cleaned up diagrams a tool is much better than paint [1].
- Know that all models will be inaccurate, only the final tested code will show the true design. All prior diagrams are incomplete hints, best treated as throw away explorations [2].
- Developers themselves should do the modelling, for themselves not to create diagrams to give to other programmers (i.e. waterfall mentality).

[1] Anyone using some crude drawing package (or worse the drawing tools in a word processor) will have their marks for presentation seriously reduced.

[2] I want to see these for the course work and I'd like some to see some of the later ones or some based on the code to be drawn up using a tool with all the right syntax.

Agile UP

UP was not intended by its creators to be either heavy or un-agile, although if the many optional activities and artefacts are taken to be compulsory then it certainly would be [49]. Ambler 8 details how agile practices can work with

the UP in principle, while Larmans book works though it in a more practice centred fashion.
In essence this means customising UP, Larman 49 notes the following:

- Prefer a small set of UP activities, that is keep it simple, all up artefacts are optional don't create any unless they add value.
- Program early and document late.
- UP is iterative and evolutionary, complete requirements and designs will not be complete before implementation starts, they will emerge over many iterations based on feedback.
- There is a plan for the whole project - the Phase Plan, but this only states which features are scheduled for which phases and the overall project end date. The Phases themselves have are planned in iterations (with many iterations to a phase,) but detailed planning is only for the next iteration.

3.8.2 ROPES

The ROPES methods was one I discovered while writing a module called Real-time and Critical Systems Design (which has long since been retired,) while it has a lot in common with UP the author (Douglass,) was one of the first I found who explicitly showed how in an approach with a fair amount of up-front planning analysis models could be mapped to architectural and mechanistic design and detailed design filled in the blanks. I'll probably re-visit some of the key ideas later in the module.

3.8.3 Do we need to design?

There are certainly some who advocate no design [35], though most (including Folwer,) recognise design as an important tool in the the box for any software developer. They provide a means to sketch ideas in a way others can understand and provide documentation at a high level allowing others to evaluate the design. They also act as a language to describe design patterns, anti-patterns and algorithms.

Larman 49 suggests that rough diagrams should always be produced so that developers can quickly consider various solutions (whiteboard is faster than programming). He also that key portions of the design should be formally recorded, for future reference and to aid processes such as re-factoring and

maintenance.

I'm going to teach you design and the diagrams, who knows what you'll be using in your first job? What if you need to read up on a design pattern and the book uses UML or similar?

A while after I wrote this found a section in Scott Amblers book on Agile Modelling entitled "Overcoming Misconceptions that Surround Modelling" [9], I'd like you all to read these. He makes a number of very convincing arguments firstly knowing how to model and secondly developing models to aid design, I think my ideas for the model match well with this.

3.9 Plan for this year

In the past I decided the best way to order the material was linear, analysis techniques, design techniques, implementation techniques and testing techniques. However, timing wise this left very little time at the end for a coursework and required students to work on the project as they went along. While iteration was encouraged it was difficult for students to fit this in with the linear teaching.

This year my plan is to teach in an Iterative and Incremental style, we'll start by looking at the UP method and applying a subset of the techniques to a case study for each of the phases of the approach. We'll then iterate looking at more techniques applicable to each phase and applying them (possibly to different case studies if the initial one isn't complex enough).

3.10 Tutorial - Evaluating Development Models

1. In your groups identify at least 3 advantages and 3 disadvantages of XP, we'll then go round the round the groups. We'll discuss all the contributions and groups will be scored for new worthy contributions to the list. I'll add the results to the notes.

2. In your groups write a short section on Scrum including advantages and disadvantages.

3.11 Summary

In summary whatever approach we follow, we always figure out what to do, how to do it, do it and make sure its done. The stage of the linear models we started with last week are still there in some form, but in current approaches (i.e. agile,) we focus more on getting the design right, coded and tested than on producing beautifully drawn diagrams which may not work. While iterative incremental methods seem to have the edge in general and agile methods are currently in vogue, we note that there are still 'home grounds' where planned approaches are still required and approaches new methods are still being developed.

3.12 Preparation for Next Week

The chapter below has a good review of software development life-cycle models, read this:

Schach, S. R. (2007c). *Object Orientated Software Engineering*, chapter The Software Process, pages 68–100. McGraw-Hill, New York, USA

I'd also recommend reading this:

Sommerville, I. (2011c). *Software Engineering*, chapter Agile Software Development, pages 56–81. Addison Wesley, 9th edition

For next weeks class please read, Section 4.3 of next weeks notes, I'm hoping to avoid talking though this and focus on the application of these concepts in UP.
YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Inception (Part 1)

4.1 Introduction

The Unified Process (or UP) has emerged as a popular iterative software development process for building object orientated systems [49]. The Rational Unified Process is derived from work on the UML (Unified Modelling Language) and is a good example of a hybrid process model 2.5 in that it brings together elements from a number of process models [111].

To remind you here four phases of UP, each one may contain one or more iterations [49]:

1. **Inception** - Produce an approximate vision (a kind of statement of the overall goal in creating the software). Develop the business case (do we want this, will it bring any value) and some vague estimates of cost/time-scale.

2. **Elaboration** - This refines the vision and over a number of iterations produces the core architecture. High risk elements are resolved, the majority of the requirements are identified (but not analysed in detail,) along with the scope of the project. More realistic estimates are produced.

3. **Construction** - The remaining low risk/easier elements of the project are developed and prepared for deployment.

4. **Translation** - System goes into beta testing (we'll cover testing later,) and is deployed.

Agile modelling was proposed by Ambler 7 this is described by Larman 49 as using models primarily to understand rather than to document. Rather the a

designer producing a large number of design diagrams which are handed over to the programmer (very waterfall,) diagrams are used to quickly explore the problem and solution [49]. This is the approach we'll be taking though the module, however we'll also learn the correct syntax and draw up some correct diagrams (to show we can).

UP is not by design heavy or un-agile, although if the many optional activities and artefacts are taken to be compulsory then it certainly would be [49]. Ambler 8 details how agile practices can work with the UP in principle, while Larmans book works though it in a more practice centred fashion, we're going to follow Larmans approach and work though the phases

4.2 Inception

Inception is a short initial step which establishes a a common vision and scope for the project [50]. Both Business Modelling and Requirements identification will begin in the inception phase and we start to define a number of artefacts (see 4.1. Larman 50 suggest that around 10% of the use case (5.3.2) will be defined along with critical non-functional requirements (4.3). A business case is produced and the development environment is configured so that programming may begin (in the next phase).

Inception is intended as a very short step to answer key questions about the project (based on [50]):

- What is the vision (think executive summary) and business case for this project?
- Is the project feasible?
- Should we buy, build or both (COTS)?
- Vague cost/time predictions i.e. is it going to take months or years, £100K or £100 million.
- Should we continue?

A key point here which Larman 50 goes to great lengths to emphasis is that this should be a very short phase (a week or so,) which aims only to work out if the project is worthy of further investigation. It is not a requirements phase or a feasibility study, the requirements of the project will be (for the most part,) identified during elaboration, in parallel with the production of

early but release ready code and testing [50].

The documents and diagrams produced at this stage will be fairly light and incomplete, e.g. there may be a 'top-level' use case diagram, that is one that shows the system as a black box showing only its most important behaviours [50]. Textural descriptions of the use case which embody the key functions of the system, along with the supplementary specification (which capture constraints on the systems behaviour,) are drafted and are importantly 'work in progress' at this stage [50]. As is the vision which acts as an executive summary of the whole project [50].

Aside: This is a bit of a pig to teach as there are a lot of things which are started in this phase and continued in elaboration. I appreciate I'll be spending a bit of time introducing them now and this may in your minds associated them with the inception phase. Keep in mind that they are started now and completed in elaboration.

There may be some programming in this phase to explore "show stopper" technological issues i.e. what can be done with current technology [50]

4.2.1 The Development Case

Prior to starting to work though the UP method, we first need to identify the practices and artefacts of up we're going to use. An example of a development case from a more complicated project is given below (from Larman 49). Ours will at this stage will be smaller. Interestingly Larman 49 suggests that the selection of artefacts is system specific, for example control systems may need a number of high level state chart diagrams while e-commerce systems may require a large number of UI prototypes.

Sample Development Case An example of a development case is shown in 4.1 (from Larman 49).

Development Case for the Module Need to make a table here :-D
The keen eyed of you will notice that the Development case is actually one of the artefacts which may be started in the inception phase of a UP project.

Discipline	Artifact Iteration-*	Incep. I1	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 2.1 Sample Development Case of UP artifacts, s - start; r - refine

Figure 4.1: from Larman [49]: A Sample Development Case

4.3 Requirements

This is a pretty major aside, before we can continue looking at UP and its practices we need to first cover some basic theory on Requirements and requirements engineering, we'll then return to looking at Evolutionary Requirements as recommended by Larman.

"The requirements of a system are the descriptions of what the system should do, the services that it provides and the constraints on its operation" [113]. Requirements in the UP are organised into a number of key artefacts (almost all of which are started in this phase). In order to balance the next few classes we'll spend some time looking at each, however please keep in mind that these are developed over many iterations spanning both the inception and elaboration phases.

4.3.1 Requirements Engineering

The term requirements is not used consistently in software engineering being used to describe both high level abstract needs of a customer and at a much lower level the behaviour of a proposed system [113]. Sommerville 113 suggests that is due in part to contractual considerations, a company will invite bids on a set of abstract requirements. The winning company would provide a detailed description of what the proposed system would do to be agreed by both parties, both descriptions would be referred to as requirements.

Sommerville 113 goes on to distinguish these as:

1. **User Requirements**, statements in natural language with diagrams of what services the system is expected to provide to users and the constraints under which it must operate.

2. **System Requirements** more detailed descriptions of the systems functions, services and operational constraints, sometimes called the functional specification. This defines exactly what must be implemented and often forms part of the contract between the system buyer and the developer.

4.2 shows a simple example, note that these are intended for different readers. The user requirements are for those not concerned with how the system is implemented but that is does. Those reading the system requirements need to know exactly what the system should do so it can be implemented, or to ensure it fulfils the business process.

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Figure 4.2: from Sommerville [113]: User and System Requirements

4.3.2 Functional and Non-Functional

Software requirements are often classified as either functional or non-functional and the vast majority of texts which discuss software development still use these terms. According to Sommerville 113:

1. *Functional Requirements* Statements of services the system should provide. How it should react to particular inputs or behaviour in a given situation, also possible what it should not do.

2. *Non-functional Requirements* Constraints on the services or functions offered, e.g. timing constraints, legal constraints, development process constraints etc. These are often imposed on the whole system rather than individual features or services.

This is not as clear-cut as we might imagine for example "Access to the system must be limited to authorized users" may be a suitable non-functional requirement but when developed in more detail there are clearly functional requirements for user authentication and possibly data encryption. We could also come across "Must comply with the data protection act" which imposes a number of requirements on all data of a particular kind (personal) within our system! In this way requirements are not independent and often on has the acts as a constraint on or generates others [113]

Functional Requirements

Functional requirements describe what the system should (and where required, should not), they will be entirely dependent on the kind of software being developed [113]. These may be expressed at a high level (User Requirements) describing what the system should do in an abstract way or as System Requirements in terms of the system functions, inputs, outputs exceptions etc. [113].

In principle functional requirements should be both [113]:

- **Complete** - all services/behaviour defined by required by the user should be defined.

- **Consistent** - the requirements should not contradict.

This is probably impossible for all but the most mundane systems, it is easy to make mistakes, the requirements are, even when very carefully documented prone to misinterpretation [113]. There are several reasons for this a key one is that stakeholders (i.e. someone who has a stake - vested interest,) in

the system often have inconsistent needs which are not always apparent [113].

Non-Functional Requirements

Non-Functional requirements are those which are not concerned with the specific services provided by the system [113]. They relate to what are known as the emergent properties of the system [113], that is things which it is, e.g. reliability, response times, usability etc. They may also describe constraints on the system imposed by hardware, interfaces with other systems or the domain of the system [113]. For example, must work on the hardware of a Boeing 747 and conform to DO-178B/ED-12B. Such requirements often cover the system as a whole and the failure to meet such requirements may render the system useless [113]. In the previous example, no matter how perfect the software if it does not conform to the relevant US/EU regulations for software safety it will not be certified to fly in those regions.

Non-functional requirements can be broadly classified as Product, Organisational and External [113]:

- *Product Requirements* Constraints on the behaviour, e.g. performance (frame rates for games, response times for database systems), memory requirements, failure rates, security, usability etc.

- *Organisational Requirements* Based on the policies and procedures of the companies involved (i.e. both producer and consumer,) these may include process requirements e.g. managers may have access to systems they will not normally use in case they are called upon to cover. On the developer side programming languages, development environments, OS, coding standards etc.

- *External Requirements* Factors external to the system and development process, e.g. regulatory and legal requirements such as the data protection act and ethical considerations and codes of conduct. We'll revisit these later in the year once we've covered the critical parts of the development process.

Further classification is possible (see Sommerville 113[p.88]).

A warning echoed by [113] is the difficulty in quantifying non-functional requirements, especially where requirements have to be met for the system to be accepted. As a crude example "Response to queries must be fast" fails

to give any quantification of what is an acceptable response time. A client could potentially keep arguing for improvements (and delay paying,) until their personal onion of 'fast' is satisfied. The flip side of this is that a lazy developer can argue that they've met the requirement as their solution is faster than reading through a printed copy of the database.

4.3.3 Requirements Engineering

Requirements engineering is the name given to the process of identifying the requirements of the system [113]. This will involve interaction with stakeholders i.e. anyone who will be involved with the system in some way, though the exact process depends on the organisation, developers and the methods selected [113]. The general process model is given by [113], see 4.3:

1. *Requirements Discovery* The process of interacting with stakeholders to discover their requirements - we'll consider this later.

2. *Requirements Classification and Organisation* Taking the unstructured collection of requirements and grouping them, this may be along the lines of the sub-systems and therefore has a natural cross over with architectural design.

3. *Requirements Prioritization and Negotiation* Working out which requirements are most important to the organisation and where there are conflicting requirements agreeing on compromises.

4. *Requirements Specification* The requirements are documented, in some form. This will largely depend on the methodology/methods being used. These will eventually form the basis of future phases (e.g. leading in to the design in waterfall,) or may be part of an ongoing process of requirements analysis feeding back into requirements gathering though a process of discussion.

In planned methods, the requirements phase comes to an end at some point and the document is complete [113]. We'll look at the Agile UP process and Evolutionary Requirements (4.4 in a later section, but even in UP there comes a point in the Elaboration phase where the requirements are stable [49].

Elicitation Techniques

There are a number of long established elicitation techniques which we'll consider. Agile methods employ some of the same techniques however as many

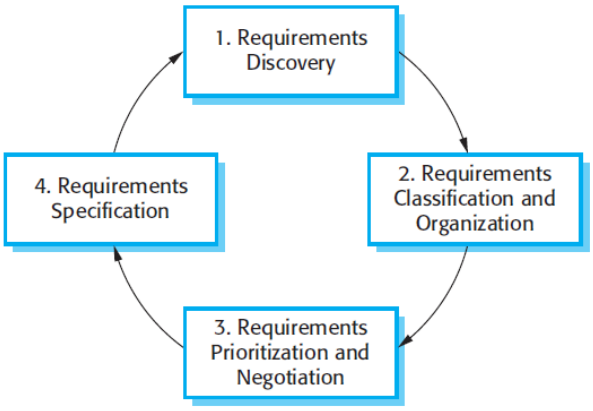


Figure 4.13 The requirements elicitation and analysis process

Figure 4.3: from Sommerville 113: The requirements elicitation and analysis process

mandate a client on site (i.e. as part of the development team,) they allow for additional informal methods. As this is prime exam question territory I'm going to get you to read into some of this as part of the tutorial.

NOTE: This is another of those difficulties in teaching Software Engineering, providing a 'realistic' experience of this part of the process difficult if not impossible.

Stakeholders Sources of information at this stage may involve documents, system stakeholders and specifications (or feature lists,) of similar systems [113]. A stakeholder is anyone who is in some way involved in the system including the person paying (often the client,) the users, their superiors/-guardians, any authority over the domain in which the software operates [113]. In addition systems which interact with ours may also make requirements of our system, for example that we can provide information in format x in response to queries in the format y, or that we can render a model in format z. These can all be represented as separate views (or view points,) which can be used to organise the requirements, however none provides a complete view of the system [113].

Traditional Techniques I'm going to briefly introduce the techniques here, for a more detailed description see Sommerville 113.

- *Interviews* Formal or informal interviews with the potential stakeholders are part of most requirements engineering processes, the team ask

questions and derive requirements from the answers [113]. There are two distinct types [113]:

1. Closed interviews - a set of pre-defined questions is asked. This is essentially a questionnaire and probably the least useful.
2. Open interviews - no pre-defined agenda, the requirements team explores a number of issues over the course of the interview.

In practice most end up being a mix of these, pre-defined questions are used to initiate discussions about key areas of the system, keep the interview focused and ensure that all participants are asked similar questions [113]. Interviews are usually a good source of information regarding roles and supplement documentation on business processes (how it is done/vs how it should be done). They are less useful for domain requirements, the stakeholder who is immersed in the domain will use terminology which may be misinterpreted or assume knowledge which is obvious to anyone in their domain [113].

- *Scenarios* - People find working though real-life situations much easier to relate to than abstract descriptions of processes [113]. A description of how they would interact with the system is much easier to follow (and correct,) and stakeholders can often think of a large number of exceptions to the norm, from this we gather requirements [113]. Typically scenarios involve some set context (assumptions about the state of the system and environment,) along with a description of the normal flow of events, exceptions and how they are handled, simultaneous events and the state of the system and environment when the scenario finishes [113].

- *Use Case* - Use Case are a combination of document and diagram we can use to represent parts of the system. They were introduced as part of the Objectory method (which eventually became UP,) and are now a part of the Unified Modelling Language (UML). They are a tool for discovering functional requirements from those who interact with the system and describe one (or a group of related) scenarios [113]. We'll be discussing them at length next week see 5.3.3.

- *Ethnography* - Software systems are part of a social and organisational context which must be accounted for, ethanography is a observational technique which attempts to understand the way people actually work rather than what the formal process defined by the organisation [113].

Agile Techniques See tutorial question!

Requirements Specification

The documentation of the requirements is one of the more difficult parts of the process, ideally the requirements should be unambiguous, easy to understand, complete and consistent [113]. In practice this is very difficult to achieve both the stakeholders and developers interpret the requirements differently and there are often inherent conflicts and inconsistencies in the requirements gathered [113].

Sommerville 113 suggests that user requirements should describe the functional and non-functional requirements in a way that can be understood by users who don't have detailed technical knowledge and should only specify the external behavour of the system. System requirements are an expanded version of these used as the basis for the design process, they should avoid where possible specifying how the system is developed. Sommerville 113 stresses that this is often impossible as the system requirements end up being organised around the architectural components of the system (e.g. a group of requirements related to the finance sub-system, or the stock-subsystem). Also where existing systems are being connected or components/frameworks being used (e.g. Swing, .Net, OpenGL,) these may constrain the design. Finally specific architectures may be mandated to achieve certification (e.g. N-Version programming for critical systems,) client-server where low powered clients are to be deployed etc.

The format of the requirements depends on the methodology being deployed, most use a combination of text and diagrams [113]. The table shown in 4.4 provides an overview of these Software Engineers will be exposed to Mathematical Specifications in the Formal Methods module. Sommerville 113 provides some good examples of the different approaches taken and considers the pros and cons of each.

In this module we'll focus on written text and UML models, however there are many more and most of you will have seen DFDs and ER diagrams from the SSADM methodology last year.

Validation

Getting the requirements right is key to any software development project and as a result the process of checking the requirements define the system

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract.

Figure 4.4: from Sommerville 113: Ways in which requirements may be documented

the customer really wants (or validation) is an essential one [113]. Several different kinds of check can be undertaken on the requirements document [113]:

1. *Validity Checks* A single stakeholder may think that a function is required, however analysis may show that additional or alternative functions are required. Due to the diverse nature of system stakeholders requirements are often a compromise across the community.

2. *Consistency Checks* Requirements in the document should not conflict.

3. *Completeness Checks* The document should be complete and describe all functions and constraints intended by the user, omissions are often found here when the systems responses to unusual conditions have not been anticipated e.g. incomplete/incorrect data from another system or the user.

4. *Realism Checks* Using knowledge of the existing technology the requirements must be checked to ensure the system can be developed, and completed on time and within budget.

5. *Verifiablity* To reduce the potential for dispute between customer and contractor requirements should be written so they can be verified, i.e.

we can show we have achieved them by means of some test.

There are a few techniques which can be employed to validate requirements [113]:

1. *Requirements Reviews* Systematic checking of requirements by a team of reviewers.

2. *Prototyping* An executable model of the system is shown to the users/customers who experiment with the model to ascertain suitability.

3. *Test-case Generation* Requirements must be demonstrated and so tests can be devised as part of the requirements generation process. If a test cannot be written, how would we show we'd completed the requirement. Developing such tests and using these to drive development is part of XP, and in test drive development the tests are a part of the system specification.

[113] acknowledges the difficulties in getting requirements right up front, its hard for developers to imagine the system in operation and more difficult for users. It is inevitable therefore that requirements will change.

Management

Requirements for large software systems are always changing, part of the reason is that they are intended to address something new e.g. a new problem, a new demand or gap in the market etc., with a result the problem they solve is not completely defined [113]. During the software process the stakeholders understanding of the problem changes and requirements must evolve to reflect this, the reasons are varied but the result is that we require some process of managing the requirements as they change.

Sommerville 113 suggests requirements management, this is a process where by the requirements are tracked once documented though various revisions and changes. This usually requires that they can be uniquely identified, that a change management process (for changes after the requirements are approved), policies to maintain links between user and system requirements and how these map to the system design. This is a very complex process due to the number of requirements, however there are tools available to support this [113]. Where the requirements document forms the basis of a formal contract Sommerville 113 recommends a formal process for change with three

principles stages: Analysis and specification of the change, analysing the effect and cost of the change and finally implementation. Urgent changes will always temp developers to implement first and update the documents later, this can cause issues if the requirements and product get out of step [113].

The issues with managing change requirements are a key driver for the adoption of more Agile development methods, XP for example uses a prioritised feature list [113]. Here an urgent change could be added to the feature list and prioritised for the next build if required.

4.4 Evolutionary Requirements

One of the challenges of traditional requirements engineering is the management of changing requirements, Larman 51 uses the term evolutionary requirements to describe the requirements process in evolutionary methods (Scrum, XP, FDD etc.,). Here production quality programming and testing begins long before the requirements have been analysed or specified, perhaps when only 10-20% of the most architectural significant, risky and high-business value requirements have been specified. He highlights studies which show that 25% of requirements change on software products and that 45% of features were never used. At the same time Larman cautions against build and fix suggesting a middle way, iterative and evolutionary requirements analysis, combined with time-boxed iterative development and frequent stakeholder participation, evaluation and feedback on partial system.

4.4.1 Skilful Means to Find Requirements

UP encourages skilful elicitation via techniques such as writing use case with custeroms, requirements workshops (with developers and customers,) focus groups with proxy customers and a demo of the results of each iteration to the customers to solicit feedback [51]. UP welcomes any requirements elicitation method which can add value and that increases user participation, even the simple XP "story card" approach [51]. Larman 51 notes that this is often difficult to use as it requires a customer-expert (a customer who knows the whole system,) in the project room which is often difficult to achieve. High levels of user interaction (as recommended by many agile approaches) are not always practical in a commercial setting (people have jobs to do,) the workshop approach recommend by Larman seems a good compromise.

4.4.2 FURPS+

In the UP requirements are categorised according to FURPS+ [51]:

- **Functional Requirements**, features, capability and security.
- **Usability**, human factors, help and documentation.
- **Reliability**, frequency of failure, recoverability and predictability.
- **Performance**, response times, throughput, accuracy, availability and resource usage.
- **Supportability**, adaptability, maintainability, internationalisation and configurablity.

The "+" in FURPS+ indicates ancillary and sub-factors such as [51]:

- **Implementation**, resource limitations, languages and tools, hardware etc.
- **Interface**, constraints imposed by interfacing with external systems.
- **Operations**, system management in its operational setting.
- **Packaging**, a physical box, packaging for application stores etc.
- **Legal**, licensing and conformance.

4.5 Where are the Requirements in Agile UP

The UP offers several requirements artefacts (though as with all UP artefacts they are optional), we'll limit ourselves to the following (as described by [51]):

- **Use-Case Model** – A set of scenarios of using a system, primarily for recording **functional** (behavioural) requirements. **Supplementary Specification** – A record of all non-functional requirements (e.g. performance, licensing etc.) also the place to record any functional diagrams which cannot easily be expressed by use cases (e.g. report generation).
- **Glossary** – Defines noteworthy terms and encompasses the data dictionary. This defines all recorded requirements related to data, such as validation rules, acceptable values etc.

- **Vision** – Summarises high-level requirements that are elaborated in the Use Case Model and Supplementary Specification along with the business case for the document. Essentially provides a short executive overview to orientated those new the the project.
- **Business Rules** – Business (or Domain) Rules describe requirements or policies which transcend individual software projects. For example the data protection act be documented as a set of domain rules. Domain rules may be recorded in the supplementary specification but as these will apply to all projects it makes more sense for them to be central and shared.

Larman 51 suggests that these could be best documented using web pages or a Wiki but also considers wall posters or any other suitable media. The RUP documentation provides templates for all artefacts (though these are optional,) we'll look at using these as a starting point. I've found some here.

4.6 Tutorial

1. In pairs search 'agile' requirements elicitation techniques (a few are mentioned in the above). Write a paragraph on each of these along with a minimum of two slides. Be prepared to talk about these.
2. Read the case study provided (4.6.1), produce a list of requirements, these should be classified using both the classifications suggested by Sommerville and that used in UP. I suggest you format them as follows:

Requirement & brief description	User, System or Domain	Functional or Non-Functional	FURPS+
Developed on Windows	Domain	Non-Functional	+(Implementation)

3. Read the case study provided (CaseStudy1-L4), based on the examples provided begin work on a glossary and vision document use the templates from here.

Questions to think about (or what kind of exam question could I get on this).

- What are the difficulties associated with requirements specification in linear methods such as the waterfall and how does the use of an incremental and iterative approach attempt to alleviate these issues?

4.6.1 Case Study

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.
The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

4.7 Summary

In summary, requirements engineering in the UP (and particularly Agile UP,) is less document heavy than in staged methods such as the waterfall. However we still need to document our requirements to ensure everyone involved in the development (both customer and developer,) know what is being created. The shift in methods is as Gordon suggests a move from a detached (and possibly aloof) attitude of 'tell us the problem and we'll build a solution' to 'getting into the shoes of the customer' and working with them towards a solution.

Remember that UP is an iterative process, there will be a few iterations in the Inception phase, perhaps working with different groups of stakeholders or considering distinct parts of the system. This iterative component is a powerful tool in itself in terms of analysis, it gives our stakeholders practice of telling what they want and us of interpreting their requests.

In terms of this module we're going to get lots of experience of doing the key practices/actives of software development.

4.8 Preparation for Next Week

4.8.1 Additional Reading

Covering current topic

The section of Sommervilles book below covers requirements analysis with a view to being broad enough for both traditional planned and agile approaches. Its far more detailed than my notes/slides and worth reading for the extra detail!

Sommerville, I. (2011c). *Software Engineering*, chapter Agile Software Development, pages 56–81. Addison Wesley, 9th edition.

You may note that Sommerville does not believe that approaches such as that taken by Larman are universal. I agree with this, but I think its a more tangible place to start than a lengthy requirements analysis as part of a more planned approach. I suggest reading these on the basis I might have missed something, they're very short chapters.

Larman (2014b). *Applying UML and Patterns*, chapter Inception is Not the Requirements Phase, pages 47–53. Prentice Hall PTR.

Larman (2014c). *Applying UML and Patterns*, chapter Evolutionary Requirements, pages 47–61. Prentice Hall PTR.

I'd also recommend reading this its a much longer chapter and may require more explanation:

Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. (2002). Agile software development methods: Review and analysis. Available from: `http://www.pss-europe.com/P478.pdf`.

4.8.2 Essential Reading

For Next Week

Please read 5.3 from next weeks notes (incomplete at the time of writing). This is based on a section of Larmans book which is avaible from the web. He actually leaves some of what I'm going to cover to a later chapter (omits some diagram syntax to focus on how we make them).

4.8.3 Background Reading

If you can find a copy ...

This optional is in our online library, can you see if you can get to it? Its not really what we want as it focuses on the digram link its in our online library.

Covers only the UML diagram syntax.

I'm ordering this: Fowler, M. (2003a). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

For the library/digital library, I'm horrified we don't have an up to date copy.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP:
Inception (Part 2)

5.1 Introduction

The aim of Inception is to produce an approximate vision (a kind of statement of the overall goal in creating the software). Develop the business case (do we want this, will it bring any value) and produce some vague estimates of cost/time-scale.

5.2 Inception - Recap

Inception is a short initial step which establishes a a common vision and scope for the project [50]. Both Business Modelling and Requirements identification will begin in the inception phase and we start to define a number of artefacts (see 4.1. Larman 50 suggest that around 10% of the use case (5.3.2) will be defined along with critical non-functional requirements (4.3). A business case is produced and the development environment is configured so that programming may begin (in the next phase).

The documents and diagrams produced at this stage will be fairly light and incomplete, e.g. there may be a 'top-level' use case diagram, that is one that shows the system as a black box showing only its most important behaviours [50]. Textural descriptions of the use case which embody the key functions of the system, along with the supplementary specification (which capture constraints on the systems behaviour,) are drafted and are importantly 'work in progress' at this stage [50]. As is the vision which acts as an executive summary of the whole project [50].

Sample Development Case In the development case is shown in 5.1 (from Larman 49) we note that the requirements part maps to four artefacts, we'll look at the use case model first.

Discipline	Artifact Iteration-*	Incep. I1	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 2.1 Sample Development Case of UP artifacts, s - start; r - refine

Figure 5.1: from Larman [49]: A Sample Development Case

Last week we looked at inception in general and how the requirements of the system were mapped to different artefacts in UP. The use case model is the primary artefact for functional requirements, these are usually the easiest to spot and so we'll look at use case models first. We'll then look at how the other requirements found in the Supplementary Specification (essentially the FURPS+ headings) and the high level vision document and glossary.
We are going to use Agile modelling (proposed by Ambler 7) and described by Larman 49 as using models primarily to understand rather than to document. Rather the a designer producing a large number of design diagrams which are handed over to the programmer (very waterfall,) diagrams are used to quickly explore the problem and solution [49]. Clearly we need to do some work on use case before we apply them as part of UP.

5.3 Use Case

Use cases are primarily text based stories used to discover requirements, these stories form the basis of the analysis/design effort and will feed into many of the subsequent artefacts [52]. Larman 52 suggests that while learning the format of the documents/diagrams is trivial the many guidelines to identify

and write good use case takes much longer. The following diagram from Larmans book, shows use case at the center of the process and its influence on other artefacts and processes 5.2.
In previous years there was a focus in the lectures on the diagrams and documentation and getting the formatting right. I'm hoping to move away from this and get you to read though this prior to the class so we can spend the time looking at and applying the techniques to obtain the use case.

We won't be talking to much abut UP and inception in the following notes on Use Case, we'll use this to build a simple high level use case for our vision diagram later. In the elaboration phase (over multiple iterations,) we'll start to use use case with more vigour.

Larman 52 is very clear on the importance of the textural use case components over the diagrams, which he sees as secondary. The text base use case he presents are very similar to other scenario based techniques whose aim is the discovery and documentation of functional requirements. In essence we capture from the users point of view the fulfilment of user goals by writing the system i.e. its cases of use [52]. The original translation from Swedish is cases of use, poor translation appears to be common in UML [52].

5.3.1 Key Concepts

There are a number of key concepts, these are introduced by Larman 52 in much more detail, the book chapter is available online via his website. The following summarises the information presented for revision purposes and to support the notes, if you're reading though this to catch up (i.e. for the first time,) it would be better to start with Larmans book chapter.

Actor

An actor is something with behaviour which is external to the system, such as a person (with identified role,) a computer system or organisation [52]. This can also include the system under discussion (SuD) itself when it calls upon the services of other systems or subsystems [52]. There are three kinds of external actors [52]:

Type	Description	Why identify?
Primary Actor	The entity has user goals fulfilled though using services of the SuD.	To find user goals which drive the use case.
Supporting Actor	Provides a service (e.g. data) to the SuD. Could be another computer system, an organisation or a person depending on the level of the use case.	Clarify external interfaces (what the system connects to,) and protocols (how the information is exchanged, and the process).
Off-stage Actor	Has an interested in the behaviour of the use case but is not primary or support.	Required to ensure all actors have their interests considered. This can be easy to miss if their are not named, e.g. in the student register system the HMRC are an interested external entity (in terms of international student attendance,) but are not involved with the system.

Scenario

Scenarios are a specific sequence of actions and interactions between actors and the system, also known as a use case instance [52]. Informally the use case is a collection of related success and failure scenarios describing an actor (with a goal) interacting with the system, note that from the above that the actor may be a part of the SuD [52].

5.3.2 Formal Definition of Use Cases

The above provide the necessary background to understand the formal definition of use case provided by the RUP documentation [88]:
"When a Use-Case applies to a subject, it specifies a set of behaviours performed by that subject, which yields an observable result that is of value for Actors or other stakeholders of the subject."

The term subject here, essentially refers to the part of the system considered by the use case (at some level of detail,) this sometimes referred to as the 'System Boundary' [88]. This seems to be an old term which has been re-

placed to cover the use of use case to represent the system at various levels of abstraction, e.g. business use case vs. sub-system use case.

5.3.3 The Use-Case Model

UP defines the Use-Case model as part of the Requirements discipline, this is essentially the set of all written use case [52]. This can be quite a difficult concept to grasp, perhaps an analogy would help. When you see a road map of the UK in a book (an old concept now perhaps,) to fit the whole of the UK on one page you have to zoom right out. This is a bit like a top level use case (e.g. one given as part of the vision,) we can go to the page that shows Swansea (and the surrounding area,) in detail and mark a route between two locations (a normal use-case). Conceptually a combination of all the journeys covering all the pages of the map gives us a model of the UK road network, but this can't be shown on a single page.

In addition to the text based use case we can show the names of the use case on a diagram and their relations, while some authors promote this as an excellent way to spot shared functionality Larman 52 suggests that the diagrams are often a distraction and should be used to summarise and provide a contextual overview only. Interestingly he mentions context diagrams here, these were common in old specification documents to show the position of the system in relation to other systems and its inputs and outputs.

Use-case then provide us with the majority of our functional requirements [52].

The use-case may be used at different levels, (note the use of Level: in 5.6, Larman uses busniess, user goal and sub-system [52]. Fowler 39 suggests that the most common classification is **System Use-Case** (interactions with the software) and **Business Use-Case** which show the whole business (its response to customers, suppliers or an event). He goes on to discuss Cockburn's approach which suggests three levels, the core of use case are at **sea-level** (aka **user goal**, Elementary Business Process (EBP) or System) and represent a discrete interaction between the primary actor and the system. When we decompose a use case at this level into interacting or related parts (which we may later map to object interactions,) we move to **fish level** (aka **sub-function level**). In this classification system **kite-level** is used to describe **Business** level use case.

5.3.4 Rational for Using Use-Case

The story like nature of the use case text and their consideration of the system from the users perspective make them an accessible tool for goal/behaviour elicitation [52]. Another key strength is the ability to scale up in terms of formality and sophistication [52], see 5.3.5.

5.3.5 Use-Case Notation

Larman 52 recommends three formats for the use case, the first is a brief summary of success, the second a more detailed but still informal group of paragraphs and the third a highly structured detailed account. The following table provides descrptions of these along with recommendations for their use, based on the descriptions given in Larman 52:

Type	Description	When used?
Brief	Terse one-paragraph summary, usually of the main success scenario (also known as the "happy" or "sunny day" scenario (though such terms are **never** used in formal documents.	Used during early requirements analysis to establish subject and scope. Should take little time to create (see 5.4).
Casual	Informal paragraph format, with more than one paragraph covering different scenarios.	Again used during early requirements analysis, when exploration of exceptions, failures or alternatives is required (see 5.5).
Fully Dressed	Much more detailed with all steps considered, supporting sections documenting other aspects.	Used when many use cases have been written up in brief. During the early stages around 10% of the architecturally significant use cases will be written up in full (see 5.6).

The fully dressed version is part of the elicitation and analysis process, filling in the sections of the template forces us to ask questions about the group of scenarios, the actors involved etc. In evolutionary UP around 10% of the critical requirements would be documented in this way during elaboration [52]. Larman 52 stresses that these should not be considered final or complete they are almost certain to be incorrect or incomplete, they serve only

as guidance to the design. It may be that we return to these documents to ensure they are complete and up to date for future related iterations.

Larman 52 considers the stakeholders and their interests to be of primary importance, the behaviours capture by the use-case must satisfy their needs. He suggests that ensuring we consider the view point of every stakeholder is a thorough and methodical procedure for discovering and recording all the required behaviours. He also notes that the special requirements recording (i.e. non-functional requirements,) should be noted here for consideration **and** included in the Supplementary Specification.

There are a number of common templates, I'm using the one from Larman which is based (as most are,) on that of Cockburn available from his web page (this seems to have been corrupted by a move to a new layout manager). The template below (5.3) has some limited guidance, Larman 52 provides more detail.

Examples

Unlike Larman 52 who elected to spread his examples though the section I'm presenting all of them here (they all come from Larman at this stage, I hope in the future to add my own).

- Brief Use Case - Example 1 (5.4).
- Casual Use Case - Example 1 (5.5).
- Formal Use Case - Example 1 (5.6).

There should be a PDF on Moodle with some examples form a course run in the states, if they're not there prompt me so I upload them.

5.4 Use-Case Diagrams

There is a lot written in the text books about Use Case diagrams and I don't want to go too much into them. I'm trying to follow Larmans approach and focus on the textural elements which capture the detail rather than the diagrams which provide more of overview.

I'm using examples and syntax from both Larmans book and the OMG (2.5) specification document which is more up to date and benefits from some recent changes, key amongst these is the renaming of the System Boundary (the box around the use case,) with Subject. According to [88] a use case may apply to more than one subject which yields an observable result for actors or other stake holders. It also states that the subject an be a system (which will almost certainly be the case in inception,) a subsystem, component or class (which is more likely in elaboration and after).

N.B Remember use case can be written at different level: Organisational, SuD and Sub-system.

5.4.1 Basic Use-Case Syntax

In his book Larman distinguishes between high level use case (with a limited syntax) used in inception at high levels of abstraction to show use-case in the system context with other actors.
In the example (5.7) the whole system is shown was the Subject (square large square box). Each use case is an oval within this and the actors are shown either as stick men or boxes with the appropriate stereotype i.e. «actor »(this dual syntax allows for tools which do not fully support the notation). The primary actor/actors are normally shown on the left with supporting actors on the right [52] off stage actors are not shown, hence the importance of the use case document.

5.4.2 Intermediate Use-Case Syntax

While Larman introduces the other syntax of use cases at a later stages (Chapter 30, to be exact [53] I'm going to get it all out of the way up front. Essentially it only amounts to a limited set of relationships between use case (there are two) and between actors (one).

Generalisation

Between Actors - Larman never mentions this, I can see perhaps why. He demonstrates a concern throughout that sometimes we do things in models because we can when it brings nothing to our system. However in some systems where data is stored about related types of actors this may be useful.

Between Use Case - Larman 53 specifically says this makes things over complicated. However other sources do provide examples 5.9.

Include

The most common and important relationship is the include relationship, this is used to indicate that some partial behaviour is common across several use case [53]. We can refer to these other use case in our use case documents by using the Include keyword and identifying the other use case by name [53]. Italics or underlined text are used in the books examples, in many of the use case tools they are implemented as navigable hyper-links, interestingly this varies between edition two and three of Larman's book. In the latter he uses the include keyword, underlined and in italics.

I'm introducing these now as they can reduce the amount of text we (**actually you**) have to write, if we spot early on that there will be commonality between use case we can factor this out. For example, looking at an ATM (which is a common example in my notes,) we note that both Withdraw Cash and Check Balance require the user to Authenticate to the account (by inserting a card then entering their pin).

The OMG documentation on UML suggests that this relation should be used for both reuse and hierarchcal composition (i.e. making complex behaviours out of simple ones) [88]. This splitting of large use case is also recommended by Larman 53 and in more planned approaches used to decompose system behaviour so this can be mapped to individual sub-systems [26]

In diagrams this is shown as an open headed arrow (crows foot) pointing towards the included use case, with «includes» above it and a dashed line. Larman 52 stresses the need to avoid focusing on the use of the correct arrows, my feeling are mixed. There are only a few arrows to get right, I don't understand why its so hard to remember!

Extend

Extensions are defined in textural use cases within the *Extensions* section and updating this to include other potential extensions is the preferred way of updating these [53]. The rational for the extends relation is that under a given condition at some point an additional behaviour *may* be included [53]. Larman 53 suggests that this is used when a use case has been base-lined (implemented,) and we wish to extend its behaviour rather than go back and

change it. I've always used these to indicate where a behaviour might be included in contrast to the include where it will be included.

In diagrams this is shown as an open headed arrow pointing towards the extended use case, with «extends» above it and dashed line. The drawing of this is quite complex, we need to include the extension points and the condition (with a [guards], or as a note). A guard (decision point,) on any kind of diagram is shown in square brackets.

5.5 Guidelines for Creating Use Case

Larman 52 draws together practical advice on writing use case from a large number of authors in addition to his own personal experience.

5.5.1 Guideline: Essential Style

When writing use case it is common to focus on the interaction with the system i.e. the expected interface rather than true intent [52] For example: A user states that his goal is to "log in" he expects to enter a username and password, however this is not his goal. His goal is to authenticate himself to the system and possibly at a higher level to maintain the system security [52]. If we use "identify and authenticate" user as the goal, then we can consider other methods of achieving this.

Larman 52 goes on to provide other examples and introduces the **concrete use-case** as being one which specifies exactly what the system will do (including the UI possibly along with images). These are intended for use in the later stages of requirements.

5.5.2 Guideline: Terse Use Cases

Remove noise words, keep the lists of actions as short and to the point as possible e.g. "System authenticates user" rather than "The System authenticates the user" [52].

5.5.3 Guideline: Black-Box Use Case

Use cases should describe the behaviour of the system as a black-box or to take a more OO metaphor we should describe the responsibilities of the system [52]. What we're trying to avoid here is say how the system will work

before we've designed it, at this stage we need to focus on **what** then in the design we can work out the **how**.
Example (from [52]): "The system records the sale." rather than "The system writes the sale to a database" or even worse "The system generates and SQL INSERT statement ...'

5.5.4 Guideline: Take an Actor and Actor-Goal Perspective

The formal definition we looked at earlier (in 5.3.2 mentioned "yields an observable result that is of value for Actors or other stakeholders" this is also found in the original definition from their creator Ivar Jacobson [52]. It may seem obvious to focus on what the users (actors) want and to consider what they value from the system, however many projects have failed after not doing so [52].

Larman 52 suggests that approaches like feature lists have facilitated this disconnect between what the user wants and what the software will. He concedes however that in the vision document a summary of features can be useful and that for some projects such as a middle-ware and libraries being licences for use in other products (e.g. sales middle ware, render engines etc).

5.5.5 Creation: Steps to Identify Use Cases

Given everything we've written about the importance of actors and their goals, the procedure for developing them (suggested by Larman 52 is):

1. Choose the system boundary: Software application? Software Application + hardware or whole organisation (see 5.5.6

2. Identify primary actors: These are the actors who have goals satisfied by services provided see 5.5.6.

3. Identify the goals for primary actors: see 5.5.6.

4. Define use case which satisfy the goals: We name them according to the goal, usually there is a one to one mapping between goals and use case but this is not always the case (e.g. we group Create, Retrieve, Update, Delete (CRUD) goals under Manage e.g. ManageUser.

This is of course iterative and evolutionary not all goals or use case will be discovered at the start it will evolve though discovery [52].

5.5.6 The System Boundary

It may be helpful to define what is outside e.g. primary actor and supporting actors [52]. E.g. from the ATM example, is the bank part of the system? There are a large number of systems where actors who are stakeholders in the system do not interact with it directly, for example the Cashier in a POS system uses the till to serve a customer. The customer never interacts directly with the system, contrast this with on-line shopping? If we were drawing a business use case (kite-level) the system may be whole business and customers and suppliers listed amongst the actors. In the user goal (sea-level) use case for the POS, the customer is actually an off stage actor (or was until we got chip and pin!)

Finding Actors and Goals

Larman 52 stresses that this is a fluid process and its artificial to try to find all the actors, then the goals, then the use case. Sometimes goals reveal actors as do use case.

The following questions may help [52]:

- Who starts and stops the system?
- Who does the system administration?
- Who does user and security management?
- Is "time" an actor because the system does something in response to a time event?
- Is there a monitoring process that restarts the system if it fails?
- Who evaluates the system activity or performance?
- How are software updates handled (push or pull update)?
- In addition to human primary actors are there any external software or robotic systems which require/provide services from the system.
- Who evaluates the logs and how are they accessed?
- Who gets notified when there are errors or failures?

There is an alternative list in Douglas' book, but Mike has borrowed it. I'll update the notes when I get it back!

Documenting Actors and Goals

While the actors and their goals are expressed in the use case, its often helpful to note these as we're thinking about them. Larman 52 recommends either drawing a use case diagram to show the actor with their goal named as a use case or drawing up and actor goal-list (a table with a column for actors and a column for goals). The latter may become a section in the Vision document.

I'd recommend doing this at to help you with the project, the coursework has some off stage actors whose needs are quite important.

Elicitation Note

The key questions here are not "What does the system do?" but "Who uses the system and what do they want to get from it?". This shift in thinking in modern SE approaches is well documented else where in the notes, however Larman 52 recommends two questions (for interview/requirements workshop):

- "What do you do?" (task orientated)
- "What are your goals whose result have a measurable value?" (goal orientated)

Answers to the first question are most likely to gain answers connected to the current system and how it works including its flaws, though with further questioning regarding the reason these task are carried out should be OK [52]. The second question (with some explanation possibly,) combined with some goal hierarchy (i.e. what is the root-goal,) should get more useful information, allow the possibility of new solutions, increase business value and get close to reflecting stakeholder demands [52].

In addition to speaking to our stakeholder we can also consider events which affect the system and how the affect it [52]. This can apply to any system, e.g. we could consider customer arrives at till as an event we can also consider more unusual events e.g. fire alarm sounds. This may be more useful for students looking at developing games and other real-time software where events such as timers can be used to update the system or re-draw the screen.

5.5.7 Checking: Tests to ensure we've found 'useful' use case

In terms of validity Larman 52 suggests that provided the use case starts with a verb (a doing word) its difficult to write an invalid use case at some arbitrary level. I would tend to agree, however I've seen some travesties produced over the years I've been teaching this module. The most common mistake is to try to include user behaviour in the system (rather than express their goal) e.g. rather than "Accept Cash Payment" or "Pay with Cash" they go for "Take Cash from Pocket".

A more practical question is what is a useful use case for the purpose of systems analysis, here there are several rules of thumb [52]:

- **The Boss Test** - If you were asked what you'd been doing all day and you gave the name of one of the use case. Would it satisfy your boss, i.e. would he consider the time spent off value to the business.
- **Essential Business Process** - Something which is done by an individual at a time which provides value to the business and leaves data in a consistent state. This is probably not a single step, does not take days and at the end of it the system is back in a stable state.
- **Size Test** - Its unusual for use case to involve a single step, consider the fully dressed form it should be 3-10 pages. Its most likely the case that this should be part (i.e. via extend or include) or a larger use case.

Reasonable exceptions are common, a use case which has been factored out due to repartition e.g. "Pay by credit card" for example, "Authenticate User" may fail the boss test but due to security concerns will require careful analysis [52].

5.6 Use-Case in UP

Use-case are used in a large number of methods, but are the driving force in UP they are [52]:

- The primary source of functional requirements
- A key part of the iterative planning process, scenarios or use case will be selected as targets for an iteration which will design and develop a part of the software.

- They drive the design. Collaborations of object are used to produce use case realisations which provide services to satisfy actor goals.
- The often influence the structure of user manuals.
- Provide test cases for system testing (of high level functionality), satisfying these may drive development in TDD.
- UI "wizards" are often created to facilitate the most common scenarios of important use case.

5.6.1 Use-Case - In Inception

Use-case evolve across the phases and iterations of UP we'll start writing code when only around 10% of the requirements are detailed [52]. In the inception phase an actor goal list may be produced along with a use case context diagram (actor-goal level for the system,) around 20 use case are identified by name and the most interesting, complexy, or risky are written using the brief format [52].

Following this the 10-20% of the most complex/architecturaly significant or risky (in some way,) use case are written in their fully dressed format [52]. This analysis aims to find any hidden detail, complexity etc., note that we're talking about 10-20% of around 20 use case. Essentially 2-4 fully dressed in total!

5.6.2 Use-Case - In Elaboration

The elaboration phase will have a small number of time-boxed iterations over which the risky, high-value or architecturally signification parts of the system will be incrementally build [52]. Designing and developing the software forces us to think about the problem and this leads to the discovery of more requirements, at regular intervals (say every 4 iterations,) there will be another requirements workshop [52]. By the end of the phase the majority (80-90%) of the use case will be completed in full (remember we only had around 20 to start with). During this phase we'll develop parts of the system (finished parts, not prototypes,) and as a result will start to get user feedback [52].

In each requirement workshop we write (those use case which are still only title) and re-write (as fully dressed use-case those which are not) and by the end of elaboration 10% of the use case should be written and have been turned into designs and tested code[52]. A further 80-90% should have been

written in full with any additional non-functional requirements being added to the Supplemental Specification and Glossary.
It is a this stage that we really start use the use case for analysis, we start to look at generalisation, includes and excludes relations to look for common behaviours etc.

Use-Case - In Construction

Use-case writing in this stage should be minimal, the time-boxed iterations are focused on system completion the core/risky/difficult parts of the have been tackled and unless there is a big change (or something we though simple turns out to be a nightmare) we shouldn't need much detailed analysis [52].

5.7 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 5.7.1, then in teams of two produce:
    - A brief use case
    - An informal use case
    - A fully dressed use case
    - A system (or sea-level use case for the application.

Questions to think about (or what kind of exam question could I get on this).

- List and describe the three kinds of textual use-case in increasing order of formality, provide an example for each.
- Name the constituent parts of a use case diagram and provide a labelled example.
- Questions of the kind - here's a scenario write/draw me a use case are also a regular feature in the exam (see tutorial above).

5.7.1 Case Study

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application

must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.
The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

5.8 Summary

In summary use-case are requirements in context which emphasise the perspective of the user and their goals in using the system [52]. They are a powerful tool for the collection of requirements (remember they are one of many,) and in many respects their use in workshops and interviews to facilitate stakeholder involvement is very much in the spirit of Agile software development. This is an advantage few methods have as there is often a learning curve in terms of interpreting diagrams (in the case of Formal Methods e.g. Z, its near vertical).

5.8.1 Text or Diagram

Larman 52 is very clear on the importance of the textural use case components over the diagrams.

There are three textual formats for use case (see 5.3.5):

1. Brief
2. Casual
3. Fully Dressed

Other books discussion Object Orientated Analysis (OOA) are much lighter on the documentation and focus on the models e.g. [106] and [114]. My previous favourite book on OOA [27] is a little more balanced in terms of the importance of the textural element. I'm inclined to agree with Larman, there is a strong case for using story based textural use case as the foundation rather than diagrams. These are a barrier to customer/user interaction and over fixation on their syntactical correctness can distract from the requirements analysis process.

We need to focus on the guidance given for generating use case. While I'd still like to see the use case you write up for maintenance (as part of the coursework,) to be picture perfect using all the right arrows etc. during the Agile Modelling we're going to do in inception and elaboration it just needs to be right enough. That is enough of syntax of the diagrams needs to be correct that when I'm marking or when your working with other students (or one day other developers,) you both understand what your looking at (see 5.4

Remember that UP is an iterative process, there will be a few iterations in the Inception phase, perhaps working with different groups of stakeholders or considering distinct parts of the system. This iterative component is a powerful tool in itself in terms of analysis, it gives our stakeholders practice of telling what they want and us of interpreting their requests.

In terms of this module we're going to get lots of experience of doing the key practices/actives of software development.

5.9 Preparation for Next Week

5.9.1 Essential Reading

In order to spend more time on the tutorial I'd like you to read 6 in full. We'll spend a little time working though the major concepts in class but our focus will be on its practical application.

It may also help to read over the sections on Evolutionary Requirements in UP 4.4 and the relevant chapter of Larman:

Larman (2014f). *Applying UML and Patterns*, chapter Other Requriements, pages 101–123. Prentice Hall PTR

5.9.2 Background Reading

Alistair Cockburn author of the popular book on use-case modelling "Writing Effective Use Cases" has written an interesting article on the development of use case diagrams.

Please read this if you haven't already is based on a section of Larmans book which is avaible from the web. He actually leaves some of what I'm going to cover to a later chapter (omits some diagram syntax to focus on how we make them).

This is my previous favourite chapter analysis using use-case, its more "planned" in flavour as its from a RTS book but Douglass is one of the best technical authors I've found.

Douglass, B. P. (2004c). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Analysis of Real-time Systems, pages

227–331. Object technology. Addison Wesley, 3rd edition
YOU will need to do the preparatory work ahead of next weeks class!

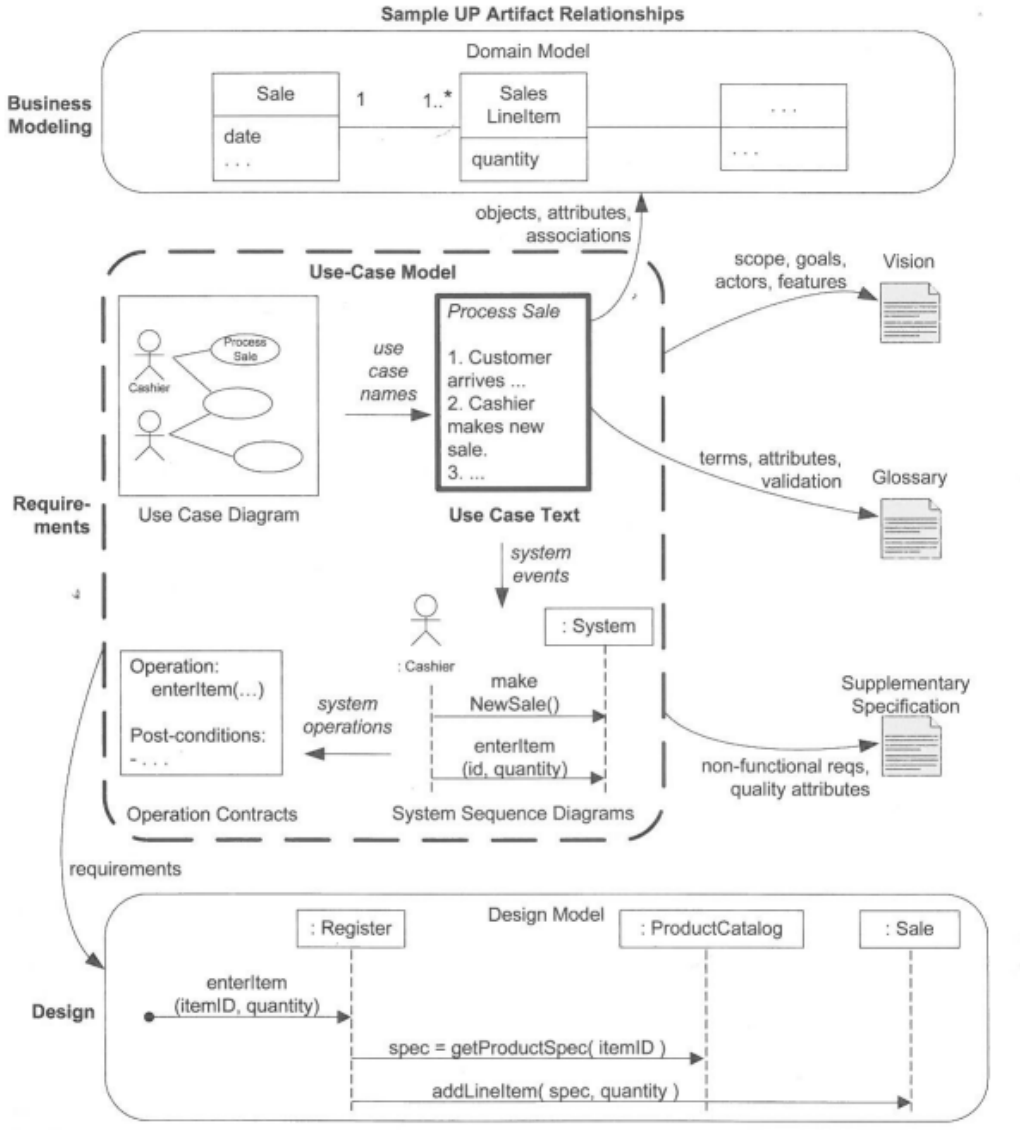


Figure 6.1 Sample UP artifact influence.

Figure 5.2: From Larman [49]: A diagram showing the use case feeding into other artefacts and processes.

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.

Use Case Section	Comment
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Figure 5.3: The template use case form from Larman 52

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Figure 5.4: from Larman [52]: A Brief Use-Case

Handle Returns

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...

Alternate Scenarios:

If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external accounting system, ...

Figure 5.5: from Larman [52]: A Casual Use-Case

Use Case UC1: Process Sale

Scope: NextGen POS application
Level: user goal
Primary Actor: Cashier
Stakeholders and Interests:
– Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
– Salesperson: Wants sales commissions updated.
– Customer: Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.
– Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
– Manager: Wants to be able to quickly perform override operations, and easily debug Cashier problems.
– Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
– Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
Preconditions: Cashier is identified and authenticated.
Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow):
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

*a. At any time, Manager requests an override operation:
    1. System enters Manager-authorized mode.
    2. Manager or Cashier performs one Manager-mode operation. e.g., cash balance change, resume a suspended sale on another register, void a sale, etc.
    3. System reverts to Cashier-authorized mode.

*b. At any time, System fails:
    To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.
    1. Cashier restarts System, logs in, and requests recovery of prior state.
    2. System reconstructs prior state.
        2a. System detects anomalies preventing recovery:
            1. System signals error to the Cashier, records the error, and enters a clean state.
            2. Cashier starts a new sale.

1a. Customer or Manager indicate to resume a suspended sale.
    1. Cashier performs resume operation, and enters the ID to retrieve the sale.
    2. System displays the state of the resumed sale, with subtotal.
        2a. Sale not found.
            1. System signals error to the Cashier.
            2. Cashier probably starts new sale and re-enters all items.
    3. Cashier continues with sale (probably entering more items or handling payment).

2-4a. Customer tells Cashier they have a tax-exempt status (e.g., seniors, native peoples)
    1. Cashier verifies, and then enters tax-exempt status code.
    2. System records status (which it will use during tax calculations)

3a. Invalid item ID (not found in system):
    1. System signals error and rejects entry.
    2. Cashier responds to the error:
        2a. There is a human-readable item ID (e.g., a numeric UPC):
            1. Cashier manually enters the item ID.
            2. System displays description and price.
                2a. Invalid item ID: System signals error. Cashier tries alternate method.
        2b. There is no item ID, but there is a price on the tag:
            1. Cashier asks Manager to perform an override operation.

2. Managers performs override.
3. Cashier indicates manual price entry, enters price, and requests standard taxation for this amount (because there is no product information, the tax engine can't otherwise deduce how to tax it)

2c. Cashier performs Find Product Help to obtain true item ID and price.

2d. Otherwise, Cashier asks an employee for the true item ID or price, and does either manual ID or manual price entry (see above).

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

3c. Item requires manual category and price entry (such as flowers or cards with a price on them):

1. Cashier enters special manual category code, plus the price.

3-6a: Customer asks Cashier to remove (i.e., void) an item from the purchase:
This is only legal if the item value is less than the void limit for Cashiers, otherwise a Manager override is needed.

1. Cashier enters item identifier for removal from sale.
2. System removes item and displays updated running total.

2a. Item price exceeds void limit for Cashiers:

1. System signals error, and suggests Manager override.
2. Cashier requests Manager override, gets it, and repeats operation.

3-6b. Customer tells Cashier to cancel sale:

1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:

1. System records sale so that it is available for retrieval on any POS register.
2. System presents a "suspend receipt" that includes the line items, and a sale ID used to retrieve and resume the sale.

4a. The system supplied item price is not wanted (e.g., Customer complained about something and is offered a lower price):

1. Cashier requests approval from Manager.
2. Manager performs override operation.
3. Cashier enters manual override price.
4. System presents new price.

5a. System detects failure to communicate with external tax calculation system service:
    1. System restarts the service on the POS node, and continues.
        1a. System detects that the service does not restart.
            1. System signals error.
            2. Cashier may manually calculate and enter the tax, or cancel the sale.
5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):
    1. Cashier signals discount request.
    2. Cashier enters Customer identification.
    3. System presents discount total, based on discount rules.
5c. Customer says they have credit in their account, to apply to the sale:
    1. Cashier signals credit request.
    2. Cashier enters Customer identification.
    3. Systems applies credit up to price=0, and reduces remaining credit.
6a. Customer says they intended to pay by cash but don't have enough cash:
    1. Cashier asks for alternate payment method.
        1a. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

7a. Paying by cash:
    1. Cashier enters the cash amount tendered.
    2. System presents the balance due, and releases the cash drawer.
    3. Cashier deposits cash tendered and returns balance in cash to Customer.
    4. System records the cash payment.
7b. Paying by credit:
    1. Customer enters their credit account information.
    2. System displays their payment for verification.
    3. Cashier confirms.
        3a. Cashier cancels payment step:
            1. System reverts to "item entry" mode.
    4. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
        4a. System detects failure to collaborate with external system:
            1. System signals error to Cashier.
            2. Cashier asks Customer for alternate payment.
    5. System receives payment approval, signals approval to Cashier, and releases cash drawer (to insert signed credit payment receipt).
        5a. System receives payment denial:
            1. System signals denial to Cashier.
            2. Cashier asks Customer for alternate payment.
        5b. Timeout waiting for response.
            1. System signals timeout to Cashier.
            2. Cashier may try again, or ask Customer for alternate payment.
    6. System records the credit payment, which includes the payment approval.
    7. System presents credit payment signature input mechanism.
    8. Cashier asks Customer for a credit payment signature. Customer enters signature.
    9. If signature on paper receipt, Cashier places receipt in cash drawer and closes it.

7c. Paying by check…
7d. Paying by debit…
7e. Cashier cancels payment step:
    1. System reverts to “item entry” mode.
7f. Customer presents coupons:
    1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.
        1a. Coupon entered is not for any purchased item:
            1. System signals error to Cashier.
9a. There are product rebates:
    1. System presents the rebate forms and rebate receipts for each item with a rebate.
9b. Customer requests gift receipt (no prices visible):
    1. Cashier requests gift receipt and System presents it.
9c. Printer out of paper.
    1. If System can detect the fault, will signal the problem.
    2. Cashier replaces paper.
    3. Cashier requests another receipt.

Special Requirements:
– Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
– Credit authorization response within 30 seconds 90% of the time.
– Somehow, we want robust recovery when access to remote services such the inventory system is failing.
– Language internationalization on the text displayed.
– Pluggable business rules to be insertable at steps 3 and 7.
– . . .

Technology and Data Variations List:
*a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.
3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
7a. Credit account information entered by card reader or keyboard.
7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:
– What are the tax law variations?
– Explore the remote service recovery issue.
– What customization is needed for different businesses?
– Must a cashier take their cash drawer when they log out?
– Can the customer directly use the card reader, or does the cashier have to do it?

Figure 5.6: from Larman [52]: A FullyDressed Use-Case

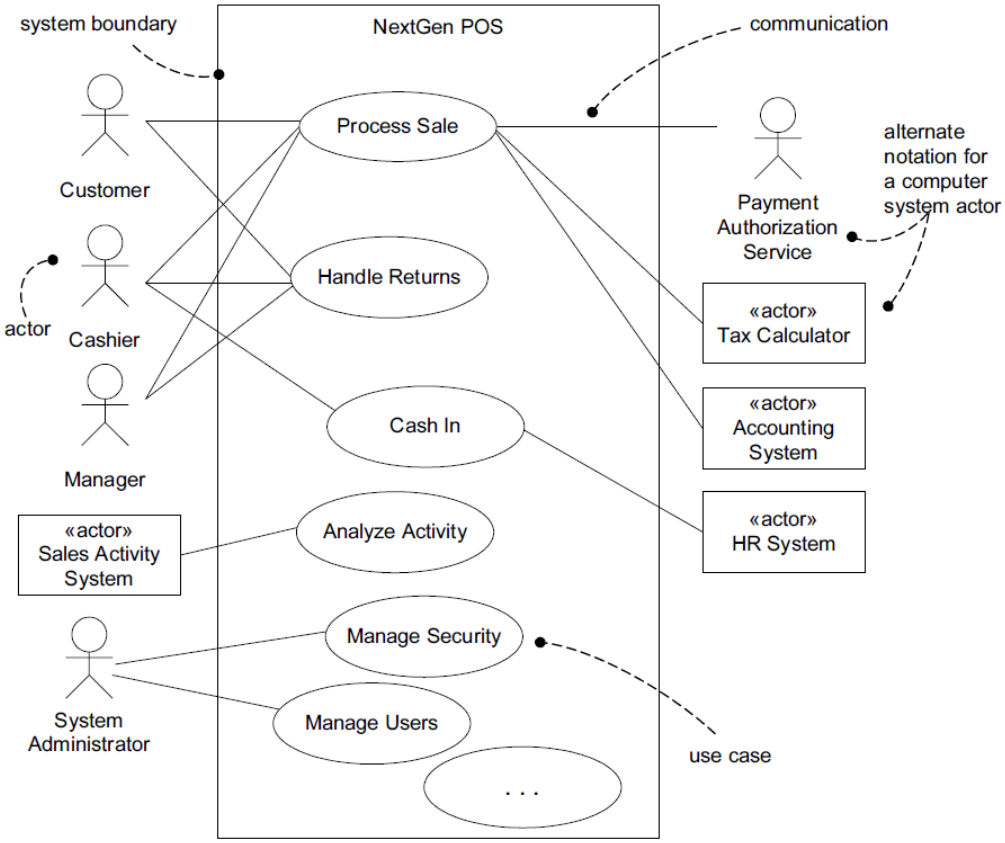


Figure 5.7: from Larman [52]: A partial Use-Case diagram - showing the basic syntax used in high level (context / business model) type use case.

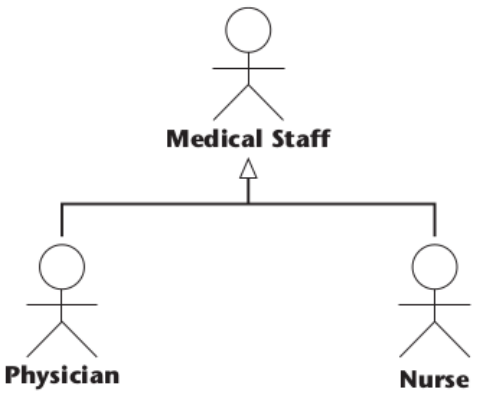


Figure 5.8: from Schach 105: Showing shared properties/behaviours of actors using the generalization (inheritance) relation.

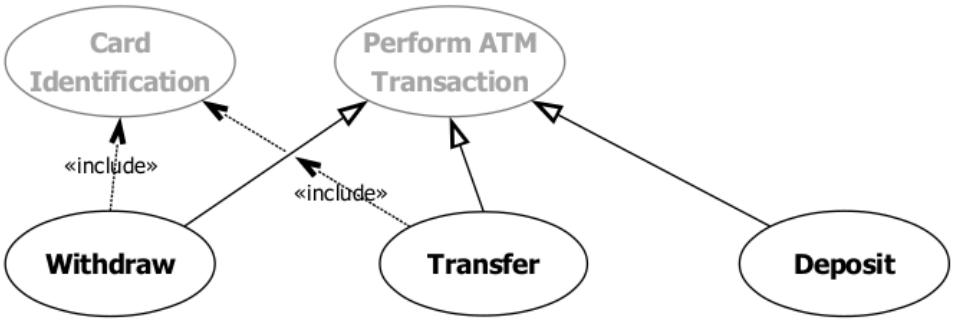


Figure 5.9: from The UML 2.5 Specification [88]: Showing similar behaviours using the generalization (inheritance) relation. Greyed out use case belong to another package in the example - we'll come back to this.

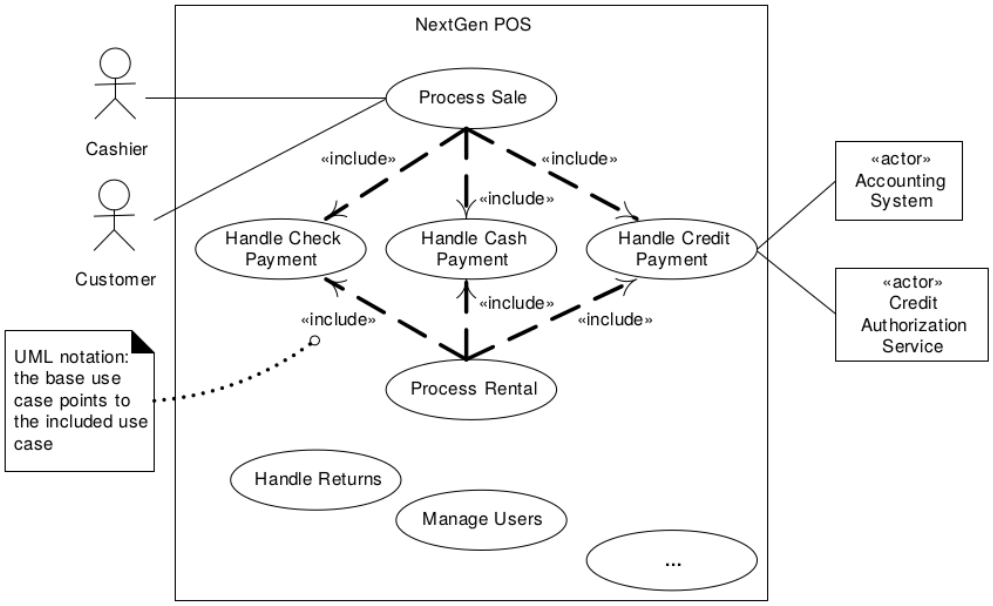


Figure 5.10: from Larman [53]: A partial Use-Case diagram - shown behavioural decomposition with the «includes» relation.

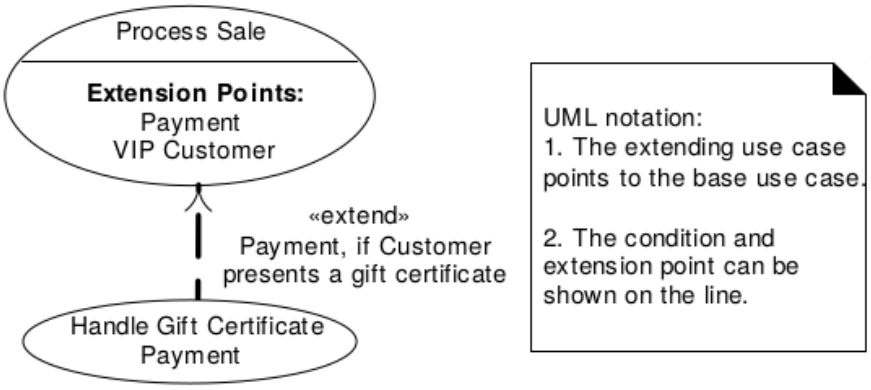


Figure 5.11: from Larman [53]: A partial Use-Case diagram - shown behavioural decomposition with the «extends» relation. Larmans syntax is no 100% here, the condition should be [Customer presents a gift certificate].

Agile UP: Inception (Part 3)

6.1 Introduction

The aim of Inception is to produce an approximate vision (a kind of statement of the overall goal in creating the software), develop the business case (do we want this, will it bring any value) and produce some vague estimates of cost/time-scale.

6.2 Inception - Recap

Inception is a short initial step which establishes a a common vision and scope for the project [50]. Both Business Modelling and Requirements identification will begin in the inception phase and we start to define a number of artefacts (see 4.1. Larman 50 suggest that around 10% of the use case (5.3) will be defined along with critical non-functional requirements (4.3). A business case is produced and the development environment is configured so that programming may begin (in the next phase).

The documents and diagrams produced at this stage will be fairly light and incomplete, e.g. there may be a 'top-level' use case diagram, that is one that shows the system as a black box showing only its most important behaviours [50]. Textural descriptions of the use case which embody the key functions of the system, along with the supplementary specification (which capture constraints on the systems behaviour,) are drafted and are importantly 'work in progress' at this stage [50]. As is the vision which acts as an executive summary of the whole project [50].

Sample Development Case In the development case is shown in 6.1 (from Larman 49) we note that the requirements part maps to four artefacts.

Discipline	Artifact Iteration-*	Incep. I1	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 2.1 Sample Development Case of UP artifacts, s - start; r - refine

Figure 6.1: from Larman [49]: A Sample Development Case

We have now considered inception in general and how the requirements of the system were mapped to different artefacts in UP. Last week we looked at the use case model as the primary artefact for functional requirements, these are usually the easiest to spot and so we'll look at use case models first. We'll now consider other requirements (mostly non-functional) found in the Supplementary Specification (essentially the FURPS+ headings), the high level vision document and glossary.

6.3 Where are the Requirements in Agile UP

The UP offers several requirements artefacts (though as with all UP artefacts they are optional), we'll limit ourselves to the following (as described by [51]):

- **Use-Case Model** – A set of scenarios of using a system, primarily for recording **functional** (behavioural) requirements. **Supplementary Specification** – A record of all non-functional requirements (e.g. performance, licensing etc.) also the place to record any functional diagrams which cannot easily be expressed by use cases (e.g. report generation).
- **Glossary** – Defines noteworthy terms and encompasses the data dictionary. This defines all recorded requirements related to data, such as validation rules, acceptable values etc.

- **Vision** – Summarises high-level requirements that are elaborated in the Use Case Model and Supplementary Specification along with the business case for the document. Essentially provides a short executive overview to orientated those new the the project.
- **Business Rules** – Business (or Domain) Rules describe requirements or policies which transcend individual software projects. For example the data protection act be documented as a set of domain rules. Domain rules may be recorded in the supplementary specification but as these will apply to all projects it makes more sense for them to be central and shared.

Larman 51 suggests that these could be best documented using web pages or a Wiki but also considers wall posters or any other suitable media. The RUP documentation provides templates for all artefacts (though these are optional,) we'll look at using these as a starting point. I've found some here. From the list above its clear we've looked use case in some detail, but not the others. We'll look at these in some detail with examples to help you with the tutorial and coursework. Remember that UP is an iterative and incremental method and as a result there is no final complete reliable specification produced here [54].

6.4 Supplementary Specification

The Supplementary Specification records all those things which cannot be captured in the use cases or Glossary, including system wide quality attributes or requirements [54]. Non functional requirements are in the first instance recorded as part of the use case in the **Special Requirements** section of the fully dressed use case, however moving them to this one document help to ensure they are not duplicated [54]. These concerns will be important when we start to consider our architectural analysis and design, it is often the case that resolving concerns such as reliability etc. must be done the architectural level [54].

6.4.1 FURPS+

In the UP requirements are categorised according to FURPS+ [51]:

- **Functional Requirements**, features, capability and security.
- **Usability**, human factors, help and documentation.

- **Reliability**, frequency of failure, recoverability and predictability.
- **Performance**, response times, throughput, accuracy, availability and resource usage.
- **Supportability**, adaptability, maintainability, internationalisation and configurablity.

The "+" in FURPS+ indicates ancillary and sub-factors such as [51]:

- **Implementation**, resource limitations, languages and tools, hardware etc.
- **Interface**, constraints imposed by interfacing with external systems.
- **Operations**, system management in its operational setting.
- **Packaging**, a physical box, packaging for application stores etc.
- **Legal**, licensing and conformance.

In addition to the FURPS+ a number of other elements may be included [54]:

- Reports which must be generated.
- Hardware and software constraints, operating systems, network constraints etc.
- Development constraints e.g. languages, processes (TDD), tools etc.
- Other design and implementation constraints (accessibility etc.)
- Internationalisation concerns (units, languages).
- Documentation (user, install, administration) and help.
- Licensing and other legal concerns.
- Packaging
- Standards (technical, safety, quality etc.)
- Physical environment concerns (e.g. heat, virbration etc.)
- Operational concerns (error handling, logging, backup routines)

- Application-specific domain rules for example (discount calculation, power-up distribution, auto save rules. etc.) these are local to the application broad domain rules such as Tax laws (VAT) or age restrictions in game content should be in UP Business Rules (6.7);
- Information in the domain so of interest (e.g. credit card payment cycle, tennis rules in a tennis game etc.) This may be in the form of links to important resources (e.g. legal documents,) or other domain resources.

An example is given in 6.2

NextGen Example: (Partial) Supplementary Specification

Supplementary Specification

Revision History

Version	Date	Description	Author
Inception draft	Jan 10, 2031	First draft. To be refined primarily during elaboration.	Craig Larman

Introduction

This document is the repository of all NextGen POS requirements not captured in the use cases.

Functionality

(Functionality common across many use cases)

Logging and Error Handling Log all errors to persistent storage. ***Pluggable Business Rules***

At various scenario points of several use cases (to be defined) support the ability to customize the functionality of the system with a set of arbitrary rules that execute at that point or event.

Security

All usage requires user authentication.

6.5 Vision

The primary purpose of the vision is to ensure that all the major stakeholders (as well as the development team,) share a common vision of the project at the start [54]. It can also be used to familiarise new team members, or those joining the user group with the project [54].

Larman 54 notes that some duplication of other requirements can be expected though normally the vision summarises these. He goes on to suggest that

this can be avoided by referring to the Supplementary Specification and other Requirements Models (use-case) for additional detail. See partial example in 6.3.

6.5.1 Important Sections

There are several key sections within the vision document, amongst these are, the Key High-level Goals and Problems of the Stakeholders and the Summary of System Features [54].

Key High-Level Goals and Problems of the Stakeholders

This section gives a summary of the problems at a high level often summarising a large number of specific use cases [54]. This section may also highlight important non-functional requirements which span the system e.g. "We need fault-tolerant sales processing" [54].

Larman 54 suggests that these are best uncovered using full group techniques such as:

- Mind Mapping
- Product Vision Box creation
- Fishbone diagrams
- Pareto diagrams
- Multi-voting
- Dot voting
- Nominal group process
- Affinity grouping
- Brainstorming
- Brainwriting

I've heard of a few of the above, but some were new to me. I would suggest they're all worth Googling, I'd also have a look at concept and argument maps.

Summary of System Features

Simply listing the use case names doesn't provide an overview of the system (there could be 30-50 use-case) we need to highlight the major features [54]. The reader of the vision may have a vested interest in non-functional requirements, or some optional use case buried in the detail e.g. "Accept Apple Pay" or "Log access to records for security audit purposes" [54]. There are also noteworthy features which span whole systems, for example "The system should make use of third party accounting, inventory and tax calculation systems" [54].

Therefore an alternative is to express these as System Features, high-level terse statements which provide a summary of the systems features [54]. Larman 54 suggest that as these describe functions of the system (though often with some additional quality parameters,) they should pass the linguistic test:

The System does ...

For example:

The System does secure web payments.

Larman 54 suggests that these can be best formatted as a pointed list and there should be around 10 in total. Where more are required these should be grouped (but no groups within groups) around particular facets of the system or abstracted (i.e. combined).

The key point Larman makes here is not too much detail, this has to be high level describing the whole system as a collection of system features.

6.6 Glossary

The Glossary provides detailed information about domain terms which are potentially new to the development team, have a specific meaning in the domain or need to be clarified within the domain [54]. A good example of this occurs when students transfer from other universities, terms like course, programme, module, unit are used differently to denote chunks of study of different sizes. See this example from Larman 54 (see 6.4).

The Glossary also acts as a data dictionary providing information on data

which has to be stored and the required validation rules [54]. This includes both atomic terms such as "Product Price" which may itself be included in composite elements such as "Sale" [54]. Other examples could include "Treatment" and "Patient Record" or "Rucksack" (in an FPS). The example Larman uses is "Payment information Request" which is a nickname for number of data elements (card data, pin data, price etc.)

6.7 Business or Domain Rules

Domain rules dictate how a domain (or business,) may operate [54]. These are not the requirements of an one application though they may influence a number of applications [54], a tangible example of this may be the Data Protection Act [2]. Also included in the possible domain rules are those related to any physical system we may be modelling [54] e.g. flow rates in a water management system, the order of a set of stations in a train monitoring system etc. Other rules may relate to company policy [54], e.g. standardisation of user interfaces or other UX elements across a companies applications or for a section of the business (e.g. games for younger children may be restricted in terms of content). See this example from Larman 54 (see 6.5).

UP uses the term Business rules as this is most common but this doesn't really cover the range of rules which may be included [54]. The important point is that these are independent of the application and may apply to multiple software projects being produced within the organisation or by the company [54].

6.8 Use-case or Vision first?

Larman 54 suggests that defining a rigid order is not helpful, however he recommends:

1. Write a brief first draft of the Vision
2. Start the Glossary (Larman only suggest this is started early, but the sooner the better).
3. Identify user goals and supporting use cases by name.
4. Write some use case in detail and based on the Special Requirements section begin work on the Supplementary Specification.
5. Refine the Vision summarising information from these.

6.9 Evolutionary Requirements and UP

Returning to the context of the UP and its diciplines (workflows) we've spent some time looking at the Requirements dicipline and its artefacts Use-case 5.3, Vision 6.5, Supplementary Specification 6.4, Glossary 6.6, Business rules (Domain Rules) 6.7. In the inception phase nothing is finished and the only produce of the phase is to decide if the project is worth doing (the real analysis work begins in Elaboration (see 8.3 [54]).

During Elaboration these artefacts are developed and continue to be developed over a number of iterations based on ongoing requirements workshops and feedback on completed iterations [54]. Larman 54 suggests that by the end of elaboration it is feasible to have a use case model, vision and supplementary specification which reasonably reflects the stabilized features of the system. At this stage stakeholders may agree what will be completed in the remainder of the project (Construction,) though flexibility will remain. Essentially at the end of Elaboration we should have an idea of "what, how much and when" [54]. At this point we need to ensure we have a rigours requirements change management process in place (with suitable scope for adjusted schedules/budgets) and that we maintain engaged with our stakeholders [54]. There is no sense in being iterative and incremental if we "sign-off" on a set of requirements at the end of elaboration, i.e. no reverting to waterfall! By construction all requirements artefacts should be just about stable and experience only minor changes [54].

6.10 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 6.10.1, then in teams of two produce:
    - A Vision Document
    - A Supplementary Specification
    - A Glossary
    - A List of Domain Rules
    - An Actor Goal List
    - A Context Diagram - system as a single use case

Questions to think about (or what kind of exam question could I get on this).

- Describe the five major artefacts of the Requirements discipline in up your answer should indicate how functional and non-functional requirements are spread across each.
- How are requirements in the UP Supplementary Specification classified, provide an explanation of the classification system with examples based around an example application of your choice. *I'd probably not write a question quite like this, I always tend to give you a short case study and base a number of question around it.*

6.10.1 Case Study

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.
The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The

user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

6.11 Summary

In summary the Vision provides a high level overview of the system and is used both to ensure common ground between developer and stakeholder and to provide short overview for those new to the project.
The supplementary specification is used to record the non-functional requirements and any functional requirements which cannot easily be expressed in a use case.

The glossary defines terms which may be new (or at least, new in this context,) to the developer or ambiguous in the domain.

Finally the business rules are used to define overarching functional/non-functional requirements which transcend the project so that they can be shared amongst projects.

The production of these (as with use case) begins in the inception phase, it will be continued into elaboration.

6.12 Preparation for Next Week

6.12.1 Essential Reading

Larman 55 Provides an overview of what we've done so far and introduces the next phase.

We're hopefully going to move on to consider object models, I'm hoping you'll read the relevant sections of the notes (i.e. those concerned with the theory) so we can simply re-cap them in class and focus on their application.

Larman (2014g). *Applying UML and Patterns*, chapter Iteration 1 - Basics, pages 123–131. Prentice Hall PTR

6.12.2 Background Reading

The following covers the development of object models really well:

Douglass, B. P. (2004d). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Anlysis of Real-time Systems, pages 331–399. Object technology. Addison Wesley, 3rd edition.
The same book has some really good sections on use case modelling, I'm slightly nervous about recommending it as I know how easy it is to get a copy.
Those of you who found this week hard going as you didn't read the chapters before hand I suggest you spend some time re-reading:
Larman (2014f). *Applying UML and Patterns*, chapter Other Requriements, pages 101–123. Prentice Hall PTR

YOU will need to do the preparatory work ahead of next weeks class!

Usability

Human Factors

The customer will be able to see a large-monitor display of the POS.Therefore:

- Text should be easily visible from 1 meter.
- Avoid colors associated with common forms of color blindness.

Speed, ease, and error-free processing are paramount in sales processing, as the buyer wishes to leave quickly, or they perceive the purchasing experience (and seller) as less positive.

The cashier is often looking at the customer or items, not the computer display. Therefore, signals and warnings should be conveyed with sound rather than only via graphics.

Reliability

Recoverability

If there is failure to use external services (payment authorizer, accounting system, ...) try to solve with a local solution (e.g., store and forward) in order to still complete a sale. Much more analysis is needed here...

Performance

As mentioned under human factors, buyers want to complete sales processing very quickly. One potential bottleneck is external payment authorization. Our goal is to achieve authorization in less than 1 minute, 90% of the time.

Supportability

Adaptability

Different customers of the NextGen POS have unique business rule and processing needs while processing a sale. Therefore, at several defined points in the scenario (for example, when a new sale is initiated, when a new line item is added) pluggable business rule will be enabled.

Configurability

Different customers desire varying network configurations for their POS systems, such as thick versus thin clients, two-tier versus N-tier physical layers, and so forth. In addition, they desire the ability to modify these configurations, to reflect their changing business and performance needs. Therefore, the system will be somewhat configurable to reflect these needs. Much more analysis is needed in this area to discover the areas and degree of flexibility, and the effort to achieve it.

Purchased Components

- Tax calculator. Must support pluggable calculators for different countries.

Free Open Source Components

In general, we recommend maximizing the use of free Java technology open source components on this project.
Although it is premature to definitively design and choose components, we suggest the following as likely candidates:

- JLog logging framework
- …

Interfaces

Noteworthy Hardware and Interfaces

- Touch screen monitor (this is perceived by operating systems as a regular monitor, and the touch gestures as mouse events)
- Barcode laser scanner (these normally attach to a special keyboard, and the scanned input is per ceived in software as keystrokes)
- Receipt printer
- Credit/debit card reader
- Signature reader (but not in release 1)

Software Interfaces

For most external collaborating systems (tax calculator, accounting, inventory, ...) we need to be able to plug in varying systems and thus varying interfaces.

Domain (Business) Rules

ID	Rule	Changeability	Source
RULE1	Signature required for credit payments.	Buyer "signature" will continue to be required, but within 2 years most of our customers want signature capture on a digital capture device, and within 5 years we expect there to be demand for support of the new unique digital code "signature" now supported by USA law.	The policy of virtually all credit authorization companies.
RULE2	Tax rules. Sales require added taxes. See government statutes for current details.	High. Tax laws change annually, at all govern-ment levels.	law
RULE3	Credit payment reversals may only be paid as a credit to the buyer's credit account, not as cash.	Low	credit authorization company policy
RULE4	Purchaser discount rules. Examples: Employee— 20% off. Preferred Customer— 1 0% off. Senior— 15% off.	High. Each retailer uses different rules.	Retailer policy.
RULE5	Sale (transaction-level) discount rules. Applies to pre-tax total. Examples: 10% off if total greater than $100 USD. 5% off each Monday. 10% off all sales between 10am and 3pm today. Tofu 50% off from 9am-10am today.	High. Each retailer uses different rules, and they may change daily or hourly.	Retailer policy.
RULE6	Product (line item level) discount rules. Examples: 10% off tractors this week. Buy 2 veggieburgers, get 1 free.	High. Each retailer uses different rules, and they may change daily or hourly.	Retailer policy.

Legal Issues

We recommend some open source components if their licensing restrictions can be resolved to allow resale of products that include open source software.

All tax rules must, by law, be applied during sales. Note that these can change frequently.

Information in Domains of Interest

Pricing

In addition to the pricing rules described in the domain rules section, note that products have an *original price,* and optionally a *permanent markdown price.* A product's price (before further discounts) is the permanent markdown price, if present. Organizations maintain the original price even if there is a permanent markdown price, for accounting and tax reasons.

Credit and Debit Payment Handling

When an electronic credit or debit payment is approved by a payment authorization service, they are responsible for paying the seller, not the buyer. Consequently, for each payment, the seller needs to record monies owing in their accounts receivable, from the authorization service. Usually on a nightly basis, the authorization service will perform an electronic funds transfer to the seller's account for the daily total owing, less a (small) per transaction fee that the service charges.

Sales Tax

Sales tax calculations can be very complex, and regularly change in response to legislation at all levels of government. Therefore, delegating tax calculations to third-party calculator software (of which there are several available) is advisable. Tax may be owing to city, region, state, and national bodies. Some items may be tax exempt without qualification, or exempt depending on the buyer or target recipient (for example, a farmer or a child).

Item Identifiers: UPCs, EANs, SKUs, Bar Codes, and Bar Code Readers

The NextGen POS needs to support various item identifier schemes. UPCs (Universal Product Codes), EANs (European Article Numbering) and SKUs (Stock Keeping Units) are three common identifier systems for products that are sold. Japanese Article Numbers (JANs) are a kind of EAN version.

SKUs are completely arbitrary identifiers defined by the retailer.

However, UPCs and EANs have a standards and regulatory component. See www.adams1.com/pub/rus-sadam/upccode.html for a good overview. Also see www.uc-council.org and www.ean-int.org.

Figure 6.2: An example supplementary specification (incomplete) from Larman [54]

Vision

Revision History

Version	Date	Description	Author
inception draft	Jan 10, 2031	First draft. To be refined primarily during elaboration.	Craig Larman

Introduction

We envision a next generation fault-tolerant point-of-sale (POS) application, NextGen POS, with the flexibility to support varying customer business rules, multiple terminal and user interface mechanisms, and integration with multiple third-party supporting systems.

Positioning

Business Opportunity

Existing POS products are not adaptable to the customer's business, in terms of varying business rules and varying network designs (for example, thin client or not; 2, 3, or 4-tier architectures). In addition, they do not scale well as terminals and business increase. And, none can work in either on-line or off-line mode, dynamically adapting depending on failures. None easily integrate with many third-party systems. None allow for new terminal technologies such as mobile PDAs. There is marketplace dissatisfaction with this inflexible state of affairs, and demand for a POS that rectifies this.

Problem Statement

Traditional POS systems are inflexible, fault intolerant, and difficult to integrate with third-party systems. This leads to problems in timely sales processing, instituting improved processes that don't match the software, and accurate and timely accounting and inventory data to support measurement and planning, among other concerns. This affects cashiers, store managers, system administrators, and corporate management.

Product Position Statement

—Terse summary of who the system is for, its outstanding features, and what differentiates it from the competition.

Alternatives and Competition...

Stakeholder Descriptions

Market Demographics...

Stakeholder (Non-User) Summary...

User Summary...

Key High-Level Goals and Problems of the Stakeholders

A one-day requirements workshop with subject matter experts and other stakeholders, and surveys at several retail outlets led to identification of the following key goals and problems:

High-Level Goal	Priority	Problems and Concerns	Current Solutions
Fast, robust, integrated sales processing	high	Reduced speed as load increases. Loss of sales processing capability if components fail. Lack of up-to-date and accurate information from accounting and other systems due to non-integration with existing accounting, inventory, and HR systems. Leads to difficulties in measuring and planning. Inability to customize business rules to unique business requirements. Difficulty in adding new terminal or user interface types (for example, mobile PDAs).	Existing POS products provide basic sales processing, but do not address these problems.
...	...	...	...

User-Level Goals

The users (and external systems) need a system to fulfill these goals:

- *Cashier*: process sales, handle returns, cash in, cash out
- *System administrator*: manage users, manage security, manage system tables
- *Manager*: start up, shut down
- *Sales activity system*: analyze sales data
- ...

User Environment...

Product Overview

Product Perspective

The NextGen POS will usually reside in stores; if mobile terminals are used, they will be in close proximity to the store network, either inside or close outside. It will provide services to users, and collaborate with other systems, as indicated in Figure Vision-1.

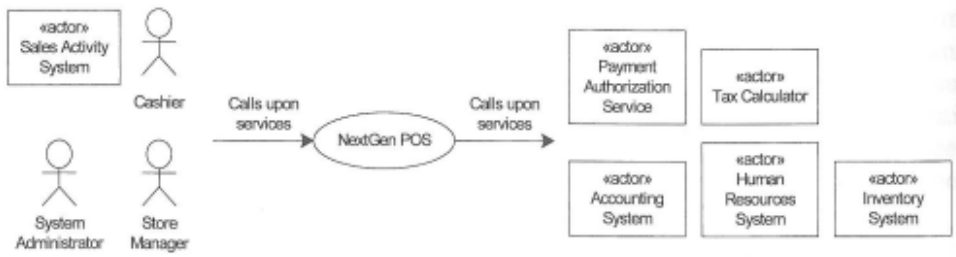


Figure Vision-1. NextGen POS system context diagram

Summary of Benefits

Supporting Feature	Stakeholder Benefit
Functionally, the system will provide all the common services a sales organization requires, including sales capture, payment authorization, return handling, and so forth.	Automated, fast point-of-sale services.
Automatic detection of failures, switching to local offline processing for unavailable services.	Continued sales processing when external components fail.
Pluggable business rules at various scenario points during sales processing.	Flexible business logic configuration.
Real-time transactions with third-party systems, using industry standard protocols.	Timely, accurate sales, accounting, and inventory information, to support measuring and planning.
...	...

Assumptions and Dependencies...

Cost and Pricing...

Licensing and Installation...

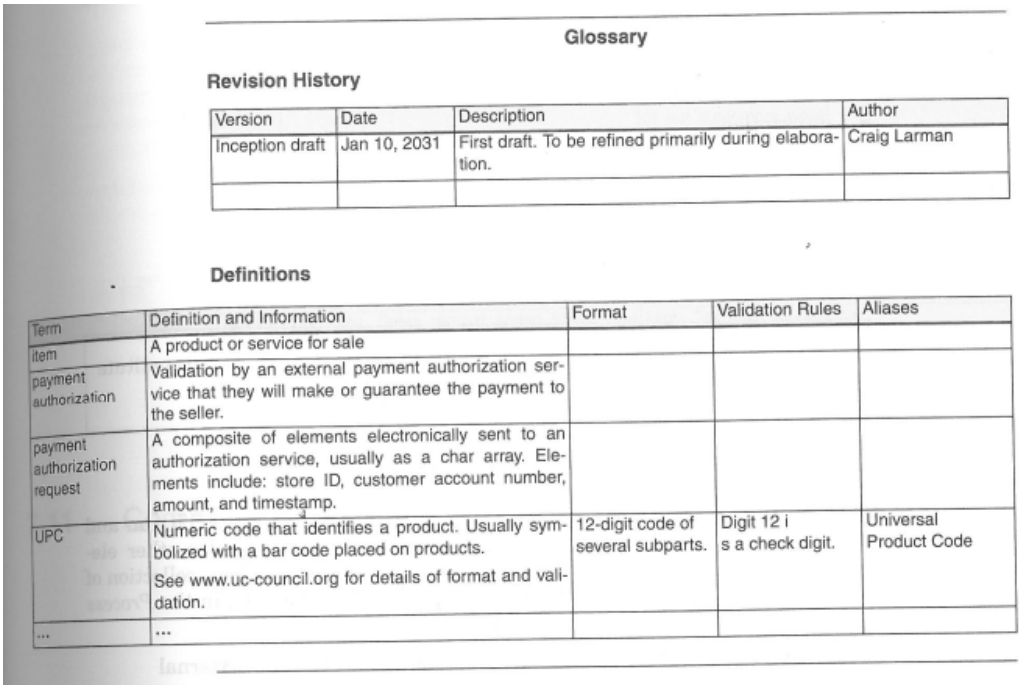
Summary of System Features

- sales capture
- payment authorization (credit, debit, check)
- system administration for users, security, code and constants tables, and so forth.
- automatic offline sales processing when external components fail
- real-time transactions, based on industry standards, with third-party systems, including inventory, accounting, human resources, tax calculators, and payment authorization services
- definition and execution of customized "pluggable" business rules at fixed, common points in the processing scenarios
- ...

Other Requirements and Constraints

Including design constraints, usability, reliability, performance, supportability, design constraints, documentation, packaging, and so forth: See the Supplementary Specification and use cases.

Figure 6.3: An example Vision (incomplete) from Larman [54]



Glossary

Revision History

Version	Date	Description	Author
Inception draft	Jan 10, 2031	First draft. To be refined primarily during elaboration.	Craig Larman

Definitions

Term	Definition and Information	Format	Validation Rules	Aliases
item	A product or service for sale			
payment authorization	Validation by an external payment authorization service that they will make or guarantee the payment to the seller.			
payment authorization request	A composite of elements electronically sent to an authorization service, usually as a char array. Elements include: store ID, customer account number, amount, and timestamp.			
UPC	Numeric code that identifies a product. Usually symbolized with a bar code placed on products. See www.uc-council.org for details of format and validation.	12-digit code of several subparts.	Digit 12 i s a check digit.	Universal Product Code
...	...			

Figure 6.4: A partial Glossary from Larman [54], note this has changed in the new edition of his book and there are more columns (he's not sorted his formatting though).

Domain Rules

Revision History

Version	Date	Description	Author
inception draft	Jan 10, 2031	First draft. To be refined primarily during elaboration.	Craig Larman

Rule List

(See also the separate Application-specific Rules in the Supplementary Specification.)

ID	Rule	Changeability	Source
RULE1	Signature required for credit payments.	Buyer "signature" will continue to be required, but within 2 years most of our customers want signature capture on a digital capture device, and within 5 years we expect there to be demand for support of the new unique digital code "signature" now supported by USA law.	The policy of virtually all credit authorization companies.
RULE2	Tax rules. Sales require added taxes. See government statutes for current details.	High. Tax laws change annually, at all government levels.	law
RULE3	Credit payment reversals may only be paid as a credit to the buyer's credit account, not as cash.	Low	credit authorization company policy

Figure 6.5: A partial Business (Domain) Rules document from Larman 54.

Project and Team Management

7.1 Introduction

In order to draw a line between inception and elaboration and to give us some more time to work though the tutorials we've started to date. This weeks class will be given over to to consider leadership and management. The background theory is quite interesting so I'm providing you some notes to read and I'll provide some small exercises.

7.2 Leadership and Management

Any discussion of management and leadership must first distinguish between the two. This is made difficult by conflicting opinions in the literature which according to Armstrong range from management and leadership being synonymous, management being a subset of leadership and finally the two being distinct [11]. Rollinson 93 indicates that while almost all texts on leadership and management go to some lengths to distinguish between these this is often ignored with the terms leader/manager and follower/subordinate being used interchangeably. He also suggest that much of the research is actually tainted, in that the leaders studied have a degree of 'position power' with which to influence their followers. Many of the studies are therefore of 'headship' rather than leadership and provided at best an indication of how this should be carried out [93].

Armstrong 11 defines leadership as"the process of getting people to do their best to achieve a desired result ... developing and communicating a vision for the future, motivating people and gaining their engagement." This is worth of further study as research has shown that leadership has a direct influence on organizational climate, variations in performance and successful organisational change [11].

	Universal Approach	Situation-Specific Approach
Person-Centered	Trait Theory	Contingency Theory (e.g. Fielder's Theory) Substitutes for Leadership
Behaviour-Centered	Style Theory and Derivatives	Transformational Leadership The Attributional Approach

Table 7.1: Theories as a matrix (from Rollinson 93).

7.2.1 Theories of Leadership

The theories can be categorised in a number of ways; based on their breath of application whether they are universal approaches (reflecting the assumption that one management theory can be used in all situations,) and the more pragmatic situation-specific approach [93]. These theories can also be can also be categorised according to their focus being either person centred (how the relatively fixed personality traits of the individual influence their management,) and behaviour centred (how patterns of behaviour within leaders influence their management) [93] see 7.1.

Trait Theories

Trait theories state that leaders have personality characteristics which are either inherited or developed in early life which result in them becoming leaders in most situations [93]. These theories make the assumptions that [93]:

1. In order to be an effective leader an individual must have certain personal traits.
2. These traits are stable and transferable to any situation.
3. Traits can be easily identified and measured allowing the prediction of leadership abilities.

Studies attempting to identify leadership traits usually find that traits are a poor predictor of leadership emergence and the original trait theory was heavily criticised as; producing endless list of traits with no agreement on a single set, incorrectly assuming that leaders who succeed do so because of their traits, incorrectly assuming that success or prominence is due to leadership (could be luck, wealth, connections etc.) and finally traits are personality variables many of the 'traits' identified are actually patterns of

behaviour and such behaviour is a function of a specific situation [93].

Trait theory has not been completely discarded, researchers agree that while there are no universal traits there are some which are likely to be important in a selected range of situations [93]. In addition two characteristics are observed which are far stronger indicators [93] have been identified:

1. The ability to accurately sense the requirements of a situation
2. A repertoire of behaviours that provide a capacity to adapt to the situation

This leads on to style theory, traits are generally agreed to lead to patterns of behaviour (some of the 'traits' identified were patterns of behaviour,) and so research focused on the style of leadership (the behaviour patterns) and their results [93].

Leadership Styles

Initial studies by Lewin and his colleagues identified three styles of leadership [93]:

1. **Autocratic Leaders** – Make all decisions and exert a high degree of control, while the group is prepared to follow they often do not understand why and do not accept responsibility when things go wrong. Group tends to have low morale and may be hostile towards their colleagues or leader.
2. **Democratic Leaders** – Involve followers in decision and delegate responsibility to the group, results in a stronger team spirit, increased responsibility and commitment to implementing decisions and greater satisfaction amongst group members.
3. **Lassez-faire Leaders** – Abdicated from the role of leader while they do not make decisions for the group or exercise control this is not out of a desire to delegate or to encourage participation but due to a disinterest in leading the group. Group performance depends on the group but if the group is not competent and self motivated members opted out of task competition and sought satisfaction though social interaction.

Further studies developed improved metrics for measuring the effectiveness of leadership independently identifying the task completion and follower satisfaction as metrics [93]. Of these the Ohio State Leadership Studies developed a theory that there were two independent dimensions of leadership Consideration (behaviours focused on interpersonal relations within the group) and Initiating structure (behaviour concerned with achieving the groups formal tasks) [93]. Over generalisations of this theory indicate an ideal manager should display a high consideration for both a duel orientation which is considered impossible[93].

The Michigan Leadership Studies by contrast decided that the contrasting dimensions were in fact different styles of behaviour which exist on a continuum stretching from production centred leadership on one side to employee cantered leadership at the other [93]. While these theories redress some of the problems with traits they still assume that one style is best for all situations and do not take into account the interpersonal relationships or the circumstances which surrounds it [93]. Style theories also assume that a manager is a leader and subordinates are followers [93].

Contingency Theories

Contingency theories of leadership acknowledge that there is no single style of leadership applicable in all circumstances and instead attempt to identify variables which make one style more appropriate than others in a given situation [93]. The contingency perspective is underpinned by the assumption that effective leadership requires a leader to adopt a style of behaviour that matches the conditions in which they are to lead [93].

Fiedler's Contingency Theory was the first contingency theory of leadership often referred to as the Least Preferred Co-worker (LPC) model [93]. Fiedler's theory suggests that the most appropriate leader behaviour is that which produces the highest productivity (ignoring follower satisfaction,) this behaviour is a function of the style of the leader and the contextual circumstances of the group [93]. The groups context can be described in terms of three variables, the relationship between leader and members (similar to that between leaders and followers), the task structure (clear or ambiguous) and the leaders position of power (their authority over the group) [93].

Leaders are assessed on a psychological scale based on their orientations to their LPC from a previous project, this is used with a matrix of the con-

text variables to identify the appropriate management style for the task [93]. This approach assumes that the leaders style (assessed using LPC) is fixed and so leaders should be matched to their situation [93]. There are however arguments which suggest that there are additional variables influencing leader effectiveness which should be taken into consideration and that LPC measures only attitude or even personality which are often poor guides to behaviour [93].
Fiedler's suggested leadership behaviour is dependent on the situation and that the favorability of the situation and therefore the leaders role and influence is controlled by a limited number of factors [77]:

1. **Leader-Member Relations** - The extent to which the leaders is trusted and liked by the group and the extent to which they are willing to follow the leader.

2. **Task-Structure** - The clarity of the task is defined for the group, or to which it can be carried out with supervision or within the normal limits of the followers

3. **Position Power** - The powered exerted by the leader over their followers, ability to reward or punish.

Task orientated leaders with low LPC scores (with a directive, controlling style) will be most effective in very favourable (good leader-member relations, structure task, strong position of power) or very unfavourable (power leader-member relations, unstructured tasks, weak position of power) situations [77]. Leaders with an interpersonal relationship orientation (who have a high LPC score,) and a participate approach, will be effective in moderately favourable situations where the variables are mixed [77]. Fiedler's initially suggested that if the leader does not fit the situation then one or another should be modified, in his later work he suggested that managers could be trained to adjust their behaviour or the situation to increase their effectiveness [93]. Although an advance on style theories Fiedler's approach is limited as the LPC test is suggestive of personality but not behaviour, focused on performance (taking no account of follower satisfaction)[93].

Vroom and Yetton produced a contingency model of leadership based aspects of a leaders decision, its quality (or rationality, i.e. The effect on group performance) , its acceptance (the motivation the group members have in implementing the decisions) and the time required to make it [77]. Vroom and Yetton go on to suggest five main management decision styles (grouped as autocratic, consultative and group,) and seven decision rules which can

be built into trees and provide guidance in applying appropriate rules for the leadership style [77]. This theory is also know as the leader-participation model [91].

Path-goal theory holds that an individuals motivation is dependent on expectations such that increased effort to achieve improved performance will be successful and that this will lead to positive rewards and avoid negative outcomes [77]. The initial research was carried out by House who suggested four main styles of leadership behaviour citepMullins2008-chap7:

1. **Directive Leadership** – Followers are give specific directions and expected to follow rules and regulations, this draws parallels with 'initiating structure' in the Ohio State Leadership studies.

2. **Supporting Leadership** – The leader adopts an approachable manner and display concern for the needs and welfare for their followers, this has parallels with 'consideration' in the Ohio State Leadership studies.

3. **Participative Leadership** – The leader consults their followers and evaluating their opinions and suggestions before making a decision.

4. **Achievement-orientated Leadership** – Followers are set challenging goals which aim to improve the performance of the follower and demonstrate confidence in them.

Leadership behaviour is determined by the personal characteristics of the followers which will dictate their perceived satisfaction, or future satisfaction and nature of the task that is whether it is structured, unstructured, routine or novel [77].

This focus on the characteristics of followers as a factor in identifying an appropriate leadership style was also developed by Hersey and Blanchard, who present a form of situational leadership based on the 'readiness' of followers [77]. To clarify this is the extent to which the follower is willing and able to follow to perform the task rather than a personal characteristic of the individual. Something overlooked in many models is that the effectiveness of any leadership style is dependent on the actions of their followers [91].

Readiness (R) is defined as a continuum between low to high [77]

R1 – represents low follower readiness, referring to follower who are unable or unwilling lacking commitment or motivation or perhaps insecure.

R2 – represents low to moderate follower readiness, followers are unable but willing or unable but confident. Followers lack the ability but are motivated to undertake the task.

R3 – moderate to high follower readiness, followers are able but unwilling or able but insecure.

R4 – high follower readiness relates to followers who are either willing and able or able and confident, they are ready to perform the task.

For each of the four levels, there are task and relationship behaviours which should be adapted combining to give four leadership styles (S) to be applied citepMullins2008-chap7:

S1 – telling, emphasizes high guidance but limited supportive (relationship) behaviour, most suitable for R1.

S2 – selling, high amounts of both guidance and support, best for R2

S3 – participating, high amounts of two-way communications and support but little guidance required, best for R2

S4 – little direction or support with low levels of both task and relationship behaviours, best for R4.

This approach has some developmental aspects, there is a suggestion that the leaders should aim to develop ability and confidence among followers [77].

Leader-member exchange (or LMX) argues that due to time pressures and personalities leaders establish closer relationships with a subsection of their followers, this group receives a disproportionate share of the leaders attention are more trusted and receive privileges as such [91]. LMX suggests that followers fall into two groups the in-group are trusted, privileged and receive a greater proportion of the leaders time [91]. The out group b contrast get less attention and have leader follower relations based on formal authority [91]. Exactly how individuals are allocated groups by the leader is unknown, it is possible that leaders select those who are most capable or have similar personality traits to the leader [91].

Neo-charismatic theories (a classification from Robbins 91) have been developed to de-emphasize theoretical complexity and provide a more accessible view of leadership. They stress symbolic and emotionally appealing leadership behaviours and attempt to explain how some leaders achieve extraordinary levels of follower commitment [91]. Key theories are reviewed below.

According to Armstrong 12 Charismatic leaders "have compelling personalities and the ability to rouse people to follow them though sheer force of the impression they make". Such individuals tend to be achievement orientated individuals with good communication skills who take calculated risks towards their goals [12]. They motivate by affecting a change in follower perceptions of the task, offering an appealing future vision, developing a collective identity and increasing belief in the capabilities of both individuals and groups [12]. There is some overlap with Transformational leadership in this final point (as highlighted by Armstrong [12], the second point may overlap with Transactional leaders, in that the appealing future may be seen as a promised reward.
Visionary Leadership goes beyond charisma and is described by Robbins 91 as the ability to create and articulate a credible, realistic, credible and attractive beginning with improvements to the present. This vision if correctly selected and articulated has a substantial motivational effect generating enthusiasm for and energy towards the implementation [91]. The vision must be something which offers opportunities to the followers, it must fit the goals and time-scales etc. of the organizations and be seen as challenging but attainable [91]. Robbins (2001) identifies visionary leaders as good communicators, who express their leaders not just verbally but in all aspects of their behaviour (for example demonstrating commitment to customer services by helping out on the counter at busy times). Finally the ability to extend the vision into different contexts, applying the same vision to different areas of an organization [91].

Transactional leaders establish a link between the expectations of their followers and a reward [12]. The leader requires power in order to evaluate the work of the subordinate, correct, train and reward effective behaviour resulting in an outcome [12].

Transformational leaders empower and encourage their followers to improve, in terms of their performance, effort and commitment [12]. This process requires that followers are; made aware of the importance and value of designated goals and how they must be achieved, encouraged to transcend their own interests in favour of group goals and that the higher order (emotional) needs of the followers are met [12]. Transformation leadership aims to encourage and support followers and assist their personal development showing them trust and respect as individuals [12].

Bass developed the "Full-Range of Leadership Model" bringing together elements of both transaction and transformational management [115]. This

model of leadership includes four transformational components[115]:

1. **Charismatic Leadership or Idealized Influence** – The leaders as a role model, respected and admired by their followers. Visionary leaders (emulated by their followers,) have a clear vision and are willing to take risks.

2. **Inspirational Motivation** – The leader behaviours in such a way as to motivate, generate enthusiasm and challenge followers. Communicate vision, expectation and goals and demonstrate a commitment to them.

3. **Intellectual Simulation** – Leaders actively solicit new ideas, solutions, practices etc. Stimulate others to be creative and never publicly correct or criticize.

4. **Individual Consideration** – Leader pays attention to individual needs and potential for developing others. A supportive climate is encouraged and differences respected, followers interact with the leader communicating their concerns.

In addition this model includes three dimensions of transnational leadership [115]:

1. **Contingent reward** – leader assigns work and rewards progress.

2. **Management-by-exception (MBE)** – The leader monitors the follower correcting where necessary, this can be passive (errors are corrected where they occur) or active (supervised closely so no errors occur).

3. **Lassiez-faire (or non-leadership behaviour)** – leadership behaviours are ignored and no transactions are carried out.

According to Stewart 115 the transnational components deal with the basic needs of an organization, where as the transformational practices encourage commitment and foster change. Bass maintains that though transnational and transformational leadership are at opposite ends of the spectrum they can be complementary [115]. Stewart 115 goes on to argue that the distinction between transnational and transformational leaders is similar to that between leaders and managers. There is also some overlap here with Change-centred leadership as suggested by Ekvall and Arvonen 33.

Situational leaders adopt one of a number of leadership styles to suite their situation, leaders are defined as [12]:

- **Directing** – leaders make decisions, define and supervise roles/tasks,
- **Delegating** – leaders are involved with decisions and problem solving, but permits autonomy and is involved at the followers description. Coaching leaders seek ideas and suggestions from the followers while retaining decision making and role/task definition power.
- **Supporting** – leaders provide greater autonomy to followers including task allocation and processes while facilitating and taking part in the decision making process.

The leadership theories above take it for granted that within a well formed group, leaders are required. There are however examples where groups perform well without formal leadership, for example highly skilled professionals such as barristers and doctors [94]. Research suggests that there are various circumstances in which leaders are not reqired, where there are leadership substitutes (situational factors which provide guidance, or work satisfaction leading to high performance,) or leadership neutralizers (something absent from the workplace which would normally make leadership important and whose absents removes the influence of a leader) [94]. These can be attributed to the characteristics of the followers, the task or the organization [94]. While the need for leadership may not be completely removed its usefulness may be significantly reduced [94]. Rollinson 94 Suggests that this theory of non-leadership another contingency theory which may be of growing importance given the increasing number of employees without direct supervisors due to downsizing and empowerment.

7.2.2 Management

Armstrong 11p9 defines management as "the process of deciding what to do and then getting it done thought the effective use of resources". The resources here may be financial, spacial, temporal or individuals within an organization. The purpose of management is to satisfy a range of stakeholders, from shareholders and customers in profit-making operations, service consumers in the public sector (patients, students etc.), to the donors and community of the voluntary sector [11].

Robbins 91 suggests that management is concerned with bringing order, consistency, structure and monitoring while leadership is concerned with coping with change, developing a vision for the future, aligning people to this vision and inspiring them to succeed. Research highlighted by Armstrong 11, agrees that the key difference is the focus of management on processes, tasks

and set goals along with the dehumanization of individuals as resources to be utilized. Armstrong 11 however suggests leadership is more focused on people, influencing them to complete tasks and fostering their support. There is also a strong emphasis on independent thought, (innovation, long term views, originality etc.) [11].

Zaleznik 123p79 states that "managers see themselves as conservators and regulators of an existing order of affairs with which they personally identify and from which they gain rewards" in contrast leaders "are separate from their environment" and "work in organizations but never actually belong to them". He goes on to suggest that managers view goals as arising out of necessity and tend to adopt an impersonal, if not passive attitude towards them [123]. Perhaps more alarmingly asserts that managers paradoxically seek out activity involving people but fail to empathize with them [123].

Strategic Management

Researchers have identified several modes of strategy making, which often coexist within organisations, the rational model, command model, symbolic model, transactive model, generative model, muddling though and externally dependent [46].

In the traditional rational model strategy is a though out planned process built on analysis, goals, targets and results [46]. Before making a decision as much data as possible is taken into account, drawing on strategic planning departments and possibly external consultants [46]. Most fashionable in the 1970s this is a dominant model in public sector organisations and large companies in stable environments [46]. The command mode is another traditional view of strategy, in which the leader or top management team are responsible for all strategy evaluated at the top level against the alternatives [46]. Research shows few executives actually spend their time thinking about strategies (more often caught up in day to day management,) and so such an approach is more appropriate in small companies influenced by founding entrepreneurs or where strategic weak company is reoriented by a new leader ([46].

The symbolic mode suggests that an organisations core values (often stemming from the organisations founders and leaders) are adopted by the majority of its workforce and have a major influence of the strategy adopted [46]. This involves the alignment of strategy with symbols of the organisations core values (also known as superordinate goals) such as vision (or strategic

intent) and mission [46].

In contrast to the deliberate strategy making in the command and rational modes, the transactive mode involves the small incremental opportunistic changes in strategy developed by lower and middle level managers based on their knowledge of the field [46]. With the support of high-level management innovative approaches are adopted across the organisation it is suggested that such approaches work well [46]. The generative has some similarities to the transactive mode, but involves more spontaneous innovative leaps in strategy from all levels of the organisation [46]. Like the transactive mode the generative mode has an emergent element (the actual changes in strategy), however there are also some deliberate elements (the creation of a culture which encourages innovation) [46].

The muddling though mode is characterised by small incremental changes (similar to the transactive mode), however the driving force tends to be political wrangling between powerful individuals or groups pursing personal goals [46]. This mode often coexists with others and strategy change is at least in some part dependent on the power exerted by the individuals involved in the change, squabbling and rivalry can delay much needed strategic changes [46]. Outside stakeholders such as governments and trade-unions often have exhibit some influence over policy, this is termed the external mode and often coexists with other modes [46]. Public sector bodies and organisations which receive government money are primarily effected by legislation will effect all companies forcing them to do things they would not otherwise have done [46].

A good strategy will be one which fits the environment in which the organisation finds itself corresponding to the survival factors of that environment and the structure of the organisation itself [46]. It should also allow the organisation to distinguish itself from others (either by what it offers or the efficiency which which it does it) and insure sustainability in the long term [46].

7.2.3 Leadership in Groups

Groups are defined as number of people who; interact with each other, are psychologically aware of each other, perceive themselves as a group, purposefully interact towards the achievement of particular goals or aims [93]. Mullins 78 attempts to disambiguate teams and groups suggesting that teams have a limited number of selected individuals managed on a shared or rotational basis were as groups can be any size, are typically non-selected and

with a single fixed leader. The perception of individuals within a team is mutual knowledge and understanding, with roles spread amongst individuals where as teams is focused on their leader and encourage convergence of roles (conformist in nature) [78]. Finally teams have a spirit of dynamic interaction where as groups promote togetherness and persecution of opponents [78].

Some groups are planned and created by management and form part of the formal organisation these are formal groups others arise from social processes and the informal organisation which arises from the interaction for people within the workplace [78]. Formal groups are well understood, groups are formed for the co-ordination of work activities with individuals selected on the basis of defined roles [78] (which by earlier definitions may make them part of a team). Formal groups tend to be permanent (though their membership may change) though temporary formal groups are also found in organisations, the groups goals are identified by management and rules, relationships and norms of behaviour are established [78].

Informal groups are created to satisfy the social and psychological needs of individuals within a workplace based on personal relationships and the agreement of the group's members [78]. These groups may or may not be related to the task and often cross cut the formal structure, or may be comprised of a part of a formal group [78]. Informal groups are established for the following reasons [78]:

1. Perpetuation of the informal group 'culture': To fit within the social structure of the organisation individuals need to broadly subscribe to a group culture (values, norms and beliefs).

2. Maintenance of Communication: Groups actively seek information which affects their welfare and will tap into formal communication sources distributing information to members of the informal group.

3. The implementation of social control: Conformity to group culture is enforced by techniques such as ridicule, ostracism and violence.

4. The provision of interest and fun in work life: Many jobs are boring, with few prospects social outlets relieve stress and boredom.

Roles within organisations are defined as expected patterns of behaviour associated with members occupying particular positions [78]. In terms of teamroles and the roles that individuals must play for successful team enterprises

Mullins 78 cites a number of sources among them Crainer who neatly summarises the roles required as:

Doers	Concentrate on the task at hand.
Knower	Provides specialist knowledge.
Solvers	Solves problems that arise.
Checkers	Ensure product quality and group productivity.
Carers	Ensure the team works as a cohesive social unit.

Belbin's research is also summarised by Mullins [78], it suggested that the best groups were those who comprised a range of roles undertaken by various members and identified the now ubiquitous 9 useful types of contribution or team-roles.

7.2.4 Power

Power is a complex and dynamic concept which is difficult to define, it can be broadly defined as control or influence over the behaviour of other people with or without their consent [79]. Power may also be interpreted as the extent of influence which can be exerted and include influence over objects as well as people [79]. Power may exist but not be used implying that power is a capacity or potential the most important aspect of which is dependency, a person cannot have power over you unless they control something you desire [79].

The Etzioni typology involves three dimensions: power, involvement and compliance (the relationship between power and involvement within an organisation) [79]. Power differs according to the means by which members of the organisation comply; Mullins 79 identifies the following:

- **Coercive power** – Relies on the use of threats, sanctions or physical force.
- **Remunerative power** – Relies on the manipulation of material resources or rewards.
- **Normative power** – Relies on the allocation and manipulation of symbolic rewards, e.g. prestige.

Involvement and the degree of commitment by members of the organisation includes (according to Mullins 79).

- **Alienative Involvement** – Occurs when members are involved against their wishes. A strong negative orientation towards the organisation.

- **Calculative involvement** – Occurs when members are involved but attachment to the organisation is motivated by external rewards, low or negative orientation to towards the organisation.
- **Moral Involvement** – Based on the individuals belief in and values placed on the goals of the organisation, high orientation towards the organisation.

Etzioni's research suggested that these are normally found in the following combinations coercive power with alienative involvement (e.g. prisons,) remunerative power with calculative involvement (e.g. most business firms) and normative power with moral involvement (e.g churches) [79].

According to French and Raven (quoted in Mullins Mullins 77,) there are several kinds of power exerted by managers (as leaders) over their subordinates, Reward power (based on the ability to reward,) Coercive power (based on fear and the ability to punish,) Legitimate power (based on the perception of the leaders role or position,) Referent power (based on the subordinates identification with the leader, including charisma, reputation, respect or esteem) and Expert power (based on the perception of the leaders knowledge or expertise). Finlay (again sighted in Mullins Mullins 77) adds Personal power (the extent to which individuals are trusted by colleagues and subordinates) and connection power (which results from personal/professional access to key people/information). The notion of power is connected to that of authority, Rollinson 95 defines authority as legitimate power to take decisions (a similar definition is provided by Mullins 80).

7.2.5 Conflict

Conflict is a situation in which the behaviour of an individual (or group) of individuals intentionally blocks or inhibits another group (or individual) from achieving their goals [96]. Conflict can be viewed from a number of perspectives.

The Unitarian perspective is a traditional one that sees harmony and cooperation as the natural state of human affairs and conflict as being either mealy disruptive or destructive [96]. The assumption is that all individuals are on the same side, committed to the same goals behind a single leader and either fail to recognise this or have been lead astray by subversive individuals.

The Pluralist perspective views and organisation as groups and individuals who share some common goals but have others that are different are po-

tentially opposed, conflict is therefore perfectly natural and so measures to handle and resolve conflict should be in place such that it does not cause disruption [96].

The radical perspective is a view derived from the Marxist idea that organisational conflicts reflect inherent conflicts in society as a whole that is between those who own the enterprise and those who work there [96]. It assumes managers are agents of the owners, that all conflicts are vertical and does not accept that managers may have their own divergent interests or the existence of horizontal conflicts [96].

The Interactionist perspective is a refined version of pluralism (and identified as the current view by Rollingson,) it suggests that while too much conflict will hamper progress and absorb a great deal of energy, while no conflict indicates a stifling of the impetus for positive change [96]. The key difference between this and pluralism is the idea of an optimum amount of conflict within an organisation [96].

Research suggests that the key to understanding conflict is to understand the styles of behaviour of the individuals, by assessing their assertiveness (desire to satisfy their own goals,) and their cooperation (willingness to satisfy the other parties concerns) [96]. These can be used to identify five distinct styles [96]:

1. **Avoidance** – Unassertive and Uncooperative behaviour in which the other party is avoided preventing visible sides of conflict.

2. **Competition** – Assertive and Uncooperative behaviour in which one or both parties attempt to overwhelm the other.

3. **Accommodation** – Behaviour is unassertive and cooperative one group sacrifices its aims for the other.

4. **Collaboration** – Behaviour is assertive and cooperative both groups attempt to satisfy their goals with a solution that fits both.

5. **Compromise** – Intermediate amounts of assertive and cooperative behaviour satisfaction of some aims for both groups but potentially not all.

7.3 Motivation

Motivation is a term used to explain the behaviour of an individual and has three components which impact on performance: direction (what the person indents), intensity (the effort exerted in undertaking their intentions) and persistence (their willingness continue with a direction despite obstacles) [97].

Motivation is one of several factors which influences the performance of an individual and so by understanding motivation managers hope to control the performance of an individual so they willingly work harder [97]. Early studies by Vroom suggest that performance (how well a task is completed,) is based on three factors[97]:

- **The Environment** – The context, equipment etc. In which the task is to be performed.
- **The Ability of the individual** – Their knowledge and skill level
- **Motivation** - Managers may be able to influence knowledge and skill of employees and the environment but motivation is internal, unobservable and cannot be directly provided or applied [97].

7.3.1 The Motivation Process

The motivation process begins with identification of needs, which are deficiencies between the actual self (an individuals identity in terms of strengths, weaknesses, beliefs etc.) and the ideal self [97]. Needs result in the setting of milestones towards reconciling the ideal and actual self, expectations that a given behaviour will result in the achievement of goals and to motives and drives, the subconscious processes which provide the energy to engage in behaviour towards goals [97]. The resulting behaviours are monitored to ensure they achieve the goals and meet expectations, the goals in turn are monitored to ensure that they result in the satisfaction of needs [97].

7.3.2 Needs and Expectations

Needs and expectations in the workplace are multi-faceted and more complex than a simple exchange of labour for financial reward, it is therefore useful to categorise the rewards [97]:

- **Extrinsic Rewards** – External to the individual in this context provided by the organisation:

  - Including;
    * Tangible – e.g. pay
    * Fringe – e.g. pensions, conditions, security
  - Extrinsic motivation comes from the expectation of receiving extrinsic reward

- Intrinsic Rewards – Internal to to the individual, psychological, mostly intangible, come from the experience of work:
  - Recognition and appreciation of effort, skill or ability
  - A sense of challenge or achievement
  - Making a difference to others Intrinsic motivation comes from the expectation of receiving extrinsic reward

- Social Rewards – Obtained though others
  - Belonging and social conformity
  - Common purpose, team membership, camaraderie
  - A source of both intrinsic and extrinsic motivation

In practice ensuring people receive the right kind of reward is very difficult, for example social activity can be promoted but its not possible to ensure social rewards are given and received [97]. Intrinsic motivation is individual and providing intrinsic rewards and therefore generating intrinsic motivation is difficult therefore there is a general reliance on extrinsic motivators [97].

7.3.3 Barriers

Circumstances can arise where the motivational chain is broken, the first of these is where job-related barriers exist i.e. something about the job itself or its nature stands in the way of an individual being motivated [97]. The second is when goal blocking occurs, this is where an individual is motivated but their goal is made unobtainable leading to frustration, which if severe enough results in one of two reactions [97]. Adaptive behaviour, where an individual tries to work around the blocking issue or non-adaptive behaviour where an individual attempts to shut out the goal blocking conditions [97]. How an individual reacts to goal blocking is difficult if not impossible to predict as it is influenced by the strength of an individuals needs, their personality and situation as well as the nature of the problem [97].

7.3.4 Motivation Theory

Motivation theories have been developed over the last century and make assumptions about people and what drives them [97]. Early thinking based on the Scientific Management approach suggest that individuals are motivated by financial rewards alone, while later approaches consider social aspects [97]. Motivation theories can be classified as either content theories, focused on needs, wants and desires or process theories based which acknowledge that the strengths of needs etc. for an individual at a given time change but focus on the factors influencing these changes and the relation to the resultant behaviour [97].

Content Based Theories

Maslow's hierarchy of needs (often portrayed as a pyramid with five levels) assumes that human need is inexhaustible and that as soon as one is satisfied another arises [97]. The lower three are considered basic needs and the top two higher order needs, briefly [97]:

- **Physiological Needs** – most basic, arise from physical imbalances (e.g. hunger) need to be satisfied at fairly frequent intervals.
- **Security Needs** – a higher level, emotional and physical security, freedom from harm, predictability, fairness and order
- **Affiliation Needs** – Social needs, the need to be part of a group etc.
- **Esteem Needs** – also known as ego needs, how we view ourselves and how we are viewed by others
- **Self Actualization** – the need to realise ones full potential, the drive to achieve or face new challenges

Maslow assumed that with few exceptions needs are satisfied hierarchically starting at the lowest level and once satisfied no longer had a motivational effect [97]. While attractive due to its intuitive nature there are various issues, there is an elitist assumption that some needs are primitive while others are advanced, no attention to cultural differences, little or no conformation from experimental studies and finally based on research with captive monkeys [97]. Maslows theories were extended by Alderfer in ERG theory, this expressed needs as a continuum from the low-level physical needs though social to personal growth [97]. Unlike Maslow, Alderfer suggested that these were not sequential or exclusive and that if an individual did not active a higher level

they would regress refocusing their attention on the level below [97].

In a departure from the idea of hierarchically ordered needs, Herzberg suggested two-factor theory which suggests that people are motivated by things which make them feel good about work but avoid things which make them feel bad [97]. The positive factors are termed motivators and are usually intrinsic in nature, these include the nature of the work itself, achievement, recognition and prospects for growth and advancement [97]. The negative factors are termed hygiene factors a name which alludes to their preventative nature avoiding them reduces dissatisfaction, for example working conditions, procedures etc. [97]. The presents of hygiene factors does not lead to satisfaction mealy a lack of dissatisfaction, likewise the absence of motivators leads to a lack of satisfaction rather than dissatisfaction [97]. Herzberg's approach is popular draws attention job design and providing motivation, however researchers have argued that this approach grossly oversimplifies sources of satisfaction and dissatisfaction and that these can be one and the same [97]. There are also issues with the application of this approach to unskilled work in which the only motivating factor is the pay and where cultural and national differences are considered [97]. The founding research is again suspect as studies are easily contaminated, bias, or bipolar [97].

McCelland's theory of needs assumes that individual needs are a reflection of cultural values acquired in childhood, three powerful needs are acknowledged [97]:

- Need for Achievement (N.Ach.) the need to succeed or excel in areas of significance to the person.
- Need for Power (N.Pow.) the need to control the activities of other people.
- Need for Affiliation (N.Affil.) the need to interact with and be accepted by others.

McCelland argues that training in adulthood can improve need for achievement and produce a stronger workforce this contradicts both psychological research and his own argument that these cultural values are permanent and fixed from childhood [97].

Process based Theories

Process based theories are closely related to cognitive psychology, rather than simply cataloguing what people do these theories focus on the variables which

account for the direction, intensity and persistence of the behaviour [98].

Expectancy Theory is currently the most influential process theory and is actually a group of theories all of which assume that; in choosing a course of action people will chose one which has a positive outcome for themselves, they are capable of identifying the actions which will lead to positive outcome and they will try to behave in a way which maximises favourable returns to themselves [98]. Vroom first articulated expectancy theory in his Valence-Expectency model (VIE) this links Motivational Force, to Performance (immediate behaviour or first level outcomes) and Results (those which flow from the first level) demonstrating the influence of Expectancy (perception that the effort will result in a first level outcome e.g. improved performance) and Instrumentality (preconceptions that first level outcomes will result in second level outcome e.g. improved pay) [98].

The Porter and Lawler Expectancy Model is a development of Vrooms ideas where the motivating force is further developed, they note that even where the valence (strength of an individuals preference) for a second level outcome is high and the instrumentality is high other factors can influence the success of an individuals actions [98]. Their model includes additional variables such as value or reward, effort on task, personal abilities and traits all of which can affect performance [98]. There are also feedback loops with satisfaction feeding back into the valuing other rewards and performance on the task feeding back into perceptions about potential rewards [98]. While this theory is comprehensive and well validated it is difficult to apply in real situations also it assumes that people only make decisions in a rational, objective and well considered way [98].

The concept of equality is well established in social science as a factor that effects work attitudes and behaviour and is important in social exchange theory, in the absence of a absolute criteria of fairness individuals evaluate how they are treated by comparisons with others in similar circumstances [98]. Equity theories of motivation are state simply that an individuals motivation to put effort into a task will be influenced by their perception of the fairness of the reward in relation to that received by others [98]. If an individual compares themselves to another and finds the equality does exist there is no issue, however if there is a perceived inequity the resulting tension leads to one of the following actions; modify inputs (alter the amount of effort they put into the task), modify the outputs (attempt to increase their reward for the task), modify their perceptions of their own inputs/outputs or their perceptions of others, change the person they compare themselves too or leave

the situation [98].

Goal setting theory is to some extent more a motivational technique than a theory which assumes that peoples behaviours are a function of their concious goals and intentions [98]. Two basic features of goals influence goal-directed effort, firstly goal difficulty that is how difficult is the goal to achieve, motivation increases with goal difficulty up to a point at which the goal becomes unachievable resulting in frustration [98]. Secondly goal specificity is the goal sufficiently clarified and explicit, well defined goals with specific outcomes generate greater motivation as it allows for progress evaluation [98]. Other important influences include goal acceptance and goal commitment [98]. Goal directed effort leads to performance (the individual working towards their goals), this is influenced by their individual ability and the training they have received and the support provided by the organisation [98]. Ultimately goals will be archived leading to intrinsic and extrinsic rewards and satisfaction [98]. While the principles of this technique are widely applied as part of performance appraisal schemes (and there is some evidence that including individuals the goal setting process aids acceptance and increases commitment,) there are a number of criticisms [98]. These include goals being accepted as motivators without question, difficulties converting quality to a quantitative measure for objectives, balancing of conflicting goals and finally some studies have suggested that goal setting can result in unethical behaviour [98].

Behaviour modification theory is an application of Skinnerian operant conditioning and is a technique applied to humans in work situations with the aim of shaping and fixing behaviour in to a pattern than is desired by a manger [98]. Essentially this process involves identifying behaviour which needs to change, measuring current performance, ensuring exceptions, rewards and punishments concerned with required behaviours are understood, evaluation and continual maintenance [98]. There is some evidence that this approach can have short term effects, however lasting change is unlikely [98]. There are some concerns that this approach takes no account of social norms in the organisation, individuality or the ethics of promoting a technique which promotes control [98]. Like some dated leadership approaches [ref] it subscribes to the idea that management are infallible and worse still have an unquestioned right exercise control over everything [98].

Robbins 92 suggests that when motivating professional individuals special considerations have to be made. Professionals tend to be well paid and receive a great deal of satisfaction from their work, their commitment to

their profession requires that they regularly update their knowledge and often work unpaid overtime [92]. The chief motivating factors of professionals tend to be the challenge of the job and the job itself, they value support and a shared sense of their work being important [92]. Robbins 92 recommends that professionals should be challenged, provided with autonomy and rewarded with educational opportunities, recognition and engagement. He also notes that a number of companies employing professionals offer alternative career paths to management for example senior scientists at IBM, offering similar pay to management rolls but without authority or responsibility [92].

7.4 Organisations

7.4.1 Organisational Culture

A culture is a pattern of basic assumptions, invented, discovered or developed by a group as it learns to cope with his problems of external adaption and internal integration, which has worked well enough to be considered valuable and is therefore taught to new members as the correct way to perceive, think and feel in relation to problems and situations [99].
Mullins 81 reports on an early approach by Handy (extending the work of Harrison,) which suggested the existence of four main types of organisation. A power culture is one in which depends on a central power source with rays of influence from a central figure throughout the organisation, which relies on trust, empathy and personal communication to be effective [81]. While there is little bureaucracy, decisions are made on the basis of influence and encouraging a very political climate [81]. Role culture is stereotyped as a bureaucracy based on logic and rationality, strong organisational 'pillas' (functional components e.g. marketing, finance, production etc.) are controlled by rules, procedures and overlooked by managers [81]. Role or job description is often more important that the individual and position is the main source of power [81].
Task culture is job or project orientated, and consists of a net like structure of individuals with links of varying strengths [81]. Power is based more on expertise than position or personality and influence is in the intricacies of the links between individuals [81]. In a person culture the individual is the central focus and the structure exists to serve the individuals within it. Although found in only a few organisations it is the personal preference of most, management structures exist by mutual consent as individuals have almost complete autonomy [81].

A more recent view of organisational culture (from Schein) suggests that culture centred on a number of basic assumptions, these are fundamental and most people in the organisation (consciously or not) subscribe to them [99]. Surrounding the basic assumptions (putting them into practice,) are a number of values and believes, these consist of consciously held reasons for people behaving as they do (for example moral and ethical codes) [99]. The most visible manifestation of culture are the artefacts and creations these include the language (courtesy, use of buzzwords), norms (conformation to codes of behaviour), myths and stories (vehicles to communicate core values), rites and ceremonies (rituals for inducting new staff or saluting retiring ones) and taboos (often humorous stories of what should not be done) [99].

According to Martin (quoted in Rollinson 99) are three perspectives on the pervasiveness of cultures though an organisation. The intergrationalist perspective suggests that the culture is an organisation wide phenomenon [99]. The fragmentationist perspective suggests that there are so many ambiguities, inconsistencies and changes that a single culture cannot exist[99]. Finally the differentiationalist perspective suggests that while an organisation has an overall culture there are variations within groups. These perspectives may all be held by different individuals within a company [99] those in management may be believe there is a single culture amongst all its employees while others may see differences in their departments culture compared to others.

Peters and Waterman identified several characteristics of organisational culture which influenced performance [99]. These are 'Bias for Action' (decisions made even in the absence of complete information,) 'Staying Close to Customers' (customers regarded as the most valuable source of information), 'Autonomy and Entrepreneurship' (company divided into units to foster innovation and initiative), 'Productivity though People' (people should be treated with dignity and respect and given opportunities), Hands on Management (managers seen walking the floor), 'Sticking to Knitting' (company avoids entering areas outside its competence and experience), 'Simple Form' (flat structures with few levels of management) and finally a Simultaneous loose-tight organisation (an organisation tight in terms of goals and values but loose in terms of rules and regulations) [99].

Goffee and Jones provide a Contingency Framework for organisational culture which has replaced the excellence framework (suggested by Peters and Waterman and their contemporaries,) which naively suggested a single best culture for all companies [99]. Goffee and Jones work from the assumption

that there is no right or best culture for all organisations but an appropriate culture for an organisation which helps it cope best with the exigencies of its environment [99]. Their approach works by measuring the Sociability (degree of friendliness between members of the community,) and the Solidarity (the degree of collectiveness, shared goals, tasks, mutual interest of individuals within the community) [99]. The relative amounts of these give rise to the following classifications the networked culture (high sociability low solidarity), the fragmented culture (low sociability and low solidarity), the mercenary culture (high solidarity and low sociability) and the communal culture (high solidarity high sociability) [99].

7.4.2 Organisational Structure

Organisational structure is the fundamental and relatively unchanging features of an organisation which are official sanctioned by those who control it and dictate how activities are undertaken and component parts are grouped, coordinated and controlled to achieve the desired outcomes [95]. Structure is essential for any organisation as all organisations need to compete certain tasks, tasks are divided into subtasks and responsibility allocated to organisational units [95]. As a minimum a structure should; ensure tasks occur in the correct order, allow activities to be monitored, ensure decisions are taken making adjustments to ensure goals are achieved, ensure the allocation (and acceptance,) of responsibility and monitor (and control) the use of resources [95].

Organisations have a structural configuration (or grouping) which can be described in terms of two features, horizontal differentiation (concerned with the division of labour,) and vertical differentiation (concerned with the integration and coordination of parts) [95]. Horizontal differentiation takes a number of forms [95]:

Functional Grouping	Groups activities together according to the major functional specialism of the groups, e.g. marketing, finance, production etc.
Process Grouping	A variant of functional grouping where the groups are based on a narrower range of specialist skills, e.g. engine manufacture, transmissions manufacturer, payroll, sales etc.
Product or Service Grouping	Groups are constructed according to product, such that each product has its own group from crosscutting disciplines.
Market or Customer Groupings	Groups are based on a market or customer group, e.g. local haulage, long-distance haulage, international haulage etc.
Geographic Grouping	This is a variant of market grouping where the markets served are in different geographical areas.
Matrix Grouping	Attempts to gain the benefits of both a functional and product grouping, functional groups divide their time between product based projects controlled by a project manager.

Horizontal differentiation divides up the overall task, by contrast vertical differentiation establishes a hierarchy of authority responsible for the integration and coordination of subtask [95]. This gives the characteristic pyramid shape of an organisation leads to a scalar chain, the direct line of responsibly from top to bottom [95]. This is also bound to the average span of control which gives an indication of the number of subordinates reporting at each level of the hierarchy, a wide average span is associated with flat organisations while a narrow span is associated with tall organisations [95]. Wide spans of control provide more autonomy for subordinates as this makes them more empowered, their jobs more challenging and interesting they are currently in vogue [95]. They also represent an opportunity to reduce the levels of middle management and reduce costs [95].

7.5 Summary

There is a great deal of research into leadership and management and various other connected areas. Being aware of the general areas and some of the key issues may have some benefit in the group project but will be more relevant when you leave and get jobs. In my experience some knowledge of the theory and recommended practices allows us to make sense of organisational structures, cultures, practices etc. There are also a large number of management training courses which promote practices without providing a link to the theory, this is potentially disastrous. One of the key observations in the above is that the type or leader, manager, motivation, organisational strategy etc., required is based on the situation (business, environment, organisation, people etc.).

7.6 Preparation for Next Week

Next week we're back to the Agile UP and the next phase, Elaboration. The following reading has been carefully selected. I want to spend the next session briefly re-capping the theory before focusing on its application, then getting you to apply it. This will only work if you do the reading.

7.6.1 Essential Reading

Please read sections 8.1 8.1 to 8.4 inclusive for next week. There is also this short introductory chapter:

Larman (2014g). *Applying UML and Patterns*, chapter Iteration 1 - Basics, pages 123–131. Prentice Hall PTR.

This is the stuff we'll be doing, focus on the syntactical elements of the object models, I'm hoping to focus identifying objects and their relations in class.

Larman (2014h). *Applying UML and Patterns*, chapter Domain Models, pages 131–172. Prentice Hall PTR.

7.6.2 Background Reading

Larmans explanations of the diagrams are not the most detailed I've found, this book is much harder to find unfortunately but his descriptions are better.

Douglass, B. P. (2004a). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Object Orientation with UML 2 - Structural Aspects, pages 77–139. Object technology. Addison Wesley, 3rd edition.

Fowler has long produced a reference guide for the diagrams, his aim seems to be to keep this as short as possible. This makes for a good reference text for the syntax of UML, but there's nothing in here on the thinking behind the diagrams.

Fowler, M. (2003a). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Object Model (Part 1)

8.1 Introduction

The aim of Inception was to produce an approximate vision, develop the business case and produce some vague estimates of cost/time-scale. We now move into the Elaboration phase this aims to refines the vision and over a number of iterations produces the core architecture. High risk elements will be resolved, the majority of the requirements identified (but not analysed in detail,) and the the scope of the project clearly defined. More realistic estimates are produced based on a better understanding of the project and developer experience.

8.2 Inception - What just happened?

Inception is a short initial step which establishes a a common vision and scope for the project [50]. Requirements identification will begin in the inception phase and we start to define a number of artefacts (see 4.1. Larman 50 suggest that around 10% of the use case (5.3) will be defined along with critical non-functional requirements (4.3). A business case is produced and the development environment is configured so that programming may begin (in the next phase).

The documents and diagrams produced at this stage will be fairly light and incomplete, e.g. there may be a 'top-level' use case diagram, that is one that shows the system as a black box showing only its most important behaviours [50]. Textural descriptions of the use case which embody the key functions of the system, along with the supplementary specification (which capture constraints on the systems behaviour,) are drafted and are importantly 'work in progress' at this stage [50]. As is the vision which acts as an executive

summary of the whole project [50].

These notes focus on the requirements-orientated artefacts other business/risk orientated output are not considered. These may include prototypes, risk lists, recommendations for purchasing components, candidates libraries/frameworks for re-use are not considered [55]. The rational is that we are focusing on the key software engineering aspects rather than those concerned with risk/business.

8.3 Elaboration

Elaboration is a initial series of iterations over which we undertake three key activities [55]:

- **Development** – We develop the core risky architecture and test it.
- **Requirements** – Discover and stabilise the majority of requirements of the system.
- **Mitigate Risk** – We attempt to mitigate risks (reduce their potential frequency or severity using preventative measures,) or retire them (find that they are no longer a risk).

During elaboration the team undertakes the serious investigation, implementation (programming and testing,) of the core architecture, clarifies requirements and tackles high risk elements [55]. Risk in this context may not related to technical risk (i.e. things that might be impossible given the limits of technology or our knowledge,) but refer to major risks to project success, that is features which if they are not correct will render it useless [55].

Elaboration proceeds over a number of iterations, usually around 4-6 weeks (the shorter the better,) each time-boxed (with a fixed end date,) and focusing on important use-case or a scenario (or part scenario,) from a use-case [55]. Use-cases may or may not not be implemented in a single iteration, we may aim to complete only part of it.

The code artefact from this stage should be production quality (not a throw away prototype,) though some authors use the term "architectural prototoype" to describe the early iterations of the core arhictecture [55]. Larman 55 suggests that a better term is **executable architecture** or **architectural baseline**. He goes on to suggest that in order to ensure that this

baseline is stable particular use-case are chose for early implementation, i.e. a volatile architecture is a key risk we seek to minimise [55].

Key Ideas & Best Practice Larman 55 suggests that there are a some key ideas and best practices which we begin in elaboration. I'll try and point these out to you as we go, but here they are so you'll know them when you see them:

- Short time-boxed risk-drive iterations.
- Start programming early.
- Adaptively design, code and test the key risky (and architecturally significant) parts of the system.
- Test early, test often and make sure the tests are realistic (i.e. consider coverage, etc. - We'll look at testing later in the year).
- Write most use case and other requirements in detail, though a series of workshops, one per elaboration.

Sample Development Case In the development case is shown in 8.1 (from Larman 49) we note that the elaboration phase continues a number of artefacts from the inception phase and begins several other artefacts in other disciplines (i.e. not Requirements). The following sections on elaborate on this.
There are some artefacts which we won't consider in detail which may also start in inception [55]:

- **Data Model** – Includes database schema and any other mappings between object and non-object formats e.g. flat config files, xml or Json files.
- **Use-case Storyboards, UI Prototypes** – Descriptions of the UI, navigation paths, wire frames, usability models etc.

8.3.1 Planning Iterations

Planning and project management are not considered in detail here, we've looked at some of the broad issues of management and group management. Hopefully in a future class we'll have time to look at project management in

Discipline	Artifact Iteration-*	Incep. 11	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 2.1 Sample Development Case of UP artifacts, s - start; r - refine

Figure 8.1: from Larman [49]: A Sample Development Case

some detail.

Larman 55 suggests that the process of scheduling requirements, use-case and features in to iterations requires their prioritisation according to their Risk, Coverage and Criticality, he defines these as:

- **Risk** includes both technical complexity, business value and other factors such as uncertainty, development effort (e.g. getting the architecture right) etc.
- **Coverage** implies that all the major parts of the system are at least touched on early on – perhaps a wide an shallow implementation across many components.
- **Criticality** refers in this case to the functions considered by the client to by high business value.

This ranking of remaining requirements (show in 8.2,) is undertaken between each iteration, the experience of each will inform the next possibly including feedback from the users of the system [55]. The adaptive nature of the process allows for re-focusing to meet changing business need or to cope with unforeseen problems in the development.

Larman 55 notes the that while working on the most architecturally significant we will also work on what he describes as the implicit **Start Up** use-case in each iteration as its is extended to meet the needs of the next iteration.

Rank	Requirement (Use Case or Feature)	Comment
High	Process Sale Logging ...	Scores high on all ranking criteria. Pervasive. Hard to add late. ...
Medium	Maintain Users ...	Affects security subdomain. ...
Low	...	...

Figure 8.2: Based on Larman 55: Ranking Requirements/use-case/scenarios according to above criteria.

8.3.2 UP Disciplines (Workflows)

Returning to the context of the UP and its disciplines (workflows) we will continue with the requirements discipline into reception and the amount of requirements analysis undertaken can be expected to continue [54]. We'll also begin working with other disciplines initially Business (Domain) Modelling, then Design [55]. The classic (R)UP diagram is shown in 8.3, the work-flows are shown horizontally.

Requirements

During Inception we spent some time looking at the Requirements discipline and its artefacts Use-case 5.3, Vision 6.5, Supplementary Specification 6.4, Glossary 6.6, Business Rules (Domain Rules) 6.7. In the inception phase nothing is finished and the only produce of the phase is to decide if the project is worth doing (the real analysis work begins in Elaboration (see 8.3 [54]).

During Elaboration these artefacts are developed and continue to be developed over a number of iterations based on ongoing requirements workshops and feedback on completed iterations [54]. Larman 54 suggests that by the end of elaboration it is feasible to have a use case model, vision and supplementary specification which reasonably reflects the stabilized features of the system. In short the work we've done to date will continue and we'll begin take more of the brief use case and those we have only named to date and analyse them in more detail, creating fully dressed use-case summarised by use-case diagrams.

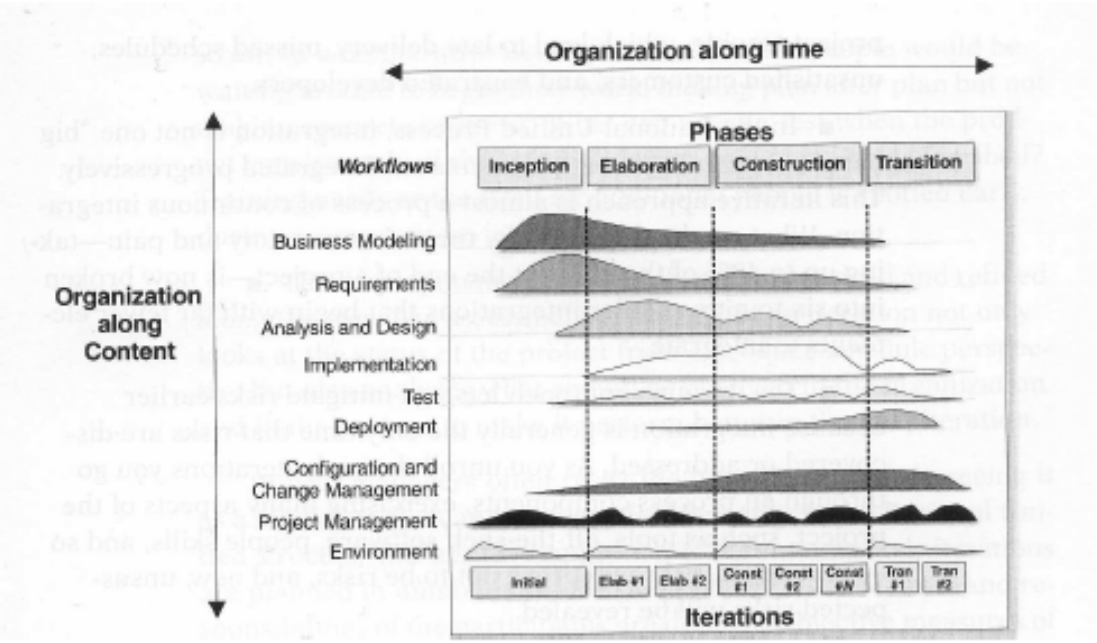


Figure 3.9 Process structure of RUP (*Source: P. Kruchten, The Rational Unified Process, An Introduction*, 2003, Addison-Wesley)

Figure 8.3: from Vilet [119]: RUP diagram

Business Modelling

Looking back to our Development Case (8.1) we note that the Domain Model which we create in this section is the first artefact from the Business Modelling discipline. Larman 56 states that the object domain model is the most important object orientated model in analysis (nothing that use-case, while useful, are not object based). The model illustrates useful concepts within the domain, i.e. things of interest which will need to be represented in software [56]. It also forms the basis of our design models here we attempt to lower the conceptual gap between our problem space and the solution by using similar names e.g. we might represent a Sale with a class called Sale.

In terms of learning this the diagrams (more UML,) see at first to be the most difficult part however the syntax is simple, consistent and very well documented. The most difficult part is using the diagrams correctly to elicit a representative domain model. As with use-case I'm hoping to get you to read ahead in terms of the lengthy explanations of the diagram syntax so we

can focus on using the diagrams.

Design

Also in this phase we'll start working on the design, the two key artefacts here are the Design Model and the software Architecture Document. The design model describes the logical design , including Software class diagrams, object interaction diagrams package diagrams etc.[55]. The Software Architecture Diagram describes how key architectural issues (driven by non-functional requirements,) have been resolved [55]. For example, one of our non-functional requirements could be that the system is highly available (i.e. minimum downtime,) the solution is to use a client server architecture with fail-over, this is an architectural decision which must be documented. The danger being that someone coming to the project may change this without appreciating the implications. Data Models, while a key part of the design will not be considered in any detail in this module as this is something you study elsewhere.

8.4 Domain Models

The Domain Model is used to show noteworthy concepts, bounded by the use-case (scenario etc.,) of the current iteration [56]. Essentially we're looking to identify things (though not always physical things,) we're interested in modelling. For example in a hotel we might include in our model, Room, Guest and Reservation only two of which are physical objects in our problem space.

Larman 56 notes that there are a number of synonyms including conceptual models, domain object models and analysis object models. A closely related terms is that of domain layer (also sometimes called the domain model,) which describes the classes below the user interface which have responsibilities (i.e. contain the logic for the application) [56].

The Domain Model is also related to the Business Object Model or BOM, in that it is a subsection of it. The BOM will show all systems and their interconnections within a company, where as the Domain Model for say the POS system will be limited to a single system [56].

8.4.1 Domain Model as a Visual Dictionary

The Domain Model will probably contain many of the items which are included in the Glossary, however its allows us to visualise and relate concepts in the domain while abstracting key details [56]. For this we use the UML class diagram notation, however we're careful to use only a subset of it omitting the method signatures from each class. The Domain Model provides a conceptual perspective and shows [56]:

- domain objects or conceptual classes
- associations between conceptual classes, e.g. Sale relates to a number of items.
- attributes of conceptual classes e.g. each Sale has a Date and Time.

The Domain Model is not a design diagram, it should have nothing which relates to implementation (e.g. data types, collections,) or to domain layer classes with responsibilities (i.e. methods) [56].

8.4.2 Domain Models and Decomposition

As with many software development (and more generally engineering) methods object orientation promotes divide an conquer [56]. Unlike traditional structured analysis which worked based on decomposition into processes or functions (think back to semester one Introduction to Programming) [56]. Object orientated development relies on the identification and modelling of entities, things (objects,) in the domain which though their interaction (working together,) perform the required functionality of the system [56]. In this respect the identification of suitable objects from the domain is a key part of the object orientated approach and its importance cannot be overlooked.

8.4.3 Conceptual Classes

The Domain Model illustrates conceptual classes or vocabulary, these are essentially ideas, things or objects [56]. More formally they can be considered in terms of:

- **Symbol** - A chosen symbol for a concept, e.g. the English word *Sale*.
- **Intention** - The intention of the symbol, e.g. to represent a purchase transaction has a data and time.
- **Extension** - All the examples of sales, e.g. the set of all sales.

8.4.4 Examples

The example from Larman 56 is show in 8.4. See 8.6 for an explanation of the syntax.

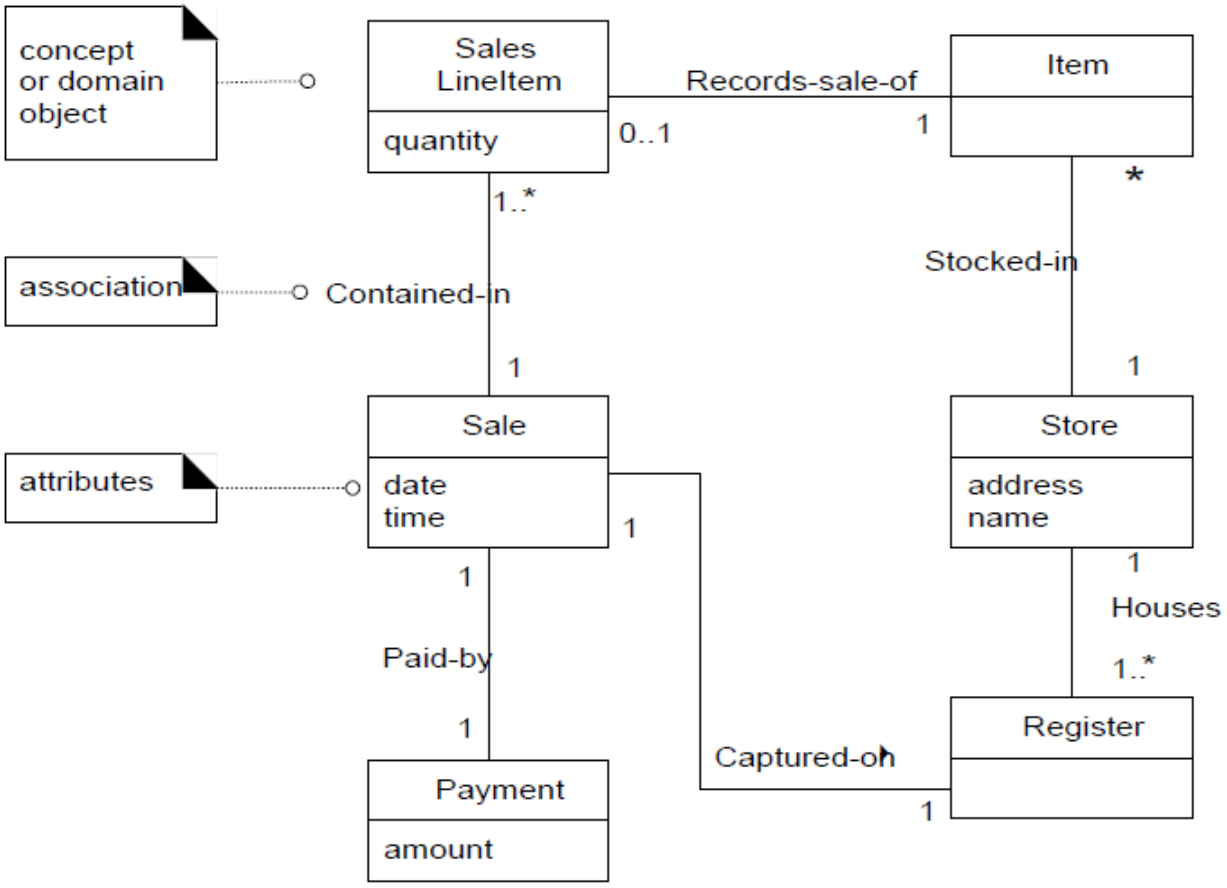


Figure 8.4: An example domain diagram from Larman 56

8.4.5 Lowering the Representational Gap

The representational gap is that between the software model of the system and our mental model of the problem space [56]. One of the advantages of object orientation over other techniques is our ability to create objects in the domain layer (below the interface layer in the software,) to represent those in the problem space [56]. Reducing the Representational Gap makes the software easier to understand and modify [56].

8.5 Domain Diagram Creation

Larman 56 provides some guidance for the creation of the Domain Model. He condenses this to three steps *Bounded by* (i.e. assuming the limitations of the current use-case/scenario etc.):

1. Find the conceptual classes (see 8.5.1).
2. Draw them as classes in a UML class diagram (see 8.6.
3. Add associations (see 9.5) and attributes (see 9.6).

8.5.1 How to find Conceptual Classes

Identifying conceptual classes is often straight forward,identifying them all and not ending up with some attributes as classes etc. is more complicated. Larman 56 suggests the following methods:

1. Reuse or Modify Existing Models.
2. Use a Category List.
3. Identify Noun Phrases.

Reuse or Modify Existing Models

There are a large number of well crafted Domain Models (and Data Models which can be made into Domain Models,) for common domains such as inventory, finance, health etc. [56]. While these are often the first port of call for the seasoned developer we'll start with some more basic methods.

Use A Category List

Category lists are as the name suggests common categories of candidate class [56], perhaps the easiest to consider is "Phyiscal Objects". The categories provided by Larman 56 are focused towards information systems (see 8.5).

8.5.2 Noun Phrase Identification

Linguistic analysis of domain documentation or descriptions of domain operations can be used to identify candidate classes [56]. Blindly mapping nouns to classes is however impossible as there will be some duplication, alternative names for objects in different roles etc., however this is an excellent source

Conceptual Class Category	Examples
physical or tangible objects	*Register* *Airplane*
specifications, designs, or descriptions of things	*ProductSpecification* *FlightDescription*
places	*Store* *Airport*
transactions	*Sale, Payment* *Reservation*
transaction line items	*SalesLineItem*
roles of people	*Cashier* *Pilot*
containers of other things	*Store, Bin* *Airplane*
things in a container	*Item* *Passenger*
other computer or electro-mechanical systems external to the system	*CreditPaymentAuthorizationSystem* *AirTrafficControl*
abstract noun concepts	*Hunger* *Acrophobia*

of inspiration [56]. The fully dressed use-case(s) provide a lot of descriptive text related to the domain and provide an excellent point at which to start this technique, Larman 56 provides a marked up example see 8.6.

Other Techniques

Douglass 29 provides the following useful techniques for object identification see 8.7

organizations	*SalesDepartment* *ObjectAirline*
events	*Sale, Payment, Meeting* *Flight, Crash, Landing*
processes (often *not* represented as a concept, but may be)	*SellingAProduct* *BookingASeat*
rules and policies	*RefundPolicy* *CancellationPolicy*
catalogs	*ProductCatalog* *PartsCatalog*
records of finance, work, contracts, legal matters	*Receipt, Ledger, EmploymentContract* *MaintenanceLog*
financial instruments and services	*LineOfCredit* *Stock*
manuals, documents, reference papers, books	*DailyPriceChangeList* *RepairManual*

Figure 8.5: Based on Larman 56: Categories to aid identification of conceptual classes.

Table 6-1: *Object Discovery Strategies*

Strategy	Description
Underline the noun	Used to gain a first-cut object list, the analyst underlines each noun or noun phrase in the problem statement and evaluates it as a potential object.
Identify causal objects	Identify the sources of actions, events, and messages; includes the coordinators of actions.
Identify services (passive contributors)	Identify the targets of actions, events, and messages as well as entities that passively provide services when requested.
Identify messages and information flow	Messages must have an object that sends them and an object that receives them as well as, possibly, another object that processes the information contained in the messages.
Identify real-world items	Real-world items are entities that exist in the real world, but that are not necessarily electronic devices. Examples include respiratory gases, air pressures, forces, anatomical organs, and chemicals.
Identify physical	Physical devices, including

Main Success Scenario (or Basic Flow):

1. **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale.**
3. **Cashier** enters **item identifier.**
4. System records **sale line item** and presents **item description, price,** and running **total.** Price calculated from a set of price rules.

Cashier repeats steps 2-3 until indicates done.

5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment.**
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions)** and **Inventory** systems (to update inventory).
9. System presents **receipt.**
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

7a. Paying by cash:

1. Cashier enters the cash **amount tendered.**
2. System presents the **balance due,** and releases the **cash drawer.**
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

Figure 8.6: Based on Larman 56: Example of noun phrase identification for objects from a fully dressed use-case.

Table 6-1: *(cont.)*

Strategy	Description
Identify persistent information	Information that must persist for significant periods of time may be objects or attributes. This persistence may extend beyond the power cycling of the device.
Identify visual elements	User interface elements that display data are objects within the user interface domain such as windows, buttons, scroll bars, menus, histograms, waveforms, icons, bitmaps, and fonts.
Identify control elements	Control elements are objects that provide the interface for the user (or some external device) to control system behavior.
Apply scenarios	Walk through scenarios using the identified objects. Missing objects will become apparent when required actions cannot be achieved with existing objects.

Figure 8.7: From Douglass 29: A selection of object discovery strategies.

8.5.3 Class Diagrams at Different Levels

Both Larman 56 and Fowler 40 describe UML being used at different levels, this relates back to our assertion at the start of the section that UML is a language used for a number of purposes. Larman 56 identifies:

- **Essential or Conceptual Perspective** – diagrams are interpreted as representing real world entities and their relations in the problem domain (this is where our Domain Diagram fits). See 8.9.
- **Specification Perspective** – diagrams are interpreted as representing software elements, including classes interfaces, packages etc. but with no reference to the features of the implementation language. We note that the notation used is the same as that for the conceptual perspective, though clearly extended a little.
- **Implementation Perspective** – diagrams are interpreted as representing software elements in a particular language i.e. including the specifics of the language chosen.

8.6 Domain Diagram Syntax

This section begins to introduce the UML (Unified Modelling Language).

It is vitally important to note that the UML is just a language and it is important to distinguish between what Larman 56 describes as raw UML with its applications within methodology defined diagrams. The Domain Diagram uses the class diagram notation, as will the Design Model we produce later in this phase. UML is a language, we use the same words and punctuation to write a report as we do to write a story or a joke. The bad news I suppose is you're going to have to learn UML while learning to use it.

Note: There is a major difficulty here, either I introduce the techniques to identify conceptual classes (classes in the domain model) and then how to represent them (which I have done,) or as I'm going to do with associations show you how they are represented then how they are found.

8.6.1 Basic Class Diagrams

Fowler 41 states that the class diagram describes the kind of objects in a system and the static relationships between them. They show the properties and operations (the UML term is **features**,) of the class along with any

constraints on its creation [41]. Fowler 41 expresses all of these in a single example 8.8, I'll add a little detail but to be honest there are so many excellent and detailed books on the UML finding information should not be a problem.

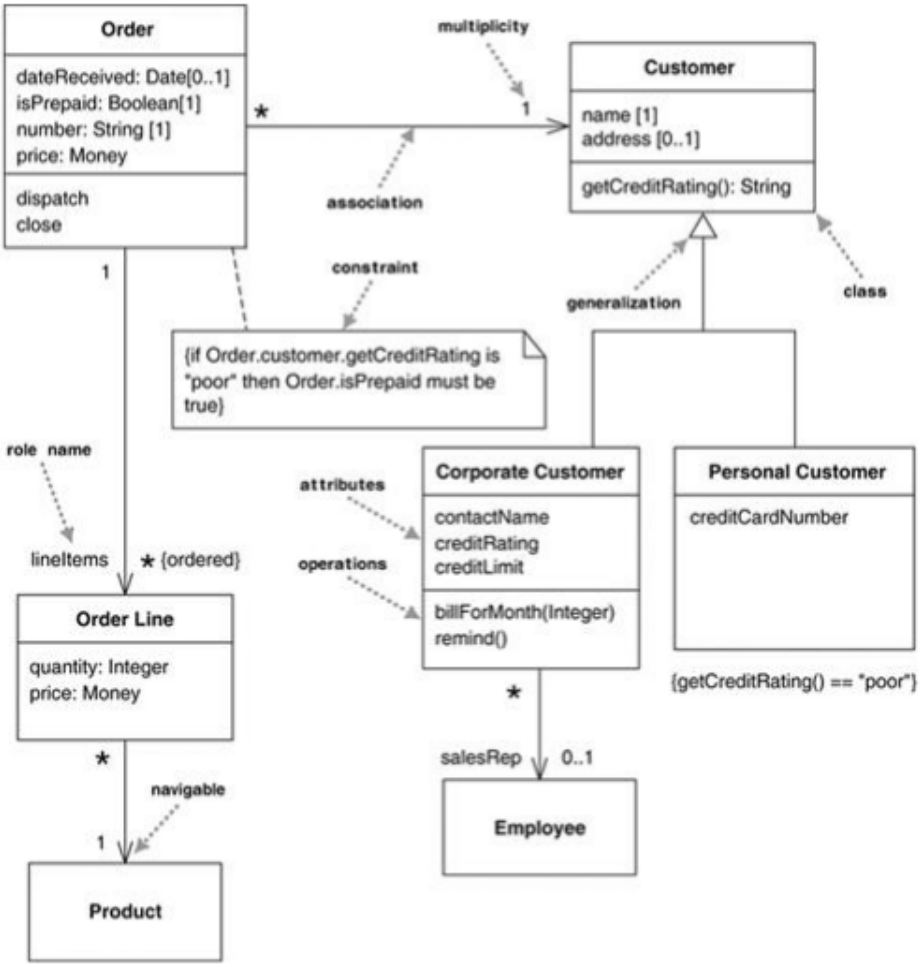


Figure 8.8: Based on Fowler 41: Class diagram example. Sorry about the quality I'm working from home and only have access to the eBook version.

Representing Classes

The class is represented by a box subdivided into three sections, the top section contains the class name, the middle section its attributes or more broadly the properties of the class while the lower section is reserved for the

behaviours or operations of the class. The syntax of the diagrams changes depended how they are being used (see 8.5.3). An example of conceptual classes (as discussed in the sections above is shown in 8.9.

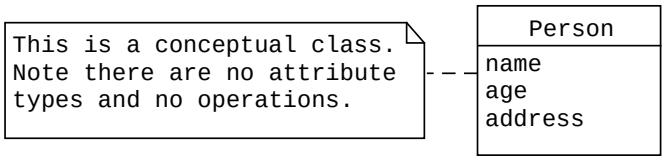


Figure 8.9: A Conceptual Class, of the kind that would be used to represent 'ideas' in the problem domain. This is the first of my own figures, not the best example to be honest. I need to come up with something better.

The UML notation provides the facility to include much more detail in a class diagram, the design and implementation diagrams use more of the syntax see 8.10 and 8.11. Fowler 41 provides a detailed discussion of these, there are also many other sources. The design class shown in 8.10 shows as much syntax as we'll need for now.

The Implementation class shows a much more information, including the get and set methods, constructors etc. which were omitted in the Design class. These are implicit (the variables are private and will need to be accessed,) but are only shown if there is something interesting about the get/set. This very verbose format would only be used rare circumstances in my opinion. I've used this format when specifying classes for students to implement and this format may be generated from code by a tool.

Some of you may by now have spotted the issue with our example, which I have allowed to cascade though the diagrams.

Representing Associations

Associations are relationships between instances of classes indicating that some meaningful and interested connection exists between them [56]. We'll consider associations and attributes in more detail next week, however as object diagrams without them are often incorrect we need to know a bit about them up front.
According to Larman 56 associations should be shown when one object needs to show that knowledge of a relationship needs to be preserved for some length

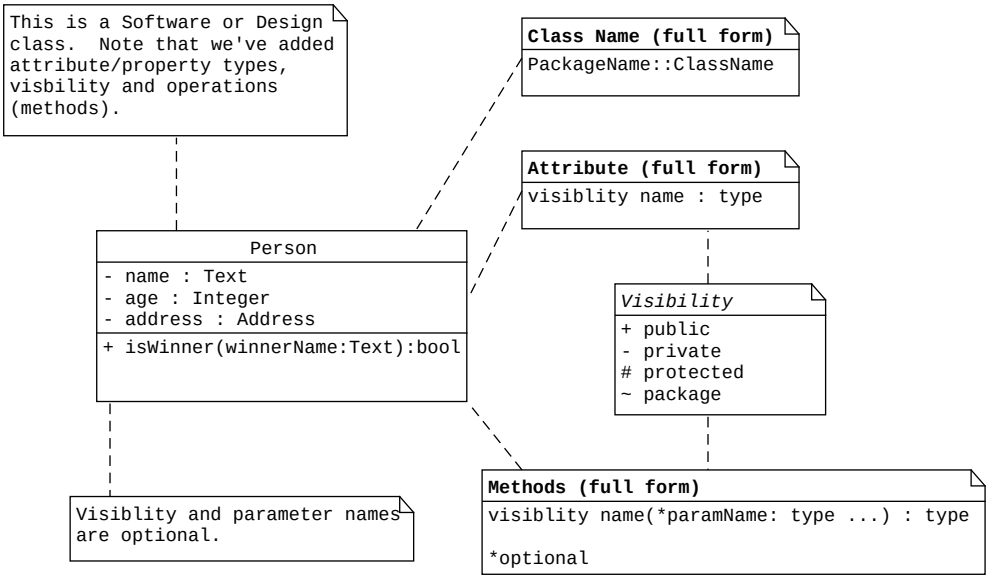


Figure 8.10: A Design Class, this is probably the more common form of class diagram and uses more of the defined 'language'. Again the example is not the most representative.

of time (milliseconds or years). At this stage (object modelling,) these will not always translate into relationships between classes in software [56]. Larman 56 suggests the following guidelines for the domain model, show only associations for which knowledge of the relationship is preserved for some duration and those derived from the Common Associations List.
We're not going to talk about this list in any detail until next week.
Representing associations is reasonably simple, we use a line with an association name [56] (see 8.12). At the end of each line we have a number or range representing multiplicity (number of) for instances of classes [56]. The association is inherently bidirectional and can be logically traversed in either direction, the association may have a reading direction arrow which suggests how the association name should be read - this has no meaning in the model just helps the reader [56].

In later sections we'll consider these relations in terms of design and include restrictions on navigation, multiplicity, names for association end points (roles) within the classes. However these are all for design models rather than domain models!

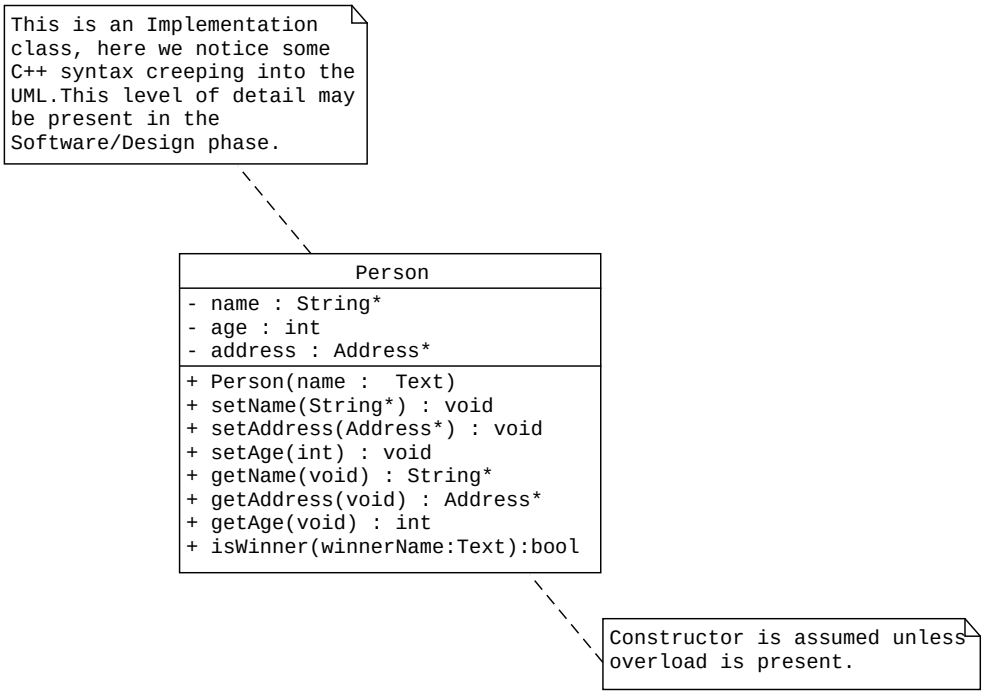


Figure 8.11: An Implementation Class, this is the kind of diagram you'd see developed for implementation by a junior/student or as an output from too which reverse engineered code into a class diagram.

We can however correct our earlier diagrams our conceptual class 8.9 had a concept as an attribute (see 8.7.2). I've fixed this in 8.13. This had not been spotted at the point we moved to the design class and is present here also, this probably wouldn't happen in the real world. There are instances however where you suddenly realise that meta-data has crept into a class, i.e something you though was an attribute has turned out to be a variable. For Example, you have a sensor which gets a single temperature reading but you later realise that it has to be calibrated and add a calibration constant. This should now be modelled as a separate class.

Classes and Class Diagrams in UP

UP defines several models which may use the class diagram, so far we've looked at the domain model [56]. Each model imposes its own terminology on the boxes (classes) of the model, for example in the Domain Diagram these

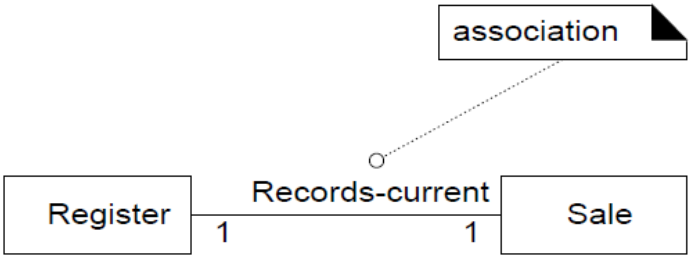


Figure 8.12: From Larman 56, a simple association with unit multiplicity.

represent domain concepts or conceptual classes [56]. To simplify things Larman 56 defines up front the terminology he's using (which is both UP and UML friendly,) and which we'll use in these notes.

Name	**UP Model**	**Perspective**	**Description**
Conceptual Class	Domain Model	Conceptual	A entity, concept or thing.
Software Class	NONE	Specification	A class representing the design or implementation of a given software component irrespective of language or method.
Design Class	Design Model	Specification or Implementation	A class in a design model, in UP this can be either a design model or an implementation model.
Implementation Class	NONE	Implementation	A class written in a given language such as Java or C++.
Class	NONE	NONE	as in the UML something which could represent any of the above depending on context.

8.7 Domain Modelling Guidelines

Larman 56 provides a number of useful pointers and additional guidance for creating domain models, a summary of which is provided here.

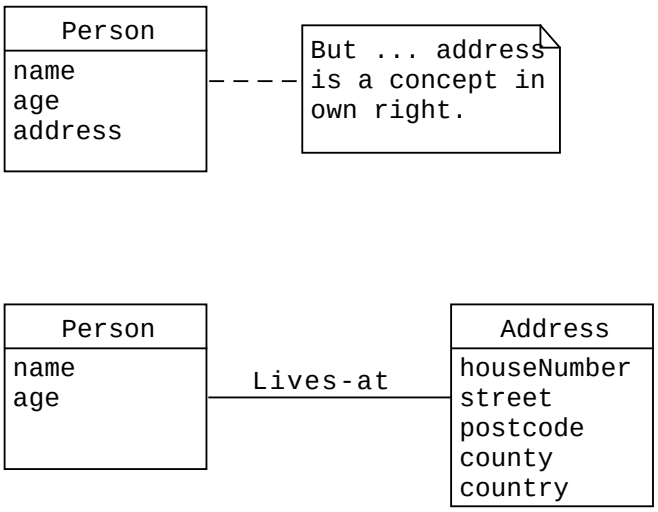


Figure 8.13: A Conceptual Class, of the kind that would be used to represent 'ideas' in the problem domain. This is a fixed version of 8.9 which separates out the x concept.

8.7.1 Think Like a Map Maker

When cartographers produce maps they use the local place names rather than creating their own, in a similar way when developing domain models we should use the terminology of the domain [56]. This applies to both objects and their attributes and possibly to the names of certain relationships [56]. This can be taken further, cartographers do not add cars, or people to their maps in the same way objects are discarded (or ignored) if they have no relevance in the problem domain [56].

8.7.2 Concept or Attribute

Larman 56 suggests that one of the most common mistakes when modelling is to misidentify a concept (object) as an attribute of another. For example, *Sale* may have an attribute called *store* to record the location from which an item was purchased. Larmans golden rule here is **If we do not think of attribute X as text or a number in the real world, then its probably a conceptual class**. In the example above *store* should be a conceptual class in its own right which has an association with sale. We'll come on to associations in a bit.

8.7.3 Description or Specification Conceptual Class

It is common to have entities in the real world which have associated information for example a description, this information should be associated with the item it describes rather than being a part of it [56]. The example shown in 8.14 shows Items, but this applies to a large number of products and services, e.g. phone contracts, flights, NPCs etc. Considering the "worse" example if we sell all of a given item, we delete the last object and loose all information about it? This cannot be right, we've sold out not eradicated them from existence. There are other concerns considering NPCs in a game, if each has a large mesh and texture associated with them that will be duplicated for each. However, this information is not specific to them it applies to all NPCs of that type.

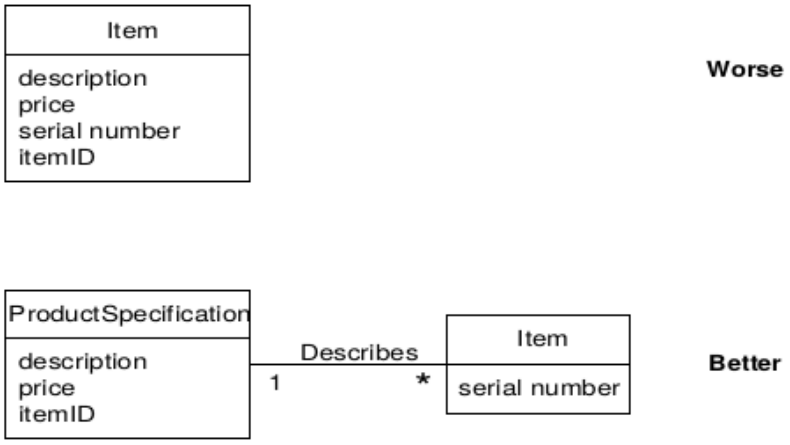


Figure 10.7 Specifications or descriptions about other things. The "*" means a multiplicity of "many." It indicates that one *ProductSpecification* may describe many (*) *Items*.

Figure 8.14: From Larman 56: Description as a separate conceptual class.

Aside: We'll have a very similar discussion at some point about object design when considering implementation and conclude that if an attribute has associated meta-date it should be made an associated class

Larman 56 suggests that a description/specification conceptual class is required when:

- There needs to be a description of an entity or service independent of the current existence of one of those services.

- Information could be lost when the last instance of an entity or a service is deleted due to being embedded in a conceptual class.
- Redundant information is stored in more than one conceptual class.

Slightly off topic here, but the last point on this list regarding duplication of information may sound familiar. A very similar concept exists in databases when we consider normalisation however the association between conceptual objects is a little more sophisticated than a foreign key.

8.8 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 8.8.1, then in teams of two produce:
    - An object model.
    - Based on the Wiki we've started and the use case we've listed so far what are the most critical?

Questions to think about (or what kind of exam question could I get on this).

- Describe the major artefacts of the Design discipline in up your answer should indicate how these relate to other disciplines such as domain modelling.
- Class diagrams can be interpreted from either a conceptual or essential perspective, from a specification perspective or from an implementation perspective. Provide an example of each of these and describe how the 'Raw UML' is interpreted in each instance.

8.8.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the

application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

8.9 Summary

In summary the process of developing an object orientated system requires that we model objects within the problem space. Using the same names for

classes as conceptual entities helps us lower the representational barrier aiding our understanding of the software.

There are a large number of suggested techniques, however experience is the key to identifying which of these are strong candidates for objects in our domain.

Next week we'll continue to develop the object model looking at the properties of the classes (attributes and associations).

8.10 Preparation for Next Week

8.10.1 Essential Reading

Read sections 9.1 through 9.8 you can skip the ones on finding finding associations/attributes we'll cover this in class.
Larman 56 Provides an overview of what we've done so far and introduces the next topic.

Larman (2014h). *Applying UML and Patterns*, chapter Domain Models, pages 131–172. Prentice Hall PTR

NOTE: Those of you who have the older version of Larman (before 3rd edition,) will find the order of the chapters has changed. The large 9th chapter is split over three smaller chapters in this book and proceeded by an unrelated chapter on System Sequence Diagrams (which we'll come back to later). So if you have that version, just read Chapter 10.

We're hopefully going to finish object models, next week and then move on to System Sequence Diagrams.

Larman (2014i). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 173–181. Prentice Hall PTR.

8.10.2 Background Reading

The following covers the development of object models really well:

Douglass, B. P. (2004d). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Anlysis of Real-time Systems, pages 331–399. Object technology. Addison Wesley, 3rd edition.

The same book has some really good sections on use case modelling, I'm slightly nervous about recommending it as I know how easy it is to get a copy.
YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Object Model (Part 2)

9.1 Introduction

In the last lesson we started the elaboration phase and the first artefact of the domain modelling discipline. The Elaboration phase this aims to refines the vision and over a number of iterations produces the core architecture. High risk elements will be resolved, the majority of the requirements identified (but not analysed in detail,) and the the scope of the project clearly defined. More realistic estimates are produced based on a better understanding of the project and developer experience.

9.2 Elaboration

Elaboration is a initial series of iterations over which we undertake three key activities [55]:

- **Development** – We develop the core risky architecture and test it.
- **Requirements** – Discover and stabilise the majority of requirements of the system.
- **Mitigate Risk** – We attempt to mitigate risks (reduce their potential frequency or severity using preventative measures,) or retire them (find that they are no longer a risk).

During elaboration the team undertakes the serious investigation, implementation (programming and testing,) of the core architecture, clarifies requirements and tackles high risk elements [55]. Risk in this context may not related to technical risk (i.e. things that might be impossible given the limits of technology or our knowledge,) but refer to major risks to project

success, that is features which if they are not correct will render it useless [55].

Elaboration proceeds over a number of iterations, usually around 4-6 weeks (the shorter the better,) each time-boxed (with a fixed end date,) and focusing on important use-case or a scenario (or part scenario,) from a use-case [55]. Use-cases may or may not not be implemented in a single iteration, we may aim to complete only part of it.

The code artefact from this stage should be production quality (not a throw away prototype,) though some authors use the term "architectural prototoype" to describe the early iterations of the core arhictecture [55]. Larman 55 suggests that a better term is **executable architecture** or **architectural baseline**. He goes on to suggest that in order to ensure that this baseline is stable particular use-case are chose for early implementation, i.e. a volatile architecture is a key risk we seek to minimise [55].

9.3 Domain Models - Recap

The Domain Model is used to show noteworthy concepts, bounded by the use-case (scenario etc.,) of the current iteration [56]. Here we're looking to identify things (though not always physical things,) we're interested in modelling. For example in a hotel we might include in our model, Room, Guest and Reservation only two of which are physical objects in our problem space.

The Domain Model will probably contain many of the items which are included in the Glossary, however it allows us to visualise and relate concepts in the domain while abstracting key details [56]. The UML class diagram notation, or rather a subset is used in the Domain Model which provides a conceptual perspective and shows [56]:

- domain objects or conceptual classes
- associations between conceptual classes, e.g. Sale relates to a number of items.
- attributes of conceptual classes e.g. each Sale has a Date and Time.

The Object Domain Model is a key artefact, object orientated development relies on the identification and modelling of entities, things (objects,) in the

domain which though their interaction (working together,) perform the required functionality of the system [56]. This has parallels with other approaches in terms of problem decomposition (divide and conquer,) we are breaking the problem into smaller parts which we can consider tackle in isolation.

9.4 Orientation - from last week

Last week we started looking at object domain diagrams, but only got as far as identifying the objects 8.5. This week we're moving on to consider the various methods to identify their associations and attributes.

9.5 Associations

It is useful to show the associations required for this iteration, to satisfy the information requirements of the current scenarios (use case) and which aid understanding of the domain [56].

An association is defined as the "semantic relationship between two or more classifiers that involve connections amongst their instances" [56] in this instance the class is our classifier so we're talking about relations between individual objects. Hopefully we covered this last year in the programming module and you've all implemented associations, you may have already associated one class with another in modules running alongside this one this year. The example shown in 9.1 provides some of the syntax, this is not all the Raw UML associated with Association (I'll cover this when we start looking at design diagrams).

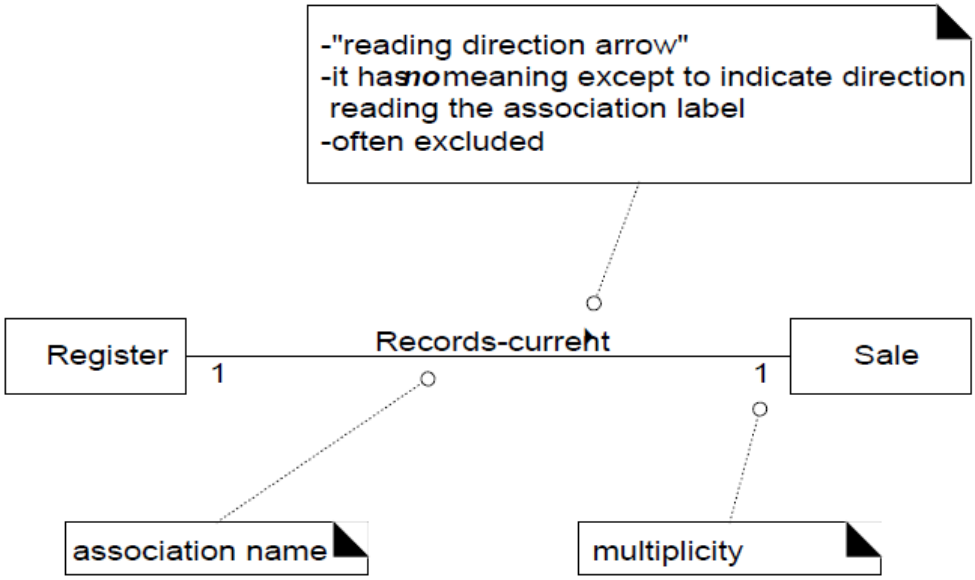


Figure 9.1: An example assocation from Larman 56

What Associations should be included?

Larman 56 states that only those relations which are preserved for some duration. This may be anything from milliseconds to years depending on the context.

Deciding exactly which ones we show and which are visual noise, is difficult and Larman 56 suggests:

- Associations for which knowledge of the relationship needs to be preserved for some duration (i.e. "need-to-know" associations).
- Associations derived from the Common Association List.

Syntax: Associations between Conceptual Classes

While we need some of the class diagram syntax for association we don't need all of it, I'm going to leave some of it until we actually need it (we'll focus on the raw UML we need for object domain diagrams for now). The diagram 9.2 shows a connection between two objects, this has a directed association name (see 9.5) and multiplicity at each end (see 9.5).

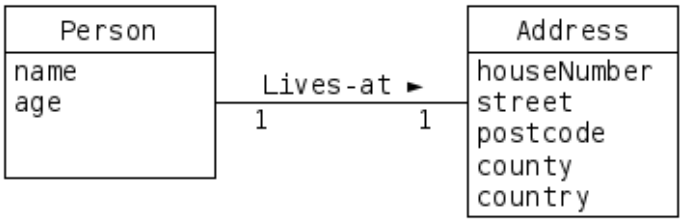


Figure 9.2: A home made example based on those in Larman 56, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image. I've had to screen shot this one to get it to work.

Naming Associations

Simple names such as "has" and "uses" are a poor choice as they do not enhance our understanding of the domain [56]. In the example 9.2 "has" could have been chosen at as the association name but would have added nothing to our understanding, however "Lives-at" tells us its their home address. Larman 56 suggests using a *ClassName-**VerbPhrase**-ClassName* format which is both informative and can be read. While his examples use a hyphenated format, camel case is also acceptable.

Roles & Multiplicity

The end of each association has a role and can have a name, multiplicity expression and navigability [56]. We're going to look at most of these later in the course when we look at design diagrams, however multiplicity is useful now.

Multiplicity :

Multiplicity simply defines how many of each class are involved in an association at one time [56]. That is how many instances can be connected, we can improve / extend our earlier example (person-address) by considering that more than one person can live at the same address. It may be that a person lives at a number of addresses over the course of their lives, but for this system they have only one current address. The examples shown in 9.3 are reasonably representative.

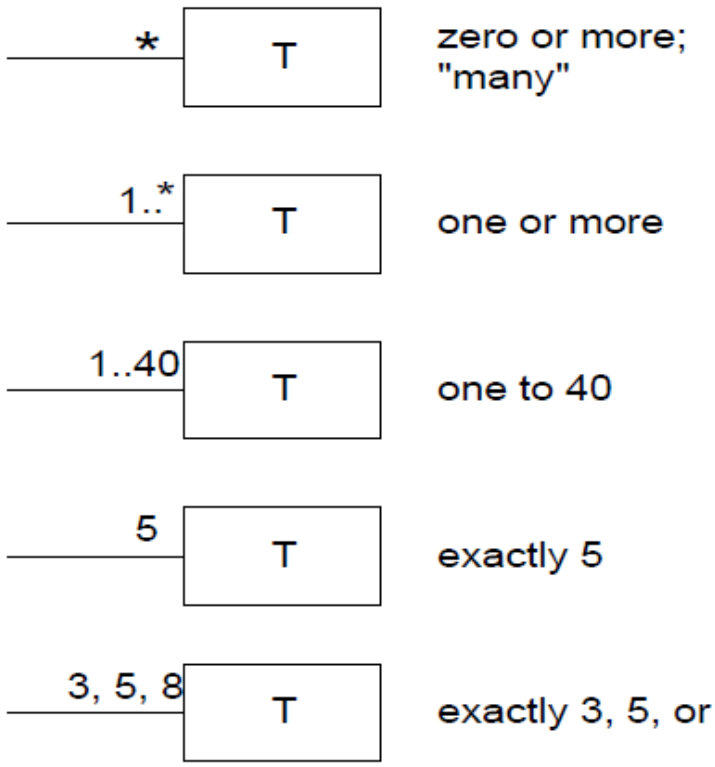


Figure 9.3: A home made example based on those in Larman 56, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image. I've had to screen shot this one to get it to work.

Larman 56 notes that there is some disconnect between the use of associations in domain diagrams and in design diagrams here. While in the real world we may not record the absence of something we may wish to do so in the software case to ensure we know that it should be there and its absence is an error.

The multiplicity of a relationship is always based on the needs of the scenario, [56] cites Rumbaugh's example where a Person is associated with a Company. For tax purposes this will probably be a list of companies (i.e. all those who the person currently works for) for a Union probably only one.

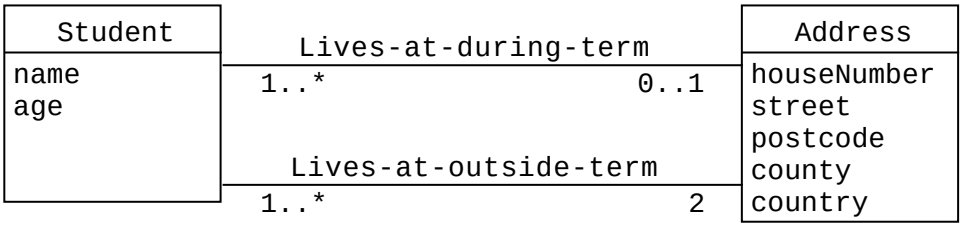


Figure 9.4: A home made example based on those in Larman 56, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image, I've not screen-shot this and its lost its arrows.

A final note is that we can have multiple many to many relations [56], this is perhaps best illustrated though an example see 9.4. Here each student has two addresses a term time address (which may or may not be present,) and a home address. Both of these are address objects, note that one or more students can live at the address in each case.

9.5.1 Identifying Associations

Larman 56 recommends a category list to identify potential associations this is shown in 9.5.

Category	Examples
A is a transaction related to another transaction B	*CashPayment—Sale* *Cancellation—Reservation*
A is a line item of a transaction B	*SalesLineItem—Sale*
A is a product or service for a transaction (or line item) B	*Item—SalesLineItem (or Sale)* *Flight—Reservation*
A is a role related to a transaction B	*Customer—Payment* *Passenger—Ticket*
A is a physical or logical part of B	*Drawer—Register* *Square—Board* *Seat—Airplane*

Category	Examples
A is physically or logically contained in/on B	*Register—Store, Item—Shelf* *Square—Board* *Passenger—Airplane*
A is a description for B	*ProductDescription—Item* *FlightDescription—Flight*
A is known/logged/recorded/reported/ captured in B	*Sale—Register* *Piece—Square* *Reservation—FlightManifest*
A is a member of B	*Cashier—Store* *Player—MonopolyGame* *Pilot—Airline*
A is an organizational subunit of B	*Department—Store* *Maintenance—Airline*
A uses or manages or owns B	*Cashier—Register* *Player—Piece* *Pilot—Airplane*
A is next to B	*SalesLineItem—SalesLineItem* *Square—Square* *City—City*

Figure 9.5: A table from Larman 56, this shows the common associations list.

For those who need further explanation/examples please see:

Larman (2014h). *Applying UML and Patterns*, chapter Domain Models, pages 131–172. Prentice Hall PTR.

9.6 Attributes

It is useful to identify attributes of conceptual classes where these are required to satisfy the information requirements of the current scenario [56]. Attributes are defined as "a logical data value of an object" [56].

9.6.1 What attributes should be shown?

Larman 56 suggests that we should only show those attributes which are suggested by the requirements artefacts (use-cases scenarios etc.) i.e. where we need to remember information.

Looking back at our previous examples our selected examples were out of context and the need for the selected variables gets lost. However, if we consider a Sale being stored in a till system we'll need to store the data and time of that sale. If we're storing a student as part of a Student Information System (at UWTSD) we'd need to store the P number along with other details such as full name, title etc.

Attribute Notation

Attributes are shown the in second compartment (from the top) (see diagram 9.6) and may have a type and other information (see 9.7. The full notation is given in 9.6 we note that we can include:

- Derived attributes which are preceded by a /(perhaps the best example being age, which can be calculated based on the data of birth)
- Visibility - which uses the same syntax as methods (i.e. $+, -, \#, \&$).
- Property String - this describes the properties of the data e.g. {Read-Only}.
- Optional values - indicates weather a given value is required 0 or not 1.

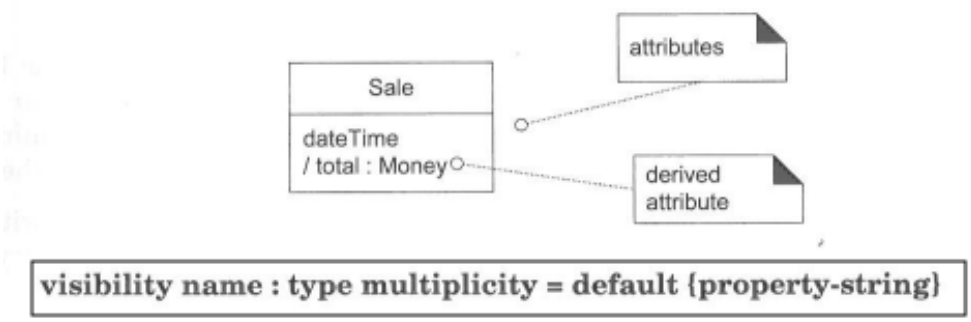


Figure 9.6: An example from Larman 56, this shows some basic syntax (as might be used in an domain object model. The full syntax (raw UML) is given for attributes in the box.

Derived Attributes - Are interesting we use them to communicate that while this is a noteworthy attribute, it is derivable from other data and therefore does not need to be stored [56]. The UML convention is to use a / symbol before the attribute name [56]. Its also useful to note that there are a number of useful attributes which can be derived from the multiplicity of a relation - e.g. we could work out the number of students in a group from the multiplicity without storing its separately [56].

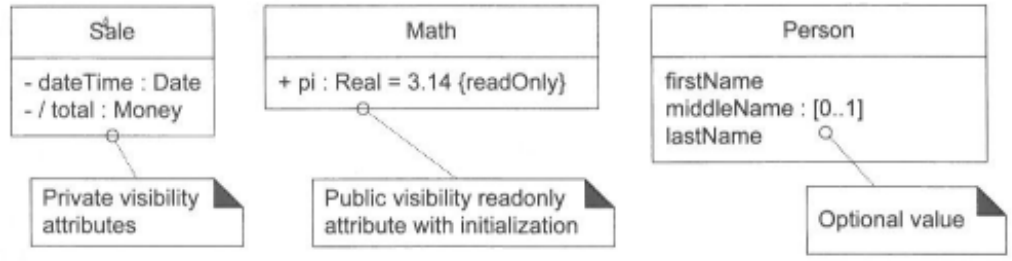


Figure 9.7: A second example from Larman 56, this shows how other information may be captured. For recording purposes Larman suggest including this in the glossary see 9.6.1

Where to Record Attribute Requirements?

Adding the additional information above, e.g. derived attributes, property strings and optional values embeds requirements or domain rules in the domain model [56]. Larman 56 suggests that while this can be useful when creating domain diagrams with domain experts (users etc.) the domain module is not always updated or used as a reference and these should therefore be transferred to the glossary.

What are Suitable Attribute Types?

Attributes should be "primitive types" that is something which is not a concept within our system (e.g. Number, Text, Boolean etc.) [56]. We avoid at this stage considering primitive types from the design/implementation perspective, i.e. int, float etc. Some of these will require an object at design time to represent them (e.g. Text/String, DateTime etc.). While Larman 56 provides quite an exhaustive list, I suggest we restrict ourselves to:

- **Basic Data Types:** Boolean, Number, Character and String (or Text).
- **Common Complex Data Types:** Date, Time and Colour

- **Geometric Primitives:** Point, Rectangle.
- **Enumerations:** e.g. Size=small,regular,large.

Larman 56 goes on to discuss data types in some detail, he concludes that the term "primitive" implies a set of values for which a unique identity is not meaningful. For example, the Integer 5 cannot be distinguished from another Integer 5, nor a string containing "dog" from another string containing "dog". However there may be more than one instances of person, with the same name. In this instance would not compare two entities by instance i.e. x == 5, we would used x.equals(y) that is compare the values of the entity.

Guideline: Defining new Data Type Classes

There are a number of instances where we encounter this issue of non "primitive" data types, often this suggests a conceptual entity which we should consider. Some of these will be very small i.e. have no meta-data but have associated rules governing their use or correctness [56]. For example consider StudentID, we may at first suggest a number (i.e ignore the P in the case of UWTSD Swansea students,) or possibly text however there may be sub-parts within the I.d. e.g. the first year of enrolment may be the first two digits, the department the second two etc.

Larman 56 provides some some common cases for making an attribute a new data type with examples:

- Attribute composed of multiple sections
  - Phone Number
  - Address
  - Full Name
- Attribute has associated operations, such as parsing, validation etc.
  - Student Number
  - Car Registration
  - ISBN
- Attribute is has other attributes (meta-data)
  - Promotional Price - has a start and end date
- Attribute is a quantity with a unit, (see 9.6.1).

  - Price - has a unit of currency e.g. Pounds, Euros, Groats etc.
  - Weight - has a unit of measure e.g. grams, kilograms etc.

- Attribute is an abstraction of one or more types with some of these quantities.
  - Larmans sale item object has an identifier which is a generalisation of specific types Universal Product Code (UPC) and European Article Number (EAN).

Whether we include these as separate objects in our domain diagram or not depends on what we are looking to emphasise in our digram [56]. At this stage (i.e. while your a student trying to show you can model in objects,) I would suggest expressing all data-type objects as separate and making no assumptions.

Guideline: Avoid Relating Objects using Attributes

Attributes should not be used to relate two conceptual classes in the domain model [56]. This is often a case of database driven thinking where **foreign key** type attributes are included [56]. For example we may include a CourseID in a student object, while we may do this at a later stage to enable us to write a student to a database as a record. This is a design decision which should be avoided at this stage [56].

Guideline: Modelling Quantities and Units

Larman 56 suggests that the majority of numeric quantities should not be represented as plain numbers, he suggests that giving a price of 20 or a weight of 100 does not provide enough information. It is common in this case to represent such values with an associated Unit, describing the units used [56]. This may be used to format output and for validation purposes.

9.6.2 Identifying Attributes

Larman 56 provides no guidance for identifying attributes at this stage.

9.7 Is the domain diagram correct?

In short no. But we're not looking for correct, our aim is to create a useful diagram one which we can use as a tool for communication and understanding

[56]. A useful model captures key domain abstractions and the information required to understand the domain, i.e. its concepts, terminology and relationships [56].

9.7.1 The Role of Domain Models in UP

The Domain Model is exploratory and may evolve over a number of iterations during inception, however by the end of which it may well be completed [56]. As we are exploring the system one use-case or scenario at a time we'll expand the domain model for each as required, however by the end of elaboration there may be no 'new' concepts to uncover. Larman 56 stresses here that we should avoid trying expand the model to cover everything, i.e. stick to the scenario/use-case at hand, avoiding 'waterfall' (big model up front) thinking.

9.8 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 9.8.1, then in teams of two produce:
    - Refine the object model from last week, adding attributes and associations as required.
    - Take the use-case 5.6 as our current iteration subject and create a complete object model.

Questions to think about (or what kind of exam question could I get on this).

- Describe the major artefacts of the Design discipline in up your answer should indicate how these relate to other disciplines such as domain modelling.
- Class diagrams can be interpreted from either a conceptual or essential perspective, from a specification perspective or from an implementation perspective. Provide an example of each of these and describe how the 'Raw UML' is interpreted in each instance.

9.8.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application

must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

9.9 Summary

In summary the process of developing an object orientated system requires that we model objects within the problem space. Using the same names for classes as conceptual entities helps us lower the representational barrier aiding our understanding of the software. The domain model is an incomplete Business Object Model which focuses on a small part of the business/system concerned with the software project at hand [56].

This week we've looked at the inclusion of relations and attributes in our domain model to capture essential details. This provides us with a first holistic structural view of our system which will be refined in the design model towards something we can implement in software.

Next weeks we'll look at System Sequence diagrams, using these as both an introduction to sequence diagrams and first holistic view of the systems behaviour. Future classes will then move into looking at the design of the system and how this is translated to running code and tested.

9.10 Preparation for Next Week

9.10.1 Essential Reading

Larman 56 Provides an overview of what we've done so far and introduces the next topic.

NOTE: Those of you who have the older version of Larman (before 3rd edition,) will find the order of the chapters has changed. The large 9th chapter is split over three smaller chapters in this book and proceeded by an unrelated chapter on System Sequence Diagrams (which we'll come back to later).

For the class after Christmas read the chapter on System Sequence Diagrams (from the 3rd edition):
Larman (2014i). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 173–181. Prentice Hall PTR

9.10.2 Background Reading

The following covers the development of object models really well:

Douglass, B. P. (2004d). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Anlysis of Real-time Systems, pages 331–399. Object technology. Addison Wesley, 3rd edition.
The same book has some really good sections on use case modelling, I'm slightly nervous about recommending it as I know how easy it is to get a copy.
YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - System Sequence Diagrams

10.1 Introduction

In the last lesson considered the first artefact of the domain modelling discipline, the domain object model as part of the elaboration phase. The Elaboration phase this aims to refines the vision and over a number of iterations produces the core architecture. Along the way high risk elements will be resolved, the majority of the requirements identified (and the most important analysed in detail.) We move now to the System Sequence Diagrams these are high level behavioural diagrams which show the system interacting with various actors in various scenarios. This is a continuation of the analysis, we're essentially adding information (or detailing,) the use case by visualising the systems communications with external entities. This feeds directly into our design, these input/output events will be passed from an interface layer (e.g. the UI,) to our application logic.

10.2 Elaboration - Recap

Elaboration is a initial series of iterations over which we undertake three key activities [55]:

- **Development** – We develop the core risky architecture and test it.
- **Requirements** – Discover and stabilise the majority of requirements of the system.
- **Mitigate Risk** – We attempt to mitigate risks (reduce their potential frequency or severity using preventative measures,) or retire them (find that they are no longer a risk).

During elaboration the team undertakes the serious investigation, implementation (programming and testing,) of the core architecture, clarifies requirements and tackles high risk elements [55]. Risk in this context may not related to technical risk (i.e. things that might be impossible given the limits of technology or our knowledge,) but refer to major risks to project success, that is features which if they are not correct will render it useless [55].

Elaboration proceeds over a number of iterations, usually around 4-6 weeks (the shorter the better,) each time-boxed (with a fixed end date,) and focusing on important use-case or a scenario (or part scenario,) from a use-case [55]. Use-cases may or may not not be implemented in a single iteration, we may aim to complete only part of it.

The code artefact from this stage should be production quality (not a throw away prototype,) though some authors use the term "architectural prototoype" to describe the early iterations of the core arhictecture [55]. Larman 55 suggests that a better term is **executable architecture** or **architectural baseline**. He goes on to suggest that in order to ensure that this baseline is stable particular use-case are chose for early implementation, i.e. a volatile architecture is a key risk we seek to minimise [55].

10.3 System Sequence Diagrams

System Sequence Diagrams (or SSD) are as their name suggests sequence diagrams for the system as a whole. They are a quickly created artefact aimed at capturing the inputs and outputs to the system (or subsystem,) under discussion [57]. They provide input to several other elements of design, key to us will be their role in object design. The UML notation provides no explicit syntax for System Sequence Diagrams, as with class diagrams and the Object Domain model we use raw UML (in this case the syntax for sequence diagrams).

Use cases describe interactions between external actors and the software system we're developing, actors generate system events to the system usually requesting some system operation to handle the event [57]. Such operations are implied from the use case text, in Larman 57 the example is "3. Cashier **enters item identifier** and ..." this implies the *enterItem* event the SSD makes this concrete and explicit [57]. We may note here that Larman is careful to avoid specifying how the item identifier is entered (i.e. could be a

bar-code reader, RFID, keyboard etc.).

The SSD is then a diagram which shows for a particular scenario of a use case the events generated by an external actor, their order and and inter-system events, each system is treated a s a black box with the emphasis being on the events which cross the boundary between system and actor [57]. Larman 57 suggests that SSDs are drawn for the success scenarios of each use case and any frequently occurring or complex alternatives. Note here that the syntax of the sequence diagram is complex enough to express ALL scenarios, but we're aiming for clarity and accuracy over completeness.

10.3.1 Key Motivation for SSD

Larman 57 note that the events coming into a system are often key to its design as the system must be designed to handle these and execute a response. Software systems typically react to three kinds of event (from [57] augmented with my own examples):

1. **External events** - Those generated by actors (humans, computers, sensors etc.).

2. **Timer events** - Usually some kind of event or interrupt generated by software or hardware. Examples include the kind of timers found in games for updating graphics/physics.

3. **Faults or exceptions** - These are often external to the software and generated as a result of a software interrupt from the operating system (or hardware interrupt from the hardware).

Knowing exactly the input events (the system events,) is a vital step in the analysis of the system [57]. At a lower level (later in the course,) we'll use the same sequence diagrams to consider the messages being passed between objects in a scenario. These often amount to services being requested from one object by another, however here we consider this at a much higher level (the system as a whole,) [57]. The SSD then forms part of the picture with the use case, the SSD details what goes into the system and comes out (with the system as a black box,) while the use case similarly defines with the system does (but not how, only constraints on how) [57].

10.4 Sequence Diagram Syntax - Part 1

As with class diagrams the plan is to front load a with the syntax of sequence diagrams a little more than Larman does in his book. My reasoning for this is twofold, there is a lot to sequence diagrams and I've found in the past students take a while to warm up to them. Second, I'd like to get this done ahead of starting the design of the system, so we don't have a huge digression into the syntax of sequence diagrams when we're getting stuck into the design.

Before we begin we first need to be aware that the sequence diagram can be used at different levels (as we've seen with the class diagram 8.5.3. Both Larman 56 and Fowler 40 describe UML being used at different levels, this relates back to our assertion that UML is a language used for a number of purposes. Larman 56 identifies:

- **Essential or Conceptual Perspective** – diagrams are interpreted as representing real world entities and their relations in the problem domain (this is where our System Sequence Diagram fits).
- **Specification Perspective** – diagrams are interpreted as representing software elements, in this case objects/classes communicating but with no reference to the features of the implementation language. We note that the notation used is the same as that for the conceptual perspective, though clearly extended a little.
- **Implementation Perspective** – diagrams are interpreted as representing software elements in a particular language i.e. including the specifics (method names, variable types etc.) of the language chosen.

10.4.1 Participants and Messages

Sequence diagrams are one of two kinds of interaction diagram provided by UML (the other being communication diagrams,) [58], which we're going to leave to a future iteration. Sequence diagrams are more commonly used than interaction diagrams as they emphasise sequence over structure, making it easy to find the next message (with time going down the page) [30]. A communication diagram by contrast is simply an object diagram (like a class diagram but showing individual objects,) which shows the messages passed during an interaction [30].

There is also the recently added Interaction Overview Diagram which shows how these are connected, this according to Larman 58 is relatively new and

yet to prove its worth. Sequence diagrams are the more notationally rich [58] and hence our first choice, 10.1 is based on Larman 58 but with some additional annotations of my own.

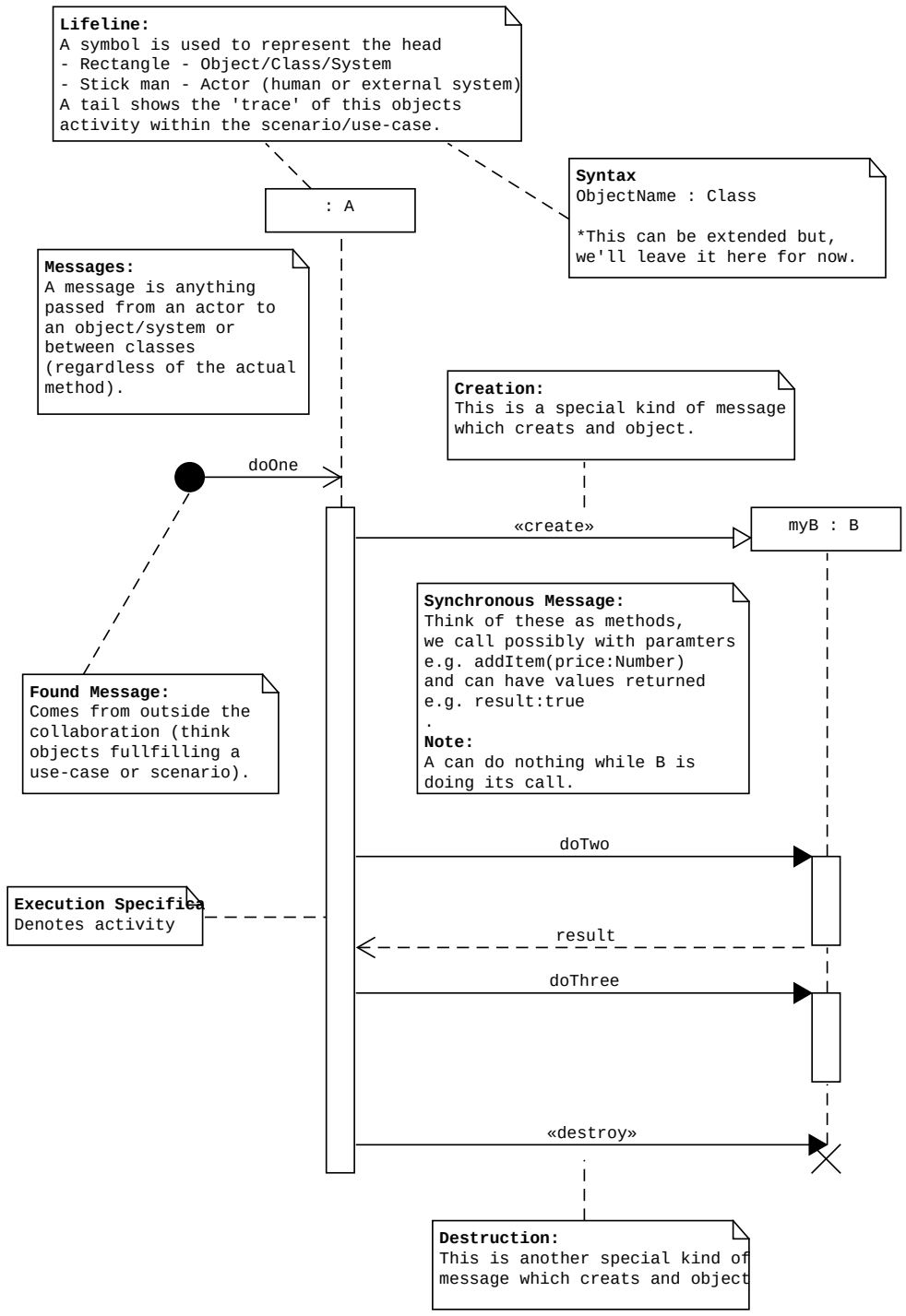


Figure 10.1: Based on Larman 57, augmented with my own annotations inspired by Fowler 42 and Douglass 30

The key elements (in 10.1) shown are:

- **Participant** - This is shown in the rectangle at the top of the lifeline, it has usually has the format **objectName:ClassName** however this can be extended to it can be used to describe multiple objects etc. objectName[n]:ClassName. The UML specification [88] provides more detail (though in a less than readable format). Common participants are show in 10.2.

- **Synchronous Message** - The sequence diagram syntax allows us to show both asynchronous and synchronous messages. The diagram above shows the latter, the best way to think of these is as one object calling a method on another, perhaps with arguments and returns (see 10.4.1 for more examples and discussion.

- **Create and Destroy Messages** These are special kinds of messages which have a stereotype associated with them i.e. «create» and «destroy». These can be though of as calling a notional constructor and destructor for the object. These are not normally shown in System Sequence diagrams.

- **Found & Lost Messages** - These are messages which leave the boundary of the current sequence diagram. Sequence diagrams are normally used to describe an interaction between classes (this could be a scenario, single use case, part of a use-case etc.) where messages enter or leave context they are represented as lost and found messages.

- **Execution Specification** - Processing within a method is often shown as a block on the lifeline, these are often used to reinforce the difference between synchronous and asynchronous messages. These are more commonly seen in diagrams created with UML tools and known as the Activation Bar in UML 1.0 [58].

NOTE: The syntax of sequence diagrams has changed over the various versions of UML, there are also a number of tools, books, authors who appear to impose their own limitations or add their own flourishes. Please make every effort to use the correct syntax, if your in any doubt check the notes. If you're writing your diagrams up in a tool and it is unable to generate the correct syntax, put a note to explain the issue. As with the other diagrams I'm not looking for perfect syntax (the logic is more important,) however there is a point at which it becomes just lines and boxes and is no longer UML.

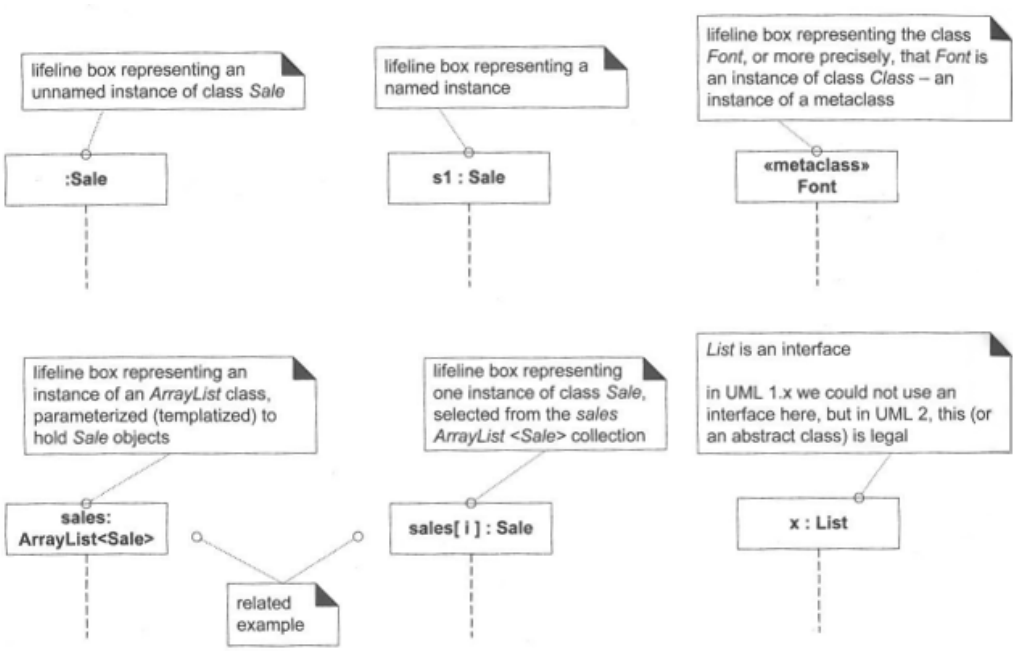


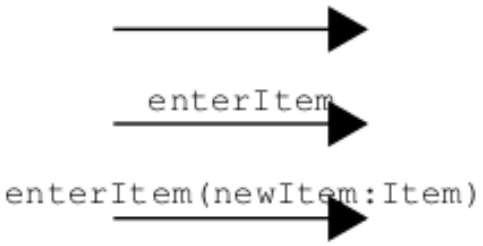
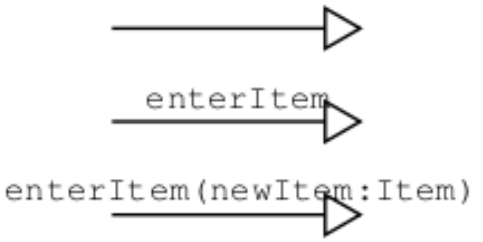
Figure 15.5 Lifeline boxes to show participants in interactions.

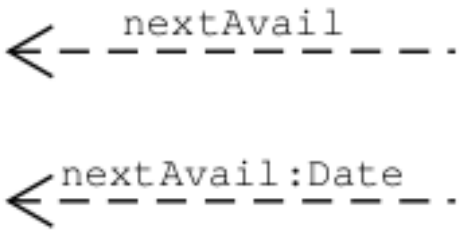
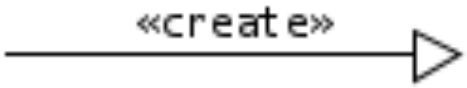
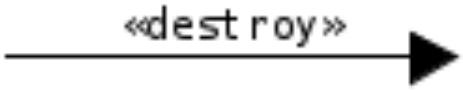
Figure 10.2: From Larman 58.

Larman 58 provides some excellent documentation on the Sequence diagram including the flowing examples of the kind of classes which may be involved at the design phase.

Messages

As mentioned above there are a number of different kinds of message which can pass between instances (or classes) in a sequence diagram, these are:

Type	Description	Symbol
Asynchronous	A message is send but the sender does not wait for a response, e.g. an event is sent to an event queue.	
Synchronous	A message is send and the sender waits until the processing is complete in the receiver before continuing. There may be a reply message involved 12.8.4	

Replies	These are messages returned as the result of a call on another object, you can think of them as method returns. There is an alternative syntax (Larman 58 provides an example, we'll stick with one way to avoid confusion.	
«create»	A special kind of synchronous message resulting in the creation of an object.	
«destroy»	A special kind of synchronous message resulting in the destruction of an object.	

Asynchronous communication is most common with real-time systems where objects have their own threads of execution, e.g. in multi-process or multi-threaded design where there is more than one active object. They are also found in System Sequence diagrams as actors as external entities have a thread of execution separate from the system.

All messages have the full form

return = messageName(paramter : paramterType) : return type

however it is not uncommon to see, calls for specific programming languages used here [58] this is common where the diagram has been generated from code, or is being used to generate code Model Driven Development (MDD). Larman 58 also suggests that empty parameter lists and void returns are often omitted, type information may also be omitted if obvious [58]. I'm not sure the latter is a good idea!

Partial Ordering

The sequence diagram shows messages being sent and received in order for each object, however between objects there is no timing implied [30] see 10.3.

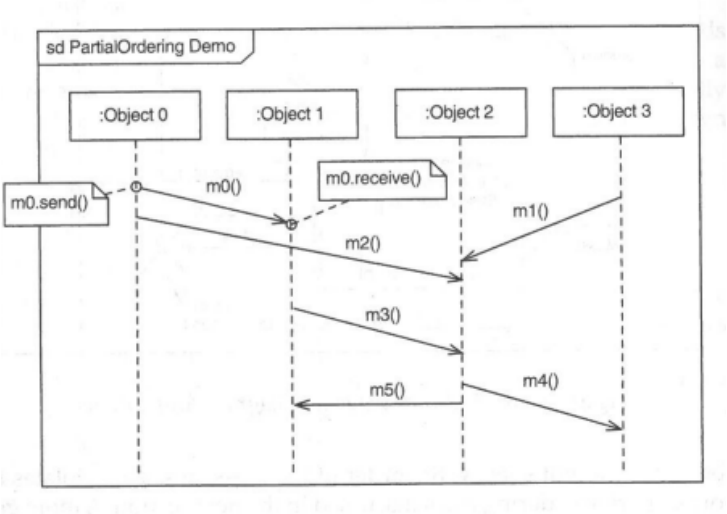


Figure 10.3: This is from Douglass 30, it shows messages being sent between objects (asynchronously,) while we can see that m0.receive() takes place before m3.sent() we can make no assumptions bout m0.send() and m1.send() or even m0.receive().

10.4.2 Frame Operators

As part of this introduction to sequence diagrams I'm including some of the Frame Operators I think we'll use at this stage. I've named the others in the table, but this is more to remind me that I need to include them when we consider sequence diagrams again. The following table is based on Larman

58 and Douglass 30.

Operator	Description
sd	Names an interaction fragment. This could be a system sequence diagram, a realisation of a use case (i.e. how objects from our class diagram will interact to get the job done,) a part of a sequence diagram we want to re-use in other diagrams. **Not covered here in detail, we don't need them until later see ??**.
ref	References another interaction fragment, i.e. where we have parts we wish to re-use (e.g. authenticateUser) we can refer to them with this. **Not covered here in detail, we don't need them until later see ??**.
alt	This provides a number of alternatives, one of which will be taken based on the guard conditions (in [ ]) an else guard is also provided which evaluates to TRUE if all others are false. This is a complex fragment [30] and is used in UML to express both if/else and switch/case type programming structures. **Note:** Larman 58 suggests the guard should be placed over the life-line of the participant who performs the test. See figure 10.4.
opt	A kind of simplified alt, a fragment which will only be executed if its guard is true. See figure 10.6.
loop	Specifies that a fragment should be repeated until some condition (expressed in a guard) becomes FALSE. There is no official **for** syntax though it is suggested one is being considered [58]. See figure 10.5.
par	This is similar in syntax to alt however, here all compartments execute rather than one. **Not covered here in detail, we don't need them until later see ??**.
region	Identifies a region which is atomic, and cannot be interleaved with others, i.e. the actions in this fragment must take place in this order with no possibility of interruption. **Not covered here in detail, we don't need them until later see ??**

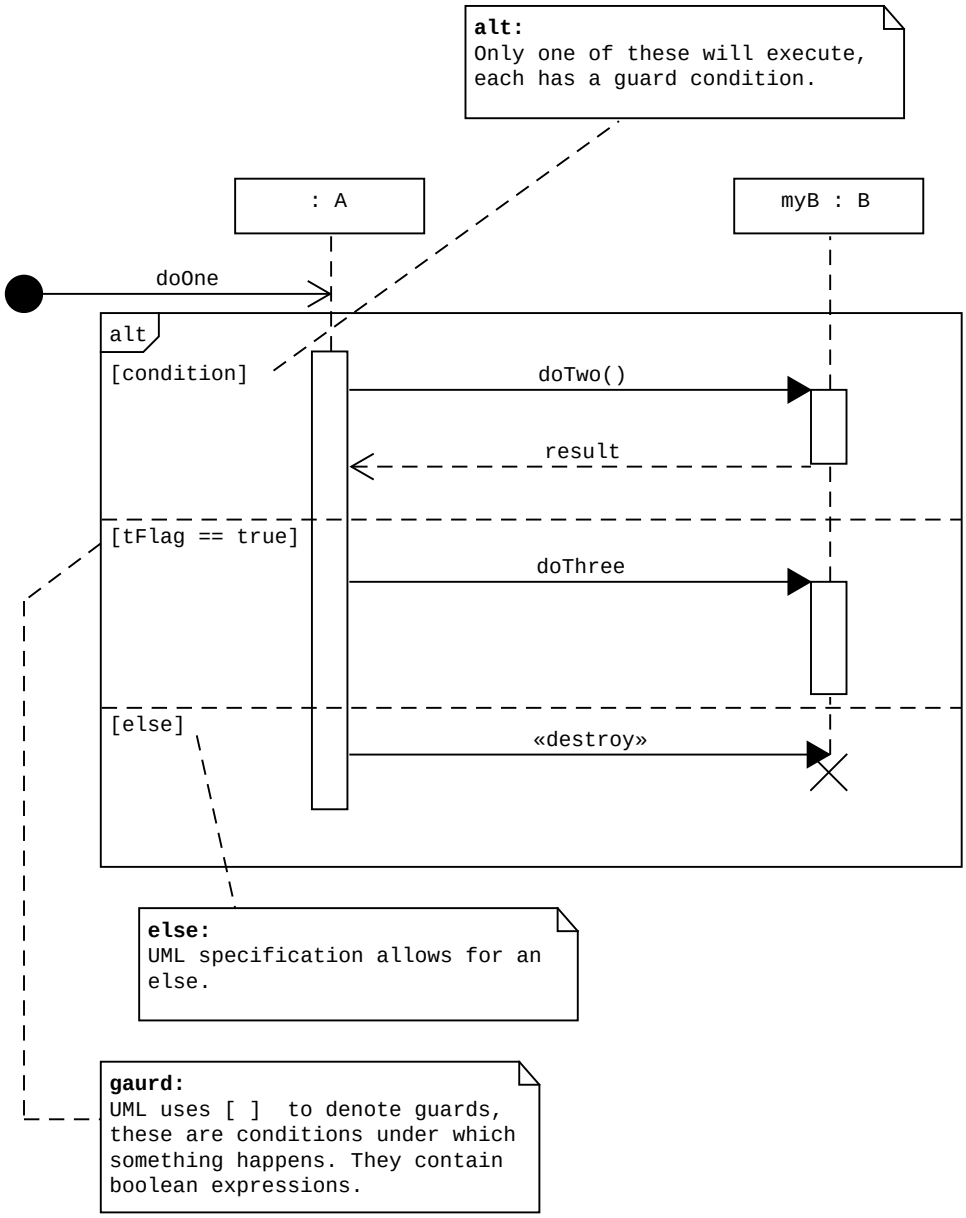


Figure 10.4: Syntax for the alt interaction fragment, this is used for both if/else and switch case type structures.

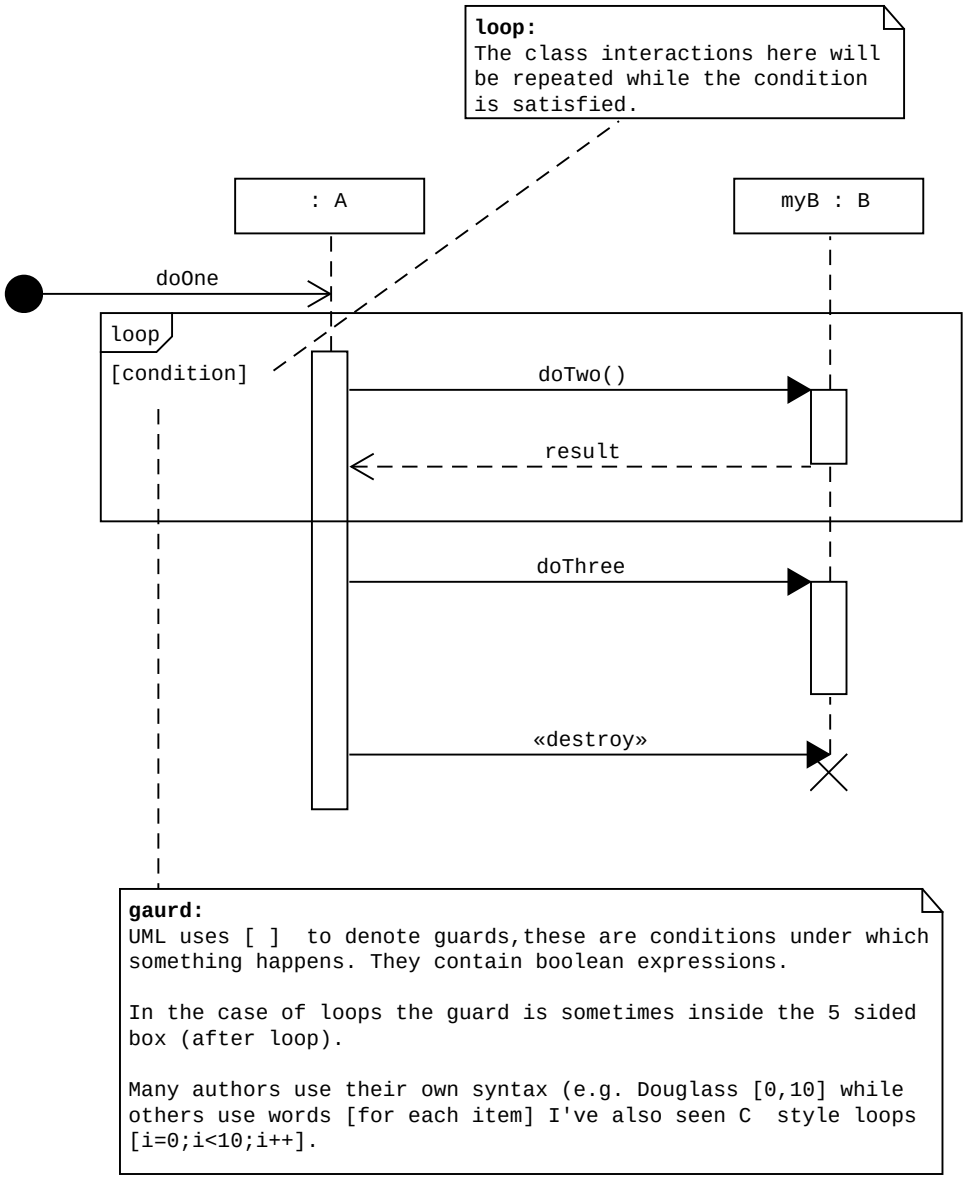


Figure 10.5: Syntax for the loop interaction fragment, this is used for both while and switch for structures.

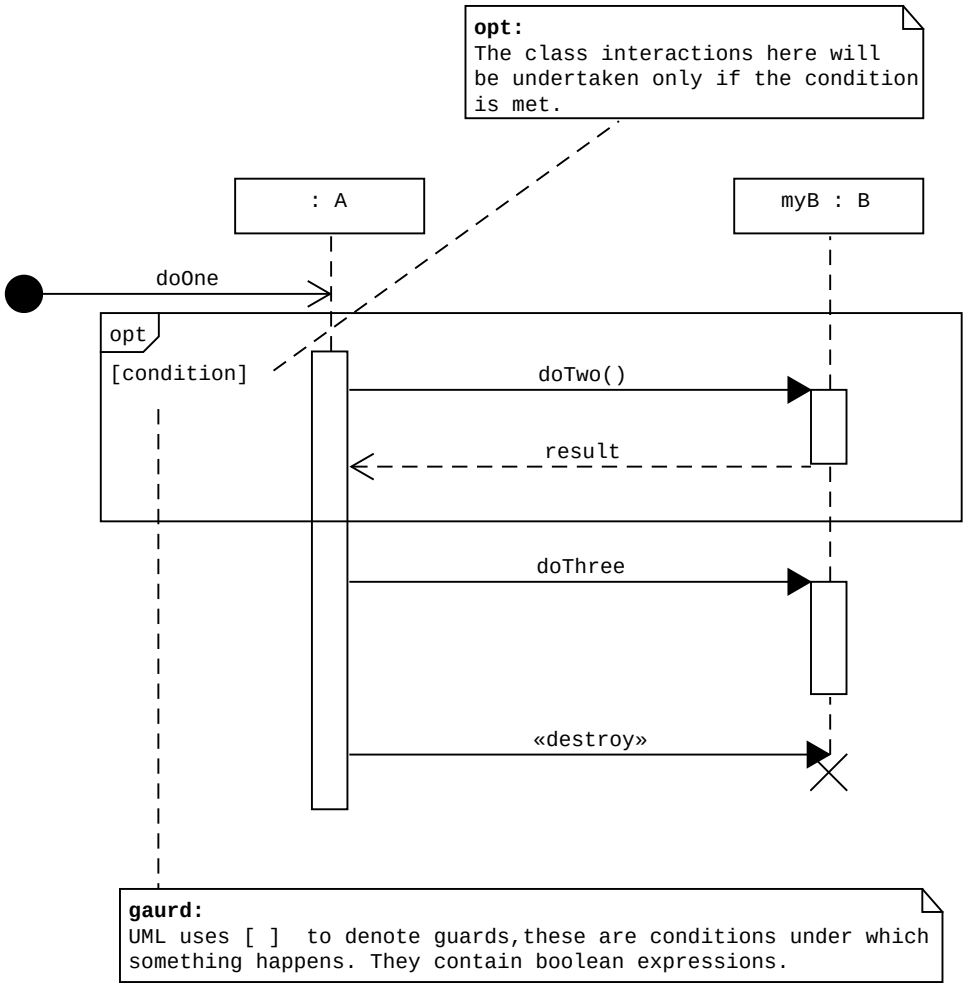


Figure 10.6: Syntax for the opt interaction fragment.

10.5 System Sequence Diagram Creation

The creation of a System Sequence Diagram (SSD) is fairly simple, we start with a use case and work though the interactions between system and actors, time proceeds downwards in the diagram [57] however events between entities are not synchronised [30] (see 10.4.1). The example below 10.7 shows the main success scenario (happy/sunny day,) along with the associated SSD.

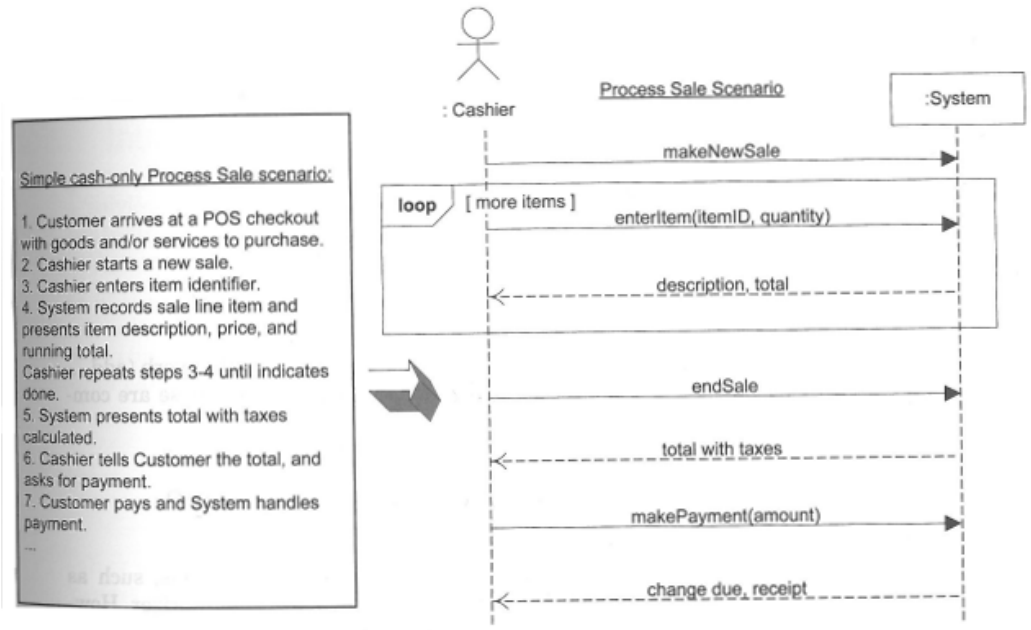


Figure 10.3 SSDs are derived from use cases; they show one scenario.

Figure 10.7: From Larman 57

Note: While Sequence diagrams are used in designing and documenting systems we use them here to explore only interactions with the system. Messages (events/requests) sent to the system are expressed without details about argument and return types.

10.5.1 Guidelines

Larman 57 suggests that:

- SSDs should be drawn for each of the main success scenarios of each use case, and frequent or complex alternative scenarios.
- System events should be expressed at the abstract level of intention rather than more concrete terms (for example "enterItem(itemID,quantity)" rather than "scanBarcode(itemID,quantity)".
- The actors interacting with the system may be other systems, its also possible that a software/hardware clock will generate events e.g. a timer generating events at a given interval.

- Inputs and outputs are shown as textural descriptions (named entities/objects) these should be included in the glossary, in Larmans example (10.7) "change due" and "receipt" are shown (the latter is clearly a complex report).

10.5.2 Should we sketch SSDs for everything?

In short, no. Larman 57 states that they should only be undertaken for the chosen scenarios of the next iteration and even then sketching should take no more than 30 minuets. Their key use is understanding the systems interaction with its users and collaborations with other systems.

10.5.3 SSDs within UP

SSDs are part of the use case model (a visualisation of the interactions implied by the use case,) but are not explicitly mentioned in the descriptions of UP or the later RUP [57]. However like other useful artefacts (e.g. Entity Relationship Diagrams) and activities (e.g. Collaboration Responsibility Cards,) they can be included in UP which encourages the adoption of anything which adds value [57].

SSDs are normally created during elaboration, their use in Inception is limited to instances where the identification of system operations is being used for comparison with other systems for estimation purposes (e.g. Function Point or COCOMO II) [57]. Clarification of the major input and output points of the system is most useful during elaboration and can be used as a starting point for other design activities [57].

10.6 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 10.6.1, then in teams of two produce:
    - A System Sequence Diagram from the 'work in progress' use case provided (10.6.2).
    - Take the use-case 5.6 as our current iteration subject and create an SSD (remember your only interested in the main success scenario (flow)).

Questions to think about (or what kind of exam question could I get on this).

- Describe System Sequence Diagrams (SSD) and explain how these can be used to compliment the UP, in your answer you should provide an example of a SSD and relate its use to the phases for UP.

10.6.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An

output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

10.6.2 Use Case

The team have produced a number of use case summarised at the highest level by the following use case diagram. The names of the use case here may differ slightly from those we identified during the tutorial.

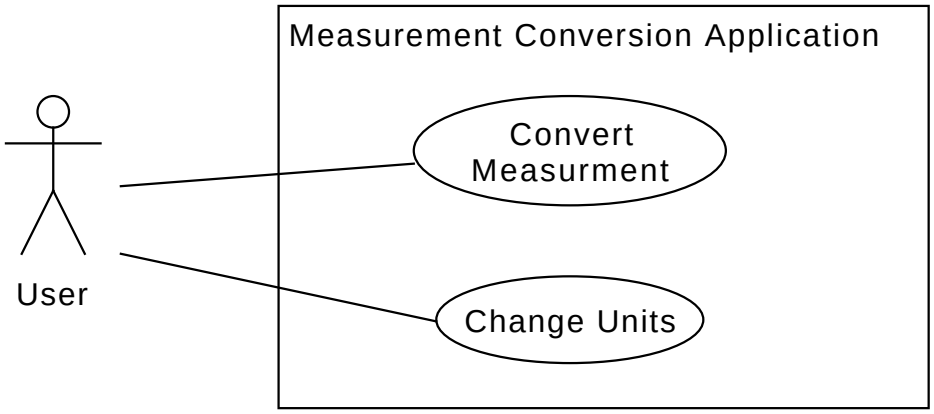


Figure 10.8: This is an incomplete use case based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough.

Use Case: Convert Measurement

UC1: Convert Measurement

Scope:	Measurement Conversion Application
Primary Actor:	User
Level:	user-goal

Stakeholders and Interests:

- User: Wants to convert between imperial and metric measurements of length.

Preconditions:

- Application has launched and the interface is displayed
- Interface is prepared for user input

Postconditions (success guarantee):

- Output is presented in a suitable format

Main Success Scenario:

1. User enters a measurement in the default unit (imperial,)
2. User requests a conversion (indicating they have completed their data entry).
3. System performs the conversion
4. System displays the converted measurement

Extensions (Alternative Flows):

- 1a. User changes units before entering a measurement.
- 2a. User enters nothing
    1. System does not attempt to convert
    2. System highlights the lack of input (e.g. change background of input field to red)
- 2b. User enters an incorrectly formatted measurement for the current conversion.
    1. System Displays an error message
    2. System highlights the incorrect input (e.g. change background of input field to red)
    3. User enters a measurement

Special Requirements:

- Must be possible to enter the measurement in a number of formats, e.g. 1'6", 1.5", 18" etc.
- The output must be such that the user can easily copy the output for use in another application.

Technology and Data Variations List:

1. ...

Frequency of Occurrence:	Expect the user to perform one or more conversions in a single session

---

Open Issues:

- Given that the input can be in a number of formats, which the system is to deduce. How will this be done?
- The output may be required in a specific format, should this option be given to the user and how? This could be a property set though a menu (with a default,) or some means of selection near the output.

10.7 Summary

In summary the process of developing an object orientated system requires that model the interactions between our system and external entities during the scenarios under consideration. System Sequence Diagrams are used to provide a first holistic view of the systems behaviour and as an introduction to sequence diagrams. The development of diagrams which show the systems external interactions though I/O events in which the system is treated as a black box have been used for decades [57]. The SSD is reasonably simple to create, though the sequence diagram adds a dimension of complexity. These diagrams are very much at the Conceptual Level (as with our domain object model,) we'll use the same raw UML (and more,) at a Specification Level later when we move into mechanistic design (see 10.4 for clarification of these terms).

Future classes will then move into looking at the design of the system and how this is translated to running code and tested. We'll start with the architectural design, before moving to mechanistic and detailed design as required. In our next iteration we'll add more detail to this early part of the elaboration phase and consider operation contracts.

10.8 Preparation for Next Week

10.8.1 Essential Reading

Larman 57 Provides an introduction to System Sequence Diagrams.

NOTE: Those of you who have the older version of Larman (before 3rd edition,) will find the order of the chapters has changed.

For the class after Christmas read chapter 12, which summarises what we've done so far and 13 which is what we're moving on to (don't panic, chapter 12 is very short).

Larman (2014k). *Applying UML and Patterns*, chapter Requirements To Design - Iteratively, pages 131–172. Prentice Hall PTR.
Larman (2014l). *Applying UML and Patterns*, chapter Logical Architecture

and UML Package Diagrams, pages 197–212. Prentice Hall PTR.

Next week we'll need more class diagram syntax (not a lot more but a bit,) so its worth going back over 8.6 and possibly reading chapter 15 of Larman.

Larman (2014j). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

10.8.2 Background Reading

The following cover the development of sequence diagrams, the writing style and level of detail is different in each. I'd like you to read both if possible and find other sources. Sequence diagrams have a lot of syntax which has changed (substantially,) between versions 1.* and 2.* of UML, so make sure you're reading a modern text.

Douglass, B. P. (2004e). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Object Orientation with UML 2 - Behavioural Elements, pages 139–215. Object technology. Addison Wesley, 3rd edition.
Larman (2014m). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Moving to Design: Architectural Design

11.1 Introduction

As part of the Elaboration phase we have considered one artefact of the domain modelling discipline, the domain object model and one requirements artefact the System Sequence Diagram (which complements the use case). The Elaboration phase this aims to refine the vision and over a number of iterations produces the core architecture. Along the way high risk elements will be resolved, the majority of the requirements identified (and the most important analysed in detail.)

At this stage we move from analysis to design, i.e. we start thinking about a solution for the current iteration in terms of collaborating objects. We'll start by figuring out how to structure these classes, both in terms of what classes we need and how they can be organised into sensible collections, the latter being termed architectural design [31]. Later we'll work out how the combined behaviours of this collection of objects will realise the use case or scenario which is the subject of this iteration in the mechanistic design [32].

NOTE: As we're working though this on a weekly basis and there's a lot to learn it may seem like we've been analysing for a long time. This process, i.e. getting from the start of inception to this point may only have taken a few hours for a skilled team [59]. In the real world proof-of-concept programming, resource management (finding people, software, ...), planning, setting up the environment (development, testing, continuous integration etc.,) may have taken many days or weeks [59].

11.2 Elaboration - Recap

Elaboration is a initial series of iterations over which we undertake three key activities [55]:

- **Development** – We develop the core risky architecture and test it.
- **Requirements** – Discover and stabilise the majority of requirements of the system.
- **Mitigate Risk** – We attempt to mitigate risks (reduce their potential frequency or severity using preventative measures,) or retire them (find that they are no longer a risk).

11.2.1 Iterations, Evolution and Change

During elaboration the team undertakes the serious investigation, implementation (programming and testing,) of the core architecture, clarifies requirements and tackles high risk elements [55]. Risk in this context may not related to technical risk (i.e. things that might be impossible given the limits of technology or our knowledge,) but refer to major risks to project success, that is features which if they are not correct will render it useless [55].

Elaboration proceeds over a number of iterations, focusing on important use-case or a scenario (or part scenario,) from a use-case [55]. The code artefact from this stage should be production quality (not a throw away prototype,) early iterations of the core architecture form the executable architecture (or **architectural baseline**).

In each iteration there will be a shift from analysis to design (which we are exploring today,) the focus on analysis in each of these these iterations will decrease as the vision and specification stabilise based on early programming, test and feedback [59].

Change is anticipated and it is normal and healthy to discover new requirements and change others [59]. Though early programming, test and feedback from demos we provoke change in the earlier iterations in order to clarify the purpose (think mini vision,) for this iteration and its requirements [59]. This helps the vision and requirements to stabilize as we keep refining our requirements based on feedback, early programming and testing (rather than speculation as in the waterfall approach) [60].

Some discipline is required here, you need to make sure your not starting to think in a waterfall way, e.g. trying to get all the analysis done up front, or working on the analysis of more than one iteration at a time. Stay focused on the current iteration. There is also a tendency to avoid questioning assumptions if the answer will result in additional work, e.g. adding new or incorrect requirements. This ostrich like behaviour is akin to taking a wrong turn and heading in the right general direction in the hope of picking up the right road. Its quicker and easier to make turn around as soon as possible and get back on track!

11.3 Moving to Design - A change of Emphasis

Larman 59 stresses the change of focus at this stage, we're moving from requirements and object orientated analysis - focused on **doing the right thing** i.e. learning about the outstanding goals of the system, the rules and constraints. To the design - focussed on **doing the thing right** i.e. designing a system which will satisfy the users needs.

11.3.1 Stages of Design

Douglass 31 discusses the phases of design in the ROPES methodology he proposes. Here he identifies, Architectural, Mechanistic and Detailed Design, terms we'll adopt in this model (see Table 11.1). The ideas they encapuslate are more general than ROPES, I've seen similar ideas expressed in terms of modules, interfaces and detailed design in assembler applications and Larman 62 focuses on Architectural Design and the deployment of Patterns in Mechanistic design. This is unsurprising, detail design is only really required when we need to be really sure it will work before we try it (e.g. critical systems,) or the actions of the object or its state changes are significant and complex.

Stage (aka Phase)	**Scope**	**UML**	**What is Specified**
Architectural	System-wide Processor-wide	Class Package Distribution	• Subsystems and components (grouping of similar software entities) • Concurrency and resources • Safety, reliability, security etc. • Deployment of software onto hardware
Mechanistic	Collaboration-wide Interaction Realisation	Sequence	• Collaborations of objects to fulfill functional requirements of a scenario/use case - i.e. the realisation of the use case • Instances of design patterns involving multiple objects
Detailed	Intra-object	Class Activity Statechart	• Details of data members (types ranges structure). • Details of methods (arguments etc.) • Algorithms used inside an object, e.g. search, sort, lighting, collision detection etc. • States, valid state transitions, actions during transition, conditions of transitions etc.

Table 11.1

Table 11.2: Phases or Stages of design, adapted from Douglass 31.
Larman 60 provides an excellent diagram which to some extent shows these

within UP, I've augmented this with some additions of my own (see 11.1).

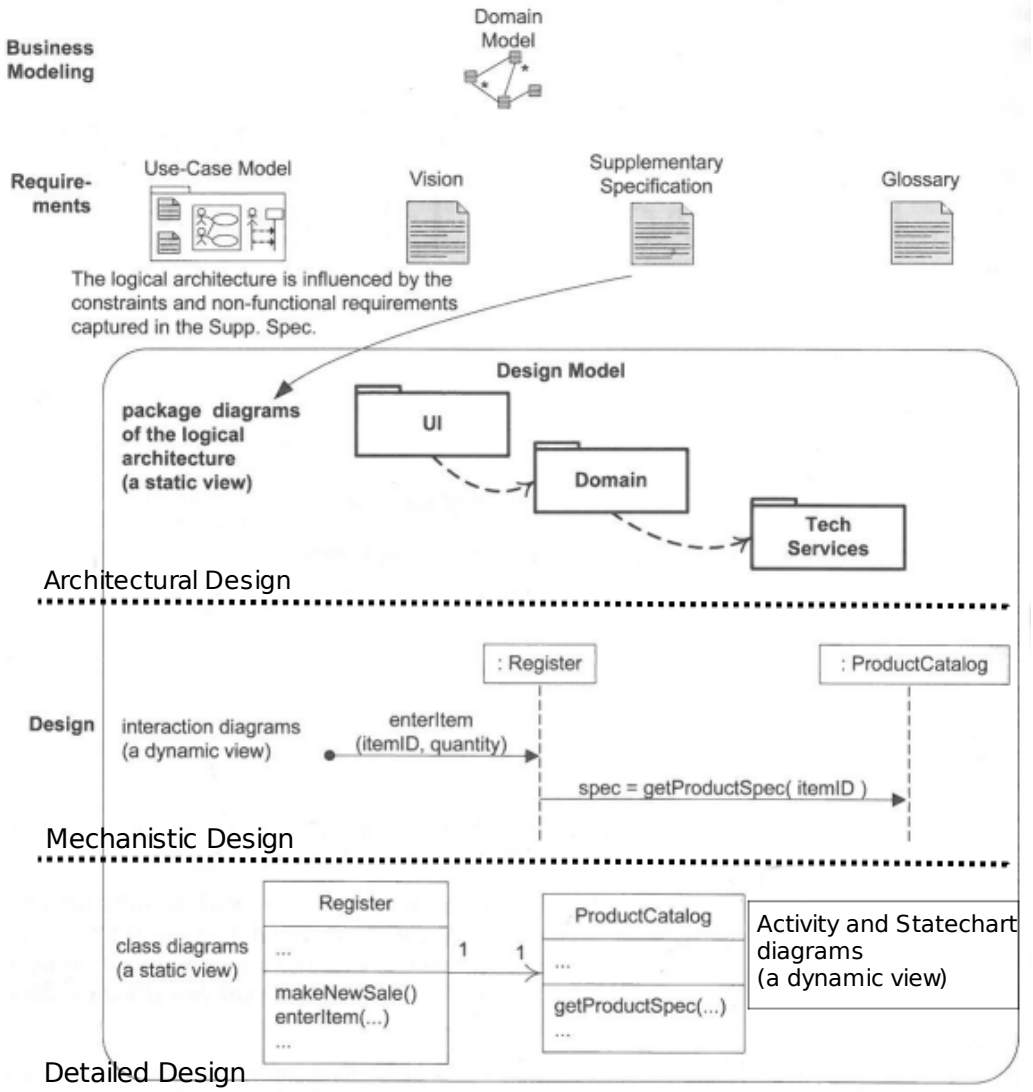


Figure 11.1: Adapted from Larman 60, note that we're currently considering the Architectural Design.

11.4 Architectural Design

In this first iteration we're going to briefly introduce architectural design focusing on the logical architecture of the system, however architectural design is a large topic [60]. If we have time we'll revisit this in a later iteration, if not I suggest you bare in mind that we've only scratched the

service and looked at one popular architectural pattern of many that might be considered.

While objects, classes and interfaces (which I know we've not done,) are small things it takes a number of them to achieve system-wide behaviour because of the complexity of current systems its unusual to find any system developed without some thinking in terms of larger scale structures [26]. There are normally two primary concerns, the first is managing large numbers of classes (possibly hundreds, or thousands,) and sharing their development/-maintenance amongst a team of developers or across many iterations [26]. The other is scalability of large runtime things, i.e. the physical system may be more complicated than a PC running an application, there may be several applications, these may run across a number of machines, or at a lower level processors (think embedded systems, graphics cards etc.) [26].
As a result the architectural design of a system is normally divided into the logical architecture and the physical (or deployment) architecture [60]. The former is concerned with the grouping of logically similar packages, for example we could have a vector, matrix and quaternion class all being part of a maths package. Douglass 31 recommends thinking of this as groups of things which exist at design time, i.e. types, classes etc. The physical architecture is concerned with how these are combined into executable elements and distributed across physical processing units, e.g. client server applicants, CPU and GPU, tiered applications etc. To extend our earlier example a video game may have vectors (from the maths package,) representing player position and velocity in both client and server applications running on separate hardware.

At this stage we're going to focus on the logical architecture and consider how this is influenced by the supplementary specification as seen in 11.1.

11.4.1 Example Logical Architecture

Larman 60 provides the following example of a architectural design (11.2, this shows a layered architecture.

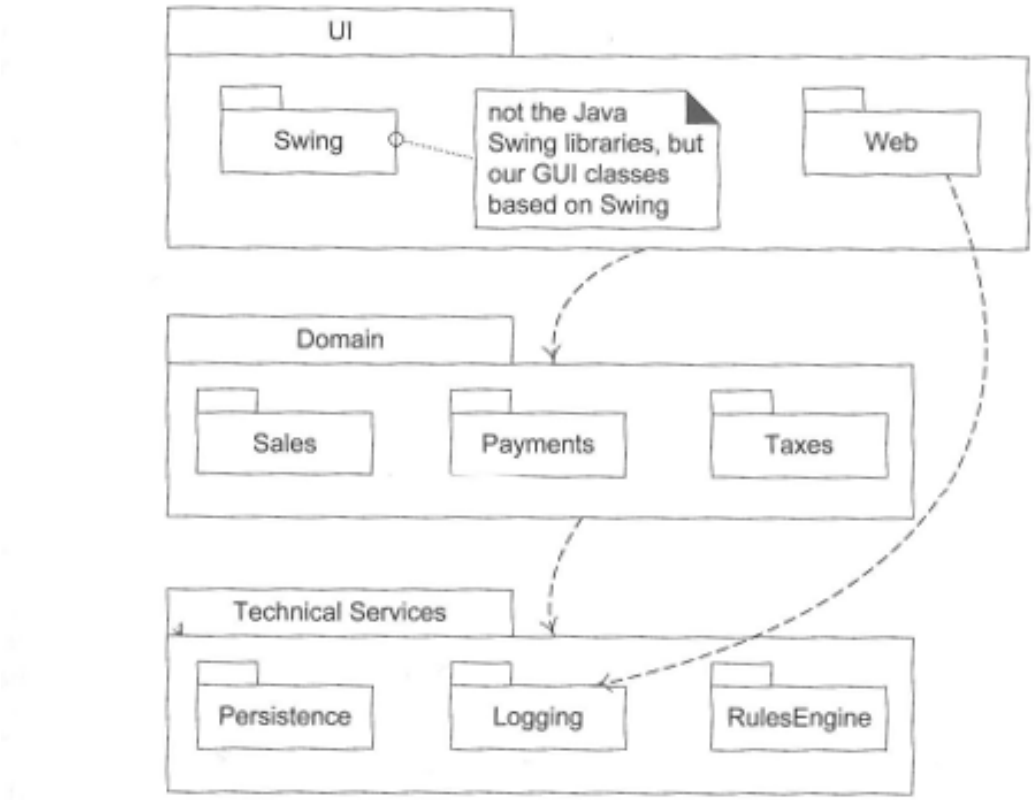


Figure 13.2 Layers shown with UML package diagram notation.

Figure 11.2: From Larman 60.

Leaving the architecture aside for now this is actually a UML package diagram 11.5, it shows named packages containing sub packages which themselves contain classes. There are in addition some specialised forms of package including subsystem and component however for now we'll just divide our classes up in to packages.

11.4.2 Layered Architectures

A layer is a high level grouping of classes, packages, components or subsystems responsible for a major aspect of the system (consider UI in the example above) [60]. Higher layers usually call upon services provided by lower layers, this is almost never reciprocated (i.e. the lower layers almost never request services from higher layers) [60]. Larman 60 suggests that a typical application would include:

- **User Interface** - All classes related with the presentation of informa-

tion to the user, we do not include the particular API we're use (e.g. Swing/.Net) but any classes which make use of them. In a computer game this might include window management, graphics context set-up and handling user input (keyboards, controllers, etc.)

- **Application Logic and Domain Objects** - All classes whose instances (software objects,) will represent domain concepts (e.g. Sale,) and that fulfil application requirements (e.g. calculate sale total). In a computer game context this would be the components of the game, e.g. car, track, tree etc. in a racing game or avatars, weapons, levels etc. in an FPS.

- **Technical Services** - Utility classes and general purpose objects, (e.g. strings, lists, etc.) subsystems providing supporting technical services e.g. database connections, error logging etc. These services are usually application-independent and may be used across other systems. In a game development context this may include wrappers for graphics APIs, maths libraries, physics, engines, graphics engines, audio engines etc.

Note: The focus of this module is the analysis, design and development of a software solution to a given problem, we're not interested in UI design, particular UI technologies (e.g. .Net) or particular Technical Services (e.g. maths libraries, database connectivity etc.). We'll focus on the domain layer and the core application logic [60], the other layers tend to be depended on the technology being used e.g. .Net, Qt or Java Swing for the UI etc.

11.4.3 Aside: Layers, Tiers and Partitions

Tiered architectures are reasonably common in enterprise computer systems (though the terminology seems to have gone out of fashion,) tier originally applied to a software layer but has become widely used to describe a separate processing node e.g. client tier [60].

The layers of an architecture are horizontal divisions, while partitions represent vertical divisions within a layer [60], for example the Technical Services layer may be divided into Logging, Data Persistence and Security see 11.3.

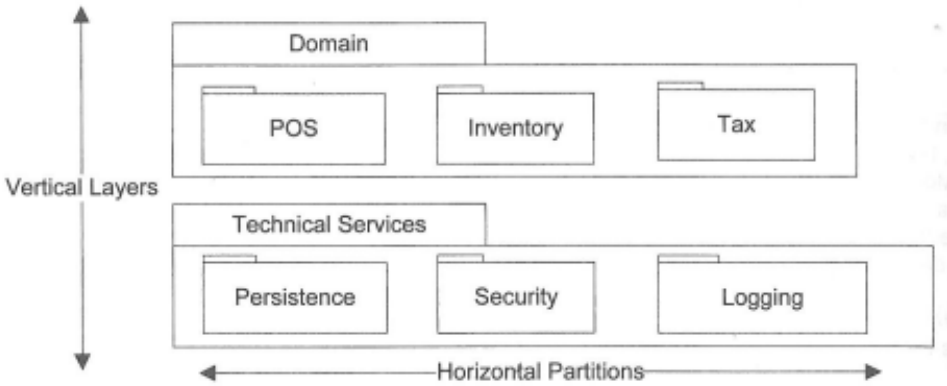


Figure 13.6 Layers and partitions.

Figure 11.3: From Larman 60

11.5 Packages Diagrams

UML package diagrams were mentioned in the above in terms of their use to describe the logical architecture, Larman 60 doesn't cover these in a huge amount of detail, showing only the basic syntax. I'll introduce a little more syntax than he does.

Package diagrams are (design-time,) models use to organise groups of related classes, each package can contain any classifier (i.e. classes, use-case or even other packages) [26]. Packages cannot be instantiated but they do provide a namespace for the model elements but no other semantics [26], that requires some explaining. So consider the following example 11.4.

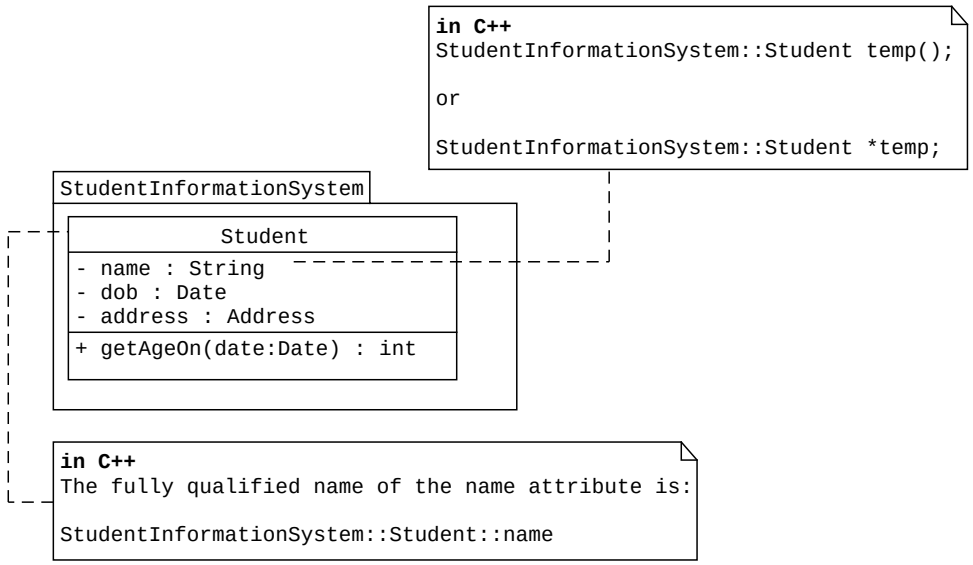


Figure 11.4: Namespaces and packages.

Packages look just like tabbed folders [26] and in their most basic form they are shown either with the name in the tab and the classes (or other classifiers,) within it (see 11.5), or as a tabbed folder with just a name [60] (see 11.6). The only valid relation between two packages is the dependency relation, this is shown as a dotted arrow [60]. This can be a little confusing as we've seen these arrows used to describe messages in sequence diagrams, however this that is an asynchronous return message and this is a dependency between packages (i.e. what the dotted arrow connects is part of how we interpret it).

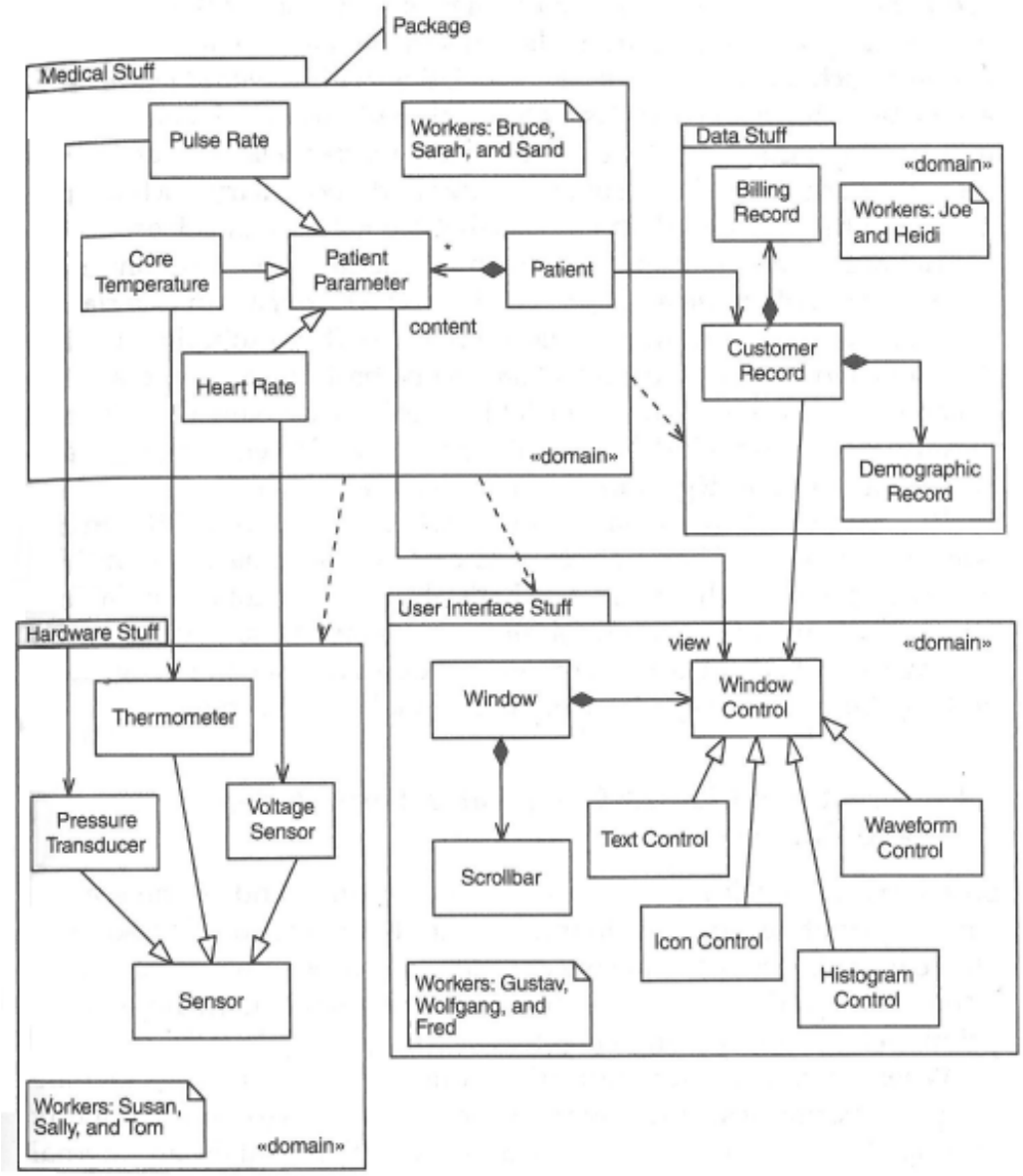


Figure 11.5: From Douglass 26 a package diagram for a patient management system.



Figure 11.6: Adapted from Larman 60

11.5.1 Reverse Engineering Package Diagrams

Larman 60 suggests that classes are usually organised into packages in a sketched package diagram during development, this often becomes outdated as future iterations add more classes and existing classes are re-factored. For this reason he recommends use a UML-CASE tool to reverse-engineer the package diagram from the code base. This could be generated at the end of each iteration and used to aid architectural design for the next, i.e. working out what can be re-used and what may need to be changed.

11.6 Guidelines for Architectural Design

The following guidelines for architectural design are mostly based on those provided by Larman 60, I'm hoping to include a few more as time goes on.

11.6.1 Design with Layers

Designing with Layers, also known as the layered architecture or the Layers pattern have been written about extensively [60]. The principle ideas are [60]:

- Large scale organisational structure organised as layers.
- Layers are distinct and offer a clear cohesive separation of concerns.
- "Lower" (meaning further from the user,) levels of the system are more general and low level (primitive,) providing general services.
- Higher layers are more application specific (though some may span applications within the same domain).
- Collaboration and coupling is from higher to lower; lower to higher coupling is avoided.

Just to stress this is not the only pattern but it is widely used and simple enough to us in an introduction.
According to Larman 60 the use of layers helps to address a number of problems:

- Source code changes ripple though the system (i.e. an unknown group of classes).
- Application logic is intertwined with UI, this makes re-use much more difficult.
- General technical services (and possibly general business logic,) is intertwined with application logic meaning this cannot be re-used or replaced.
- There is high coupling between different areas of concern, this makes distributing work between developers or teams of developers difficult.

While the pattern is fairly general the number of layers and their purpose varies between domains, the example provided by Larman is shown in 11.7. The advantages of the layered architecture all stem from the separation of concerns, high level application specific from general services [60]. This reduce coupling and dependency and improves cohesion leading to increased reuse potential and clarity [60]. Other benefits include ease of replacement of underlying services (though application and UI can also be replaced,) ease of distribution and improved team based development (less overlap between developers/teams) [60].

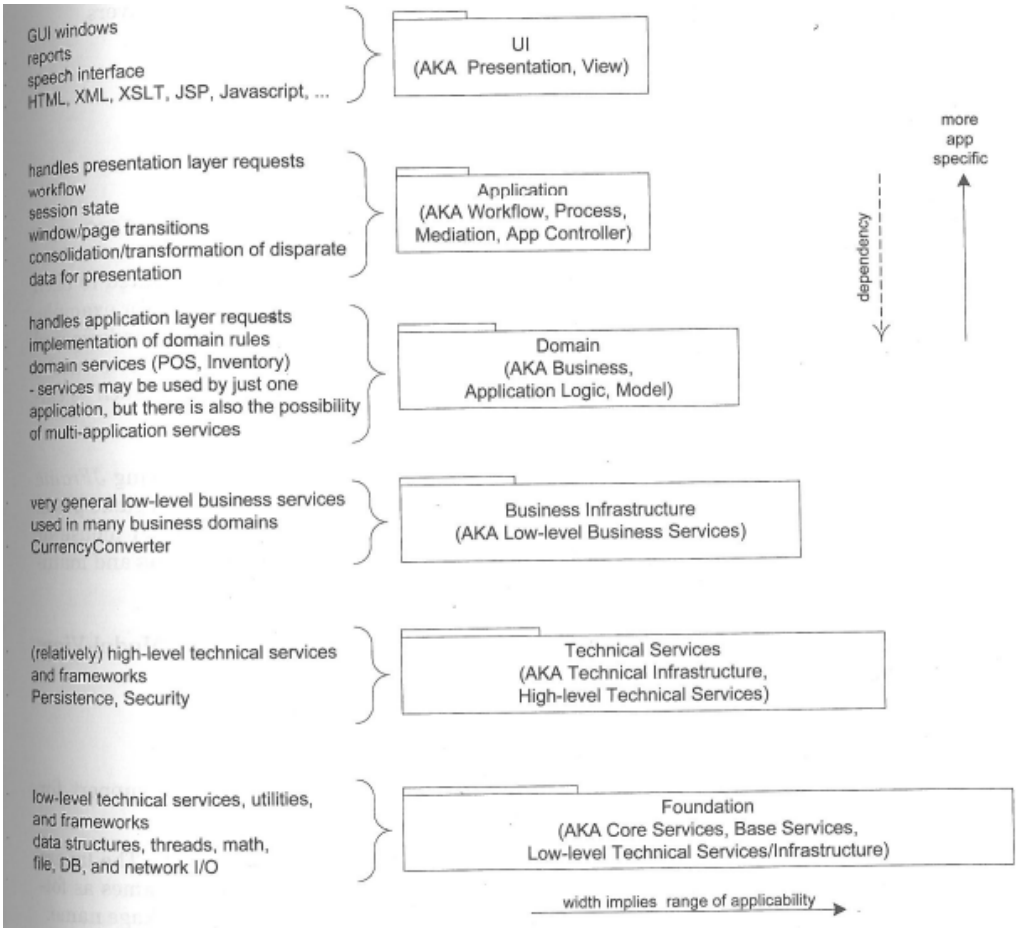


Figure 11.7: From Larman 60.

11.6.2 Maintain Separation of Concerns

Separation of Concerns is an often used term in the literature regarding architectural design, in this case we mean that objects in a given layer should be responsible for one area of the system, e.g. the UI layer should be responsible for creating windows, wigets etc. and gathering user input via the mouse and keyboard [60]. It should not contain and domain logic, such as tax calculations etc., or undertake any low level operations such as storing files, opening database connections etc. [60]. Maintaining separation of concerns allows us to maintain high cohesion, i.e. the classes of the layer are not highly dependent on (or tightly coupled to,) classes outside of the layer [60]. We noted earlier that as part of the pattern we try to ensure that the layer is depended only one the layer below, we try to make this interaction as decoupled as possible (the dependent object knows nothing of the workings

its dependency).

11.6.3 Name Packages Carefully

Careful package naming can help us here, use the application name in lower level libraries discourages re-use [60]. Larman 60 provides some examples using Java, I've made my own along the same principles with C++ syntax.

Bad	UWTSD::Registry::StudentInformationSystem::DatabaseHandler
"	UWTSD::Registry::StudentInformationSystem::OracleDBHanlder
Better	UWTSD::DatabaseHandler
OK	UWTSD::Registry::StudentInformationSystem::UI::*

11.6.4 Do now show Physical Resources below the Foundation/Technical Services Layer

I've included this as Larman 60 suggests its important however, we've not really looked at the physical layer and the distribution of resources so maybe its premature. The logical architectural design should not contain and reference to deployed components such as MySQL databases, local APIs or Services e.g. LDAP/FreeIPA/Active Directory the logical design should stop at the point our components wrap the functionality of these entities. For example, in the case of MySQL database we would have a class (or group of classes,) responsible for database connectivity, in a game we'd have a graphics library which provided us with access to our chosen graphics API.

11.7 Application Logic in the Domain Layer

This is a key concept in Object Orientation!
Larman 60 is keen to point out that understanding where the application should go within these layers of objects (and to some extent where it shouldn't go) is a critical part of Object Orientated design. Adapting Larmans example, consider a simple application with a Java GUI to calculate VAT. The system has some UI logic, i.e. what happens when the calculate button is pressed and some application logic i.e. how to calculate VAT. We could create a single class and put both the UI and application logic in but this is messy (and not in the spirit of OO).

The recommended OO approach [60]:

- Create software objects with names and operations similar to the real-world domain.
- Assign application logic responsibilities to them.

In Larman 60 the Point of Sale example involves sales and payments, so in our software system create a Sale and Payment class and give them 'application logic responsibilities'. This means that in addition to representing the real world element e.g. a sale the Sale class also takes responsibility for associated business logic, e.g. calculating the total, logging errors, etc. This kind of object is called a **domain object** and occupies a layer containing application logic connected to the domain (the **domain layer**) [60].

11.7.1 Domain Layer and Domain Object Model

Larman 60 suggests that the domain model should provide inspiration for the domain layer, we have to remember however that they are very different, the former being a conceptual perspective model (part of the analysis,) and the other being a design model. However by using the same names we lower the representational gap, is clear that the concept Sale (from the domain model,) and the design class Sale in the design model are related [60] (this is shown really well in 11.8. This is more informative than it being called MyClass, Basket, Goods etc.

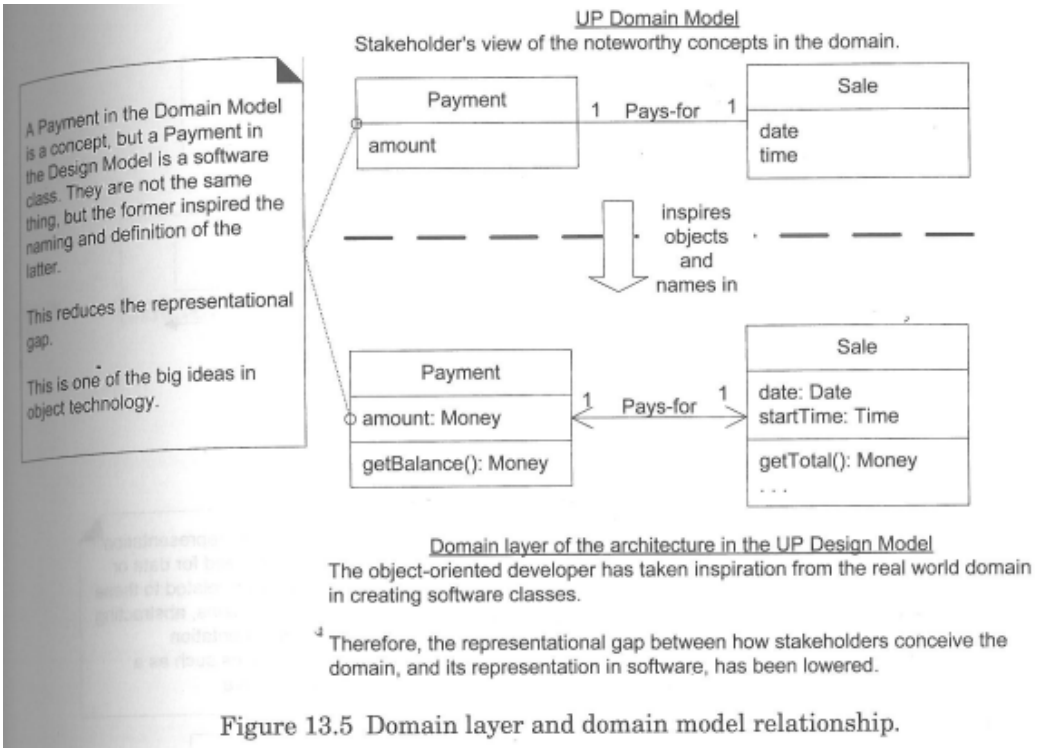


Figure 13.5 Domain layer and domain model relationship.

Figure 11.8: From Larman 60.

11.8 The Model View Separation Principle

The separation of the UI from the domain classes (and therefore the application logic,) raises some questions about how visibility of classes within these packages should be managed and how non-window classes will communicated with windows (to display updates, error messages, results etc.) [60].

The principle has two parts [60]:

1. Do not connect or couple non-UI objects directly to UI objects.
    - Note the direction of the above non-UI $\rightarrow$ UI is a **NO**, but UI $\rightarrow$ non-UI is fine.
    - Example: The Tax class calculates VAT on a given price, then calls a method on the VatForm class to put the message on screen.
    - The window is tied to the application, ideally the non-windowing

object may be re-used in another application (or the same application with a different interface).

2. Do not put application logic in the UI object.

    - UI Objects should only initialise UI elements and receive UI events (such as mouse/keyboard events) themselves.
    - All other behaviour is delegated to domain objects.

In the context of our layered architecture the model is the domain layer and the view is the UI objects (these may be windows, Web pages, etc.) this terminology originated in the late 1970s and was originally used in this context [60]. In the original Smalltalk-80 pattern it was a key part of the MVC (Model View Controller,) pattern in which the domain object (model,) were separated from views (GUI widgets) and controllers (keyboard and mouse event handlers) [60].

One noteworthy exception to the Model View principle is the observer pattern (which we'll discuss later **??**) where domain objects send messages to UI objects viewed only in terms of an interface [60]. This is a little too much to fully explain at this stage but essentially the UI widget registers istardpoweredtrombone tself as an interested in updates from the domain level widget and is passed a message when something its interested in happens. Decoupling still occurs, the domain class only knows that a given interface (e.g. PropertyListener in Java,) is implemented by the object it has no other knowledge of the element. I know we've not done interfaces (its coming up **??**) best to think of them as an abstract parent (with only pure virtual methods) for now.

A further part of this principle is that as the domain classes contain all the information and the behaviours related to application logic the classes of the UI layer will be very thin, responsible only for input and output [60]. Larman suggests examples including a JFrame in Java and Web JSP pages, perhaps a more tangible example is some of the work you've done in other modules? I know some of the examples in my Qt notes violate this principle :-(

The motivation for ensuring Model-View separation include:

- To ensure that the modeller (designer,) is not distracted by the user interface by ensuring that the domain layer is dedicated to the domain.

- To allow experts in different fields to work on the development of the UI and domain layers, UX (user experience,) is a growing field and one which is taken very seriously.
- Minimise the impact of interfaces changes and additions to the UI layer (additional views), it should be that adding a new report, data view etc. has no effect on the underlying domain model.
- Allow many views of the same underlying data, e.g. show tabular format and summary charts at the same time.
- To allow execution of the domain (model) layer independently of the interfaces, this allows batch processing perhaps using the application via scripts on the command line and facilitates extensive, automated testing of this key layer.
- Reduce porting effort, operating system and library dependencies are likely to be found in the UI later (e.g. if we're using Qt this part of the system will be heavily dependent on Qt,) porting the domain (model) part is much easier if it contains no API/platform specific code.

11.8.1 Layers and SSDs

Earlier we looked at System Sequence Diagrams (SSDs) as a useful way to sketch the key input and output operations of the system and visualise the principle flow [60]. These focus on the operation of the system and ignore the UI, a good UI layer will transparently capture both the operation and the require data and forward (or delegate,) this to the domain layer [60]. Put simply the messages sent from the UI layer to the domain layer (the ones we need to model the all important domain layer,) will be the ones identified as input/output messages in the SSD. This is show in 11.9.

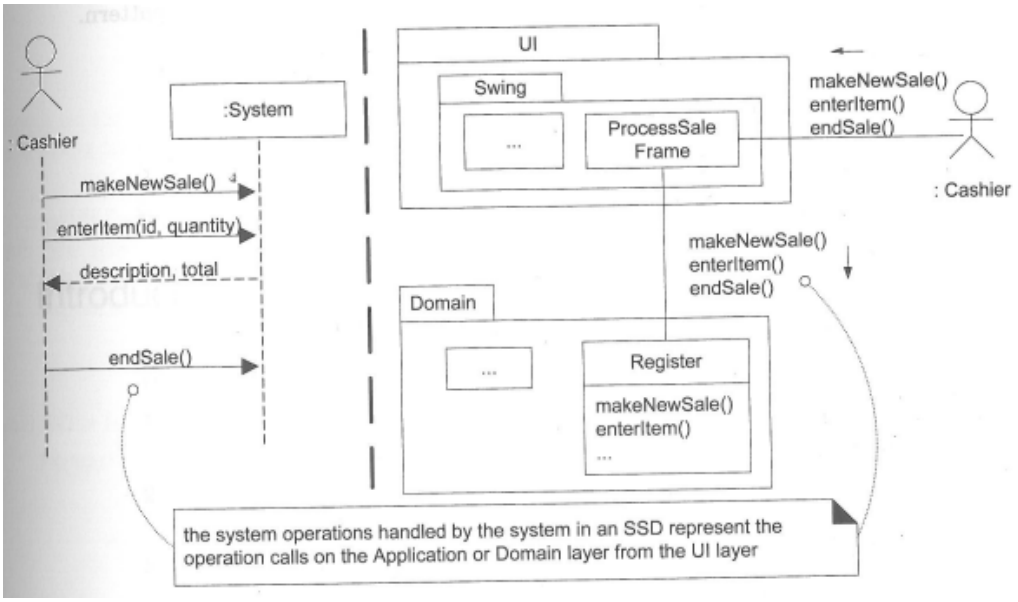


Figure 11.9: From Larman 60.

11.9 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 10.6.1, and consider the use case 11.9.2 and 11.9.3 then in teams of two produce:
   - An architectural design (I strongly suggest you consider layers)
   - Suggest candidate classes for the UI and Technical Services layer (you'll have to base this on your study in other modules).
2. The following is a list of classes from a fictional application, organise these into appropriate packages in a layered architecture: ChatClientWindow, ChatClient, ClientConnection, Message, ServerConnection, Server, Logger.

Questions to think about (or what kind of exam question could I get on this).

- Explain the Model-View Separation Principle and why such an approach would be used in the development of a commercial video game.

11.9.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches

and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

11.9.2 Use Case

The team have produced a number of use case summarised at the highest level by the following use case diagram. The names of the use case here may differ slightly from those we identified during the tutorial.

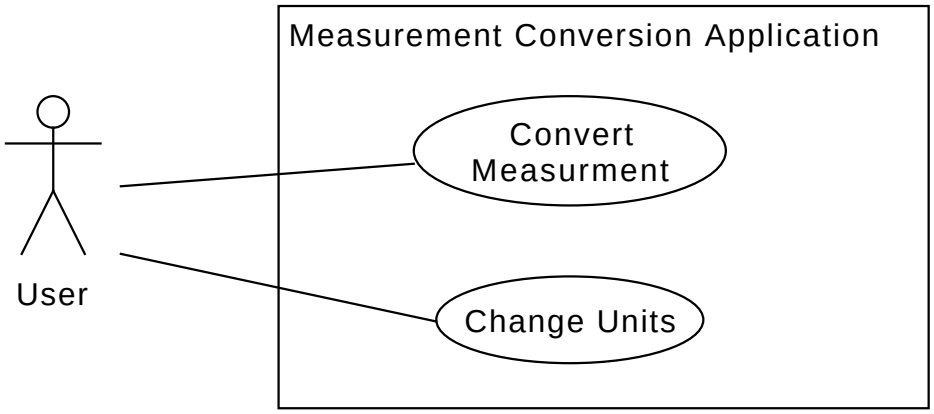


Figure 11.10: This is an incomplete use case based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough.

Use Case: Convert Measurement

UC2: Convert Measurement-L11	
Scope:	Measurement Conversion Application
Primary Actor:	User
Level:	user-goal

Stakeholders and Interests:

- User: Wants to convert between imperial and metric measurements of length.

Preconditions:

- Application has launched and the interface is displayed
- Interface is prepared for user input

Postconditions (success guarantee):

- Output is presented in a suitable format

Main Success Scenario:

1. User enters a measurement in the default unit (imperial,)
2. User requests a conversion (indicating they have completed their data entry).
3. System performs the conversion
4. System displays the converted measurement

Extensions (Alternative Flows):

- 1a. User changes units before entering a measurement.
- 2a. User enters nothing
    1. System does not attempt to convert
    2. System highlights the lack of input (e.g. change background of input field to red)
- 2b. User enters an incorrectly formatted measurement for the current conversion.
    1. System Displays an error message
    2. System highlights the incorrect input (e.g. change background of input field to red)
    3. User enters a measurement

Special Requirements:

- Must be possible to enter the measurement in a number of formats, e.g. 1'6", 1.5", 18" etc.
- The output must be such that the user can easily copy the output for use in another application.

Technology and Data Variations List:

1. ...

Frequency of Occurrence:	Expect the user to perform one or more conversions in a single session

---

Open Issues:

- Given that the input can be in a number of formats, which the system is to deduce. How will this be done?
- The output may be required in a specific format, should this option be given to the user and how? This could be a property set though a menu (with a default,) or some means of selection near the output.

11.9.3 Domain Object Model

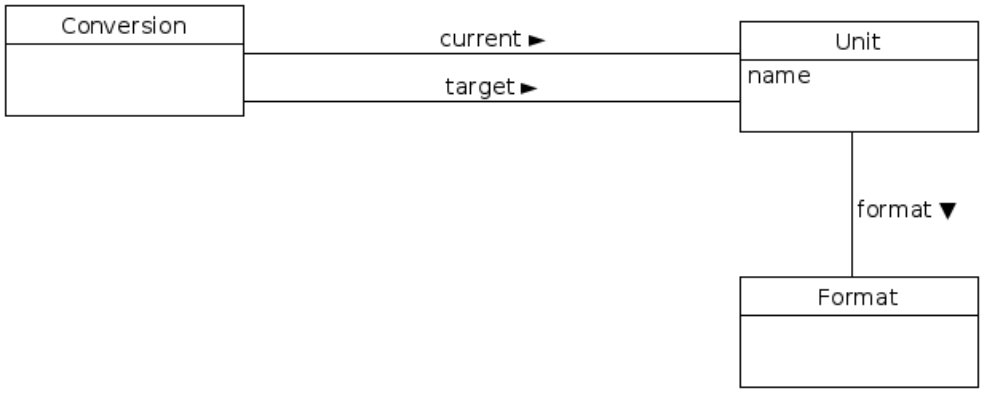


Figure 11.11: This is a domain object model based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough. I've avoided describing this as a draft (suggesting its incomplete,) or using any other term which would suggest it is final. In the spirit of Agile Modelling we are using it to inform sketches of the design so we can discuss and evaluate design options. Changes may be made as a result of this discussion and its probably that future iterations would contain some of the conceptual classes.

11.10 Summary

The design aspect of each elaboration phase begins with some consideration of the architecture of the system and the elements which are being added (or perhaps modified,) in this iteration. Later stages of design will consider how use case and scenarios are mapped to objects within this architecture and the objects themselves (the latter begins in some respects with the domain object model). We have introduced some new UML (though a limited amount in terms of the package diagram and its use for logical architectural design).

In the short term we'll lay the foundation for the next phase of elaboration, there is more UML to cover and some important theory (low level design patterns or principles,) which needs to be carefully introduced before we can

proceed. We'll then look at the process of use case elaboration which in short is the mapping of use case (or scenarios) to the objects of our architecture.

11.11 Preparation for Next Week

11.11.1 Essential Reading

Larman 60 Provides an introduction to Architectural Design

Larman (2014l). *Applying UML and Patterns*, chapter Logical Architecture and UML Package Diagrams, pages 197–212. Prentice Hall PTR.

Reading for next week - TBA (when I've written the notes).

11.11.2 Background Reading

Douglass, B. P. (2004a). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Object Orientation with UML 2 - Structural Aspects, pages 77–139. Object technology. Addison Wesley, 3rd edition
YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Designing with Patterns/Principles: Mechanistic Design - Part 1

12.1 Introduction

As part of the Elaboration phase we have considered one artefact of the domain modelling discipline, the domain object model and one requirements artefact the System Sequence Diagram (which complements the use case). The Elaboration phase this aims to refine the vision and over a number of iterations produces the core architecture. Along the way high risk elements will be resolved, the majority of the requirements identified (and the most important analysed in detail.)

We've started to move from analysis to design, i.e. we're thinking about a solution for the current iteration in terms of collaborating objects. Last week we considered how to structure these classes, both in terms of what classes we need and how they can be organised into sensible collections, the latter being termed architectural design [31]. We're now starting to consider how the combined behaviours of this collection of objects will realise the use case or scenario which is the subject of this iteration in the mechanistic design [32]. We're going to look at Responsibility Driven Design, which provides coherent guidance over the design process. In essence we allocate responsibility for required behaviour to classes in our architecture (adding new ones where required,) and then use OO principles or patterns to guide the design of their interaction.

NOTE: Remember we're looking at a single iteration in elaboration, there would be more than one of these in a real project and each time you'd go back to the start.

ANOTHER NOTE: This is a complex topic, we're going to take a few weeks to cover. It is however seminal to how object orientated systems are designed and built.

12.2 Elaboration - Recap

To recap elaboration is a initial series of iterations over which we undertake three key activities [55]:

- **Development** – We develop the core risky architecture and test it.
- **Requirements** – Discover and stabilise the majority of requirements of the system.
- **Mitigate Risk** – We attempt to mitigate risks (reduce their potential frequency or severity using preventative measures,) or retire them (find that they are no longer a risk).

12.2.1 Iterations, Evolution and Change

During elaboration the team undertakes the serious investigation, implementation (programming and testing,) of the core architecture, clarifies requirements and tackles high risk elements [55]. Risk in this context may not related to technical risk (i.e. things that might be impossible given the limits of technology or our knowledge,) but refer to major risks to project success, that is features which if they are not correct will render it useless [55].

Elaboration proceeds over a number of iterations, focusing on important use-case or a scenario (or part scenario,) from a use-case [55]. The code artefact from this stage should be production quality (not a throw away prototype,) early iterations of the core architecture form the executable architecture (or **architectural baseline**).

In each iteration there will be a shift from analysis to design (which we are exploring today,) the focus on analysis in each of these these iterations will decrease as the vision and specification stabilise based on early programming, test and feedback [59].

Change is anticipated and it is normal and healthy to discover new requirements and change others [59]. Though early programming, test and feedback from demos we provoke change in the earlier iterations in order to clarify the purpose (think mini vision,) for this iteration and its requirements [59]. This helps the vision and requirements to stabilize as we keep refining our requirements based on feedback, early programming and testing (rather than speculation as in the waterfall approach) [60].

Some discipline is required here, you need to make sure your not starting to think in a waterfall way, e.g. trying to get all the analysis done up front, or working on the analysis of more than one iteration at a time. Stay focused on the current iteration. There is also a tendency to avoid questioning assumptions if the answer will result in additional work, e.g. adding new or incorrect requirements. This ostrich like behaviour is akin to taking a wrong turn and heading in the right general direction in the hope of picking up the right road. Its quicker and easier to make turn around as soon as possible and get back on track!

12.3 Towards Object Design

Larman 63 suggests that object design is usually undertaken in one of the following ways:

1. **Code.** Design-while-coding, ideally with a powerful IDE which supports refactoring and other iterative practices. Here Larman mentions "Refactoring" a plug-in for visual studio which provides support for refactoring (making changes to improve the design of existing code). This is often the method of choice for simple problems.

2. **Draw, then code.** Drawing some UML, on a whiteboard, on paper, or in a tool then switching to 1. When tackling new, complex or important problems a draw then code approach may significantly reduce the refactoring effort required over 1. It also provides an opportunity for pair/team based design which may yield more considered solutions. If you have no idea where to start then some rough ideas on paper are the place to start.

3. **Only Draw** Here the tool somehow generates everything. Tools which can do this are rare and expensive, the diagrams for generation have to be very highly detailed and there may be additional detail specified in a text based programming language attached to the tool.

12.3.1 Agile Modelling and Lightweight UML Drawing

Agile modelling focuses on using UML as a sketch to improve understanding and communicate while reducing the drawing overhead [63]. My concerns in terms of this module are that the time taken to learn UML and to find your way around some of the UML tools at best distracts from the true purpose of designing the system and at worst puts you off design altogether. Larman 63 recommends that agile modelling should include:

- **Modelling with others**. Using shared drawing areas, e.g. multiple whiteboards, large sheets of paper etc. Electronic alternatives exist but are not freely available collaborative drawing tools may be a partial solution e.g. https://www.draw.io/.
- **Creating several models in parallel**. Moving between dynamic (sequence diagrams,) and static (class diagrams,) views of the system is essential to capture the detail as our design develops.

UML CASE Tools

While we're advocating the use of hand drawn communal diagrams these may not always be suitable/desired, if for example the project needs to be documented over a long time, or is to be implemented by another team more detailed clearer diagrams may be required. While we can take photos of whiteboards to capture our models, formalising these may sometimes be required (N.B. for the course work I want to see at least some of the diagrams in this format). Larman 63 suggests that ideally the UML tool should allow good integration with a strong IDE and be able to reverse engineering both structural and dynamic diagrams (the latter is uncommon, but tools keep improving).

Time spent modelling and the quality of the models created

For each time-boxed iteration (of around 3 weeks,) for an iteration in elaboration a few hours of a single day will be spent sketching diagrams [63]. These can be photographed or otherwise stored (depending on format,) and initial development of these designs can begin, note that the code and design may diverge but we need a starting point [63]. There may be other shorter modelling sessions focused on particular issues throughout the iteration. The models themselves are being used to articulate design ideas for discussion/exploration the syntax of the UML within them is less important [63], though you need to be close enough so that others

can read the diagrams. Think about mispelling words, if its close enough people know what you mean but a long jumble of letters is incomprehensbile. Larman 63 argues that what matters are design skills, being able to assign responsibilities to objects and deploy design principles and patterns.

Note: As suggested in my earlier note, we spent quite a short amount of time drawing UML and even then the drawing is the easy part. Figuring out what needs to be drawn i.e. the actual design of the code is the hard/important bit.

12.3.2 Stages of Design

Douglass 31 discusses the phases of design in the ROPES methodology he proposes. Here he identifies, Architectural, Mechanistic and Detailed Design, terms we'll adopt in this model (see Table 11.1). The ideas they encapuslate are more general than ROPES, I've seen similar ideas expressed in terms of modules, interfaces and detailed design in assembler applications and Larman 62 focuses on Architectural Design and the deployment of Patterns in Mechanistic design. This is unsurprising, detail design is only really required when we need to be really sure it will work before we try it (e.g. critical systems,) or the actions of the object or its state changes are significant and complex.

Stage (aka Phase)	**Scope**	**UML**	**What is Specified**
Architectural	System-wide Processor-wide	Class Package Distribution	• Subsystems and components (grouping of similar software entities) • Concurrency and resources • Safety, reliability, security etc. • Deployment of software onto hardware
Mechanistic	Collaboration-wide Interaction Realisation	Sequence	• Collaborations of objects to fulfill functional requirements of a scenario/use case - i.e. the realisation of the use case • Instances of design patterns involving multiple objects
Detailed	Intra-object	Class Activity Statechart	• Details of data members (types ranges structure). • Details of methods (arguments etc.) • Algorithms used inside an object, e.g. search, sort, lighting, collision detection etc. • States, valid state transitions, actions during transition, conditions of transitions etc.

Table 12.1: Phases or Stages of design, adapted from Douglass 31.
Larman 60 provides an excellent diagram which to some extent shows these within UP, I've augmented this with some additions of my own (see 12.1).

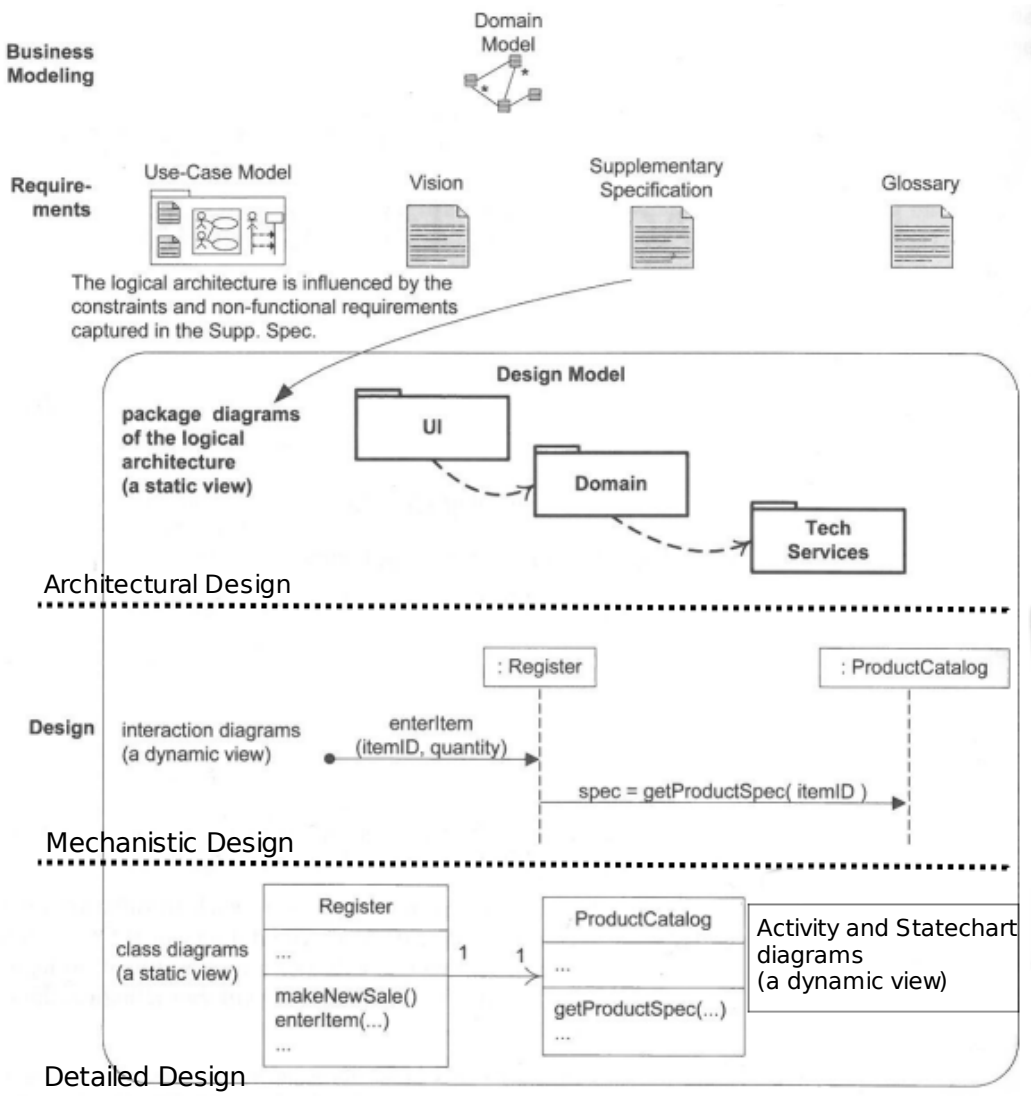


Figure 12.1: Adapted from Larman 60, note that we're currently considering the Architectural Design.

We're at the mechanistic stage, where we work out how our behaviour maps to objects in our system. This is not a two step process i.e. we don't finish architectural design and start mechanistic, we draft the architecture and then as we start to consider how object behaviour is mapped to this architecture we add classes as required.

12.4 Mechanistic Design

12.5 Models in Mechanistic Design

We've talked a lot about models the question is what are the models, well there are static and dynamic views of the objects being considered for this iteration both have their own diagrams [63]:

- **Dynamic Diagrams** These express the dynamic view of the system, i.e. for a given scenario or use-case what actually happens. Which object is responsible for handling the system request, does it coordinate the activities of others to complete the request (which ones and how)? **Important:**
  - These diagrams are where we make the big decisions about design.
  - Novice designers often think the static diagrams are more important, their not.
  - This is where we deploy Responsibility-Driven-Design (RDD), GRASP Principles and Design Patterns.
  - These dynamic diagrams are key to discovering the messages passed (i.e. method calls, signal handlers etc.,) and information exchanged, which can be simple types or additional objects.
- **Static Diagrams** These show the structure, which objects are composed of others (via composition, aggregation and inheritance) etc.
  - The static models provide us with the attributes of the classes to be developed.

Other Models

There are other static diagrams in **Architectural Design** we've seen the Package Diagrams being used to design the logical architecture, later we'll encounter deployment diagrams. In **Detailed Design** design within an object we use state chart (or state machine,) diagrams to design objects with complex state behaviour and activity diagrams to design complex processes (algorithms or work-flows,) within an object.

12.6 Use Case Realisation

Use Case Realisation or perhaps more accurately scenario realisation is the process by which we describe the design (in terms of objects,) or one or more use case [64]. The term Use Case Realisation comes from UP and is used to reinforce the connection between the use case in analysis and the design of the system [64]. The RUP definition is "A use-case realisation describes how a particular use case is realized within the Design Model, in terms of collaborating objects" [RUP].

Larman 64 argues that object orientated design is simply a question of assigning responsibilities to objects and designing collaborations and that both can be logically rationally explained and learned. There is no "magic" no unjustifiable decision the assignment of responsibilities and the choice of collaborations can always be explained. That's not to say that we don't get so used to doing something that it becomes second nature.

Larman 65 suggests that the assigning responsibilities and designing collaborations is complex as the decisions taken have far reaching consequences. Also that mastering OO design requires the deployment of a large number of soft principles (with many degrees of freedom,) this seems daunting but these principles can be named and learned. The distinction between patterns and principles is a difficult one, from my interpretation of the text a principle is a very small pattern.

12.6.1 Orientation: What's next?

The plan is as follows:

- **GRASP** We'll begin by looking at the simple object orientated design patterns. These provided guidance for solving really small design problems, e.g. Who should create object X?
- **GoF Patterns** We'll the move on to look at some (or perhaps all) of the Gang of Four patterns, these are well know and well used design patterns which will give us a flavour for their use.
- **More Class Diagrams and Sequence Diagrams** Along side the principles and patterns we'll be looking at more UML, which we'll need to visualise the patterns - though we'll build some in C++ as we go!

Essentially we're working up to realisation, starting small with the GRASP - small patterns (or principles,) these should be enough to get us started.

12.7 Responsibilities and Responsibility Driven Design

Designing software objects by thinking in terms of responsibilities, roles and collaboration is both popular and well established in Software Engineering [65]. This is part of a larger approach called responsibility-driven design (or RDD,) but may also be familiar from other techniques such as CRC cards **??**. Larman is keen to point out the generality of this approach and we can think of the work we've done with structures and functions in these terms.

12.7.1 Aside: Class Responsibility Communication Cards

Class Responsibility Communication (CRC) cards are a popular text orientated technique created by Kent Beck and Ward Cunningham (founders of the ideas of XP and design patterns) [63]. Larman 63 suggests the primary reason for this may be cognitive styles, however other factors experience, comfort etc. may contribute. Using CRC cards for design is a reasonably social activity (see next paragraph,) again this doesn't suit everyone.

CRC cards are essentially index card (which you may have never seen - see ,) with written on it the name of the class it represents and its responsibilities (what it is responsible for knowing and doing,) and who collaborates with it in each case [63]. The example below is a hand drawn template from Larmans book **??**.

The process of using the cards involves taking the conceptual classes as a starting point sitting around a table working though "what if' scenarios or use case with the cards [63]. This is almost a role playing exercise with individuals playing the part of various classes, oversights and omissions can be easily spotted and this provides a good starting point for responsibility based design. We however are starting lower down with the GRASP patterns, however the CRC technique is useful at a higher level.

12.7.2 RRD

In RRD objects are attributed responsibilities, in terms of what they are responsible for knowing and what they are responsible for doing [65]. The UML defines responsibilities as "A contract or an obligation of an element in its relationship to other elements" [**?**], this is an abstraction [65].

There are two types of responsibility knowing and doing [65]:

- **Doing** - Responsibilities may include:
  - Doing something itself, e.g. a calculation, creating and object etc.
  - Causing something to happen in another object, or changing its state.
  - Controlling and coordinating activities in other objects (e.g. **??**)
- **Knowing** - Responsibilities may include:
  - Holding data itself in private attributes
  - Knowing about related objects
  - Knowing about data it can derive or calculate

One key observation (which sort of links us back to CRC cards,) is that objects can collaborate (work with other objects,) while fulfilling the responsibilities in either knowing or doing [65]. Another is that while methods will ultimately full fill the responsibilities of a class, responsibilities are not methods [65], I've tried to adapt some examples from Larmans book to give you an idea.

Class	Responsibility	Collaboration Classes	Collaboration Method
Sale	Responsible for creating *SalesLineItem*s (doing)	None	None
Sale	Responsible for knowing the sale total (knowing)	*SalesLineItem*s	*Sale* is associated with *SalesLineItem*s which are part of the sale. It calculates the total by asking each in turn for their totals (price * quantity). This uses *getSubTotal()* in *SalesLineItem*.

Table 12.2: Examples of objects with responsibilities, appropriate collaborators and collaboration methods (adapted from Larman 65

Guideline: Look for Responsibilities in the Domain Model

Any attributes which are shown in the domain model are things which the equivalent class in the design model is responsible for knowing [65]. For example, if the *Sale* class from Larmans example has an attribute called *time* in the domain model it follows that class it inspires with the same name in the design model (remember our aim to lower the representational gap,) will be responsible for knowing the time of the sale.

Guideline: More than one Responsible class

Small (simple) responsibility may only require a single method (for example "create a sale",) where as more complex responsibilities ("store the sale data in a relational database") may require the interaction of many more classes, possibly including a subsystem with a responsibility to "provide access to a relational database" [65]. RDD then includes the idea of collaboration, responsibilities are fulfilled either by a method or with other objects methods [65] (see 12.2).
Larman 65 notes that RDD is a general metaphor for thinking about OO software design, objects are like people with different roles working together to achieve some goal. Each is responsible for part of the tasks and and like objects they operate as a community of collaborating responsible individuals.

12.7.3 GRASP - Design Principles (mini-patterns)

GRASP stands for General Responsibility Assignment Software Patterns (technically GRAS Patterns,) with the name being specifically chosen to emphasise the importance of grasping these principles for the development of software [65]. GRASP is a learning aid which names and describes some basic principles to assign responsibilities to objects, proving a simple first step towards RDD [65]. It also provides a foundation for applying more complex patterns such as the Gang of Four (GoF) patterns which are 'larger' and deal with coordinating higher level behaviour. Some of these (or perhaps all,) will not really be new to you, but you won't have seen them named and their pros and cons discussed. Naming them allows them to be explained and applied [65] leading to a more systematic approach to the delegation of responsibilities (and therefore object behaviour).

When do we assign Responsibilities?

Responsibility are assigned either during design when we draw interaction diagrams and decide where the methods which fulfil these responsibilities go or while coding [65]. The GRASP principles can be applied to guide choices about assigning responsibilities and can therefore be applied during design (while drawing interaction diagrams,) or during coding [65].

Patterns

Experienced software developers build up and deploy a "repertoire of both general principles and idiomatic solutions" (that is a generalised bits of solutions,) which if codified in a appropriate format, i.e. named, with a description of the problem and solution give a pattern [65]. Named patterns provide a way chunking and incorporating that concept in to our understanding and memory [65]. That is if we see something described as a pattern (and named,) we can think of it as a pattern and remember it, perhaps associated with other similar pattern. This also facilitates communication we can talk about using the Abstract Factory Pattern, to solve a problem and other developers instantly know what we're talking about [65].

According to Larman 65 Patterns are usually expressed in terms of a general problem and solution which can be applied in many contexts, there may also be some consideration of the trade-offs to be considered. The pattern will then suggest which responsibilities should be associated with which objects (given their roles in the pattern).

Pattern Name:	**Information Expert**
Problem:	What is the basic principle by which to assign responsibilities to objects.
Solution:	Assign a responsibility to the class that has the information needed to fulfil it.

Table 12.3: An example pattern from Larman 65 using his format.

Clearly the pattern shown in 12.3 is both very general and perhaps obvious, however naming it helps use learn, discuss and deploy [65].

Before looking at some patterns being used in a real scenario we'll note some final points from Larman 54, patterns should not be new ideas (they should be established ones which are being named). That being said I've found a lot of new variations on old patterns, which I suppose is just people taking the time to recognise that there are many versions of something. The GRASP patterns do not state new ideas they name widely used principles, to an expert they should be fundamental and familiar [65]. To you (second year students,) they'll hopefully be thing which you were starting to get to grips with in your object originated coding, or even guidance for problems which are on the horizon in your coding.

12.7.4 Example - Solving A Simple Design Problem using a GRASP Pattern

Rather than start with a full scenario (or use case,) which is where we'd normally start in terms of mechanistic design, I'm going to start with some 'parts' of a larger solution. Seeing these simple patterns deployed in a number of examples will allow us to work up to a full scenario, essentially this will be the detail behind the call to system from our SSD (see 10.3). I've stolen these examples from Larman 65, we'll work though something similar for other examples later.

Question: Who creates the Square object?

Larman 65 provides a very detailed discussion of this example and the case study it comes from, this is a much abbreviated version and I recommend you read the full one. Larman really explains his though process well.

A common (and basic,) problem in OO design is who creates a particular object, this is a 'doing' responsibility which must be assigned to some

object [65]. The example we'll use is from a scenario/use-case for the first elaboration iteration of a Monopoly game, where the game is initially set-up, a single piece is added to the board and moved around the board by dice roles (without player interaction). Larman 66 covers this in some detail but we're only looking at some elements of it anyway.

The question is:

"Who creates the Squares which represent the Monopoly board?"

First lets look back at the object domain model to see what classes we have in addition to square (see 12.2).

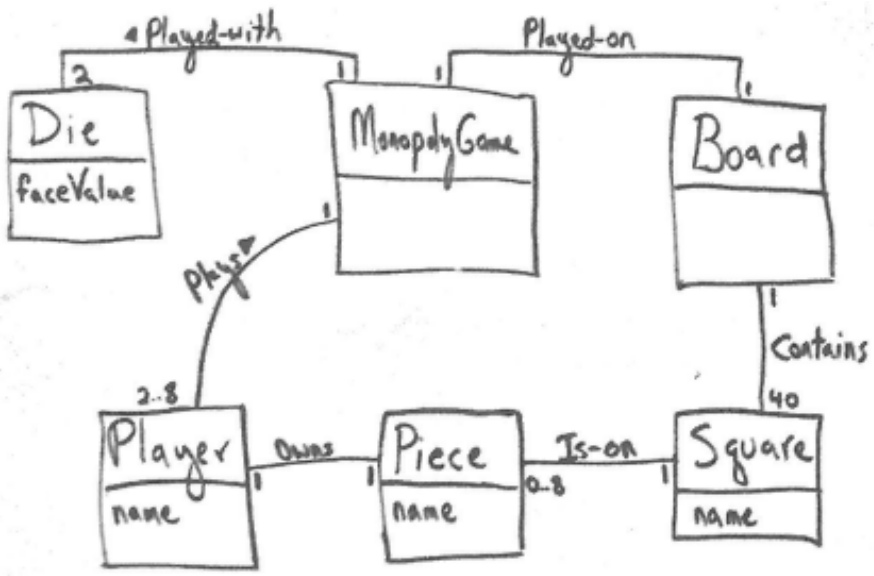


Figure 12.2: The domain model for the first elaboration iteration from Larman 65. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

There are several options according to Larman 65, I've put his discussion of each into a tabular form to aid clarity (see 12.4).

Class Name:	Patterns or Principles	Evaluation
Dog	Lower Representational Gap (LRG)	An arbitrary class could create squares, however this doesn't fit well with our mental model of the problem.
Player	LRG	Conceptually the player is associated with a piece which is associated with a square, again this doesn't fit.
Board	Creator	This is the obvious choice, (and Larman 65 suggests the choice of Software professionals he's encountered world wide), its also in keeping with the Creator pattern.

Table 12.4: Evaulation of the candidates for creating square based on Larman 65.

Note that this is essentially responsibility assignment, we've assigned the responsibility for creating square to Board.

Lets look at the creator pattern (**??**) and then look at the process by which it is used.

Pattern Name:	**Creator**
Problem:	Who Creates A?
Solution: (this an be viewed as advice)	Assign class B the responsibility to create an instance of class A if one or more of these) is true: 1. B "contains" or compositely aggregates A. 2. B records A. 3. B closely uses A. 4. B has the initializing data for A.

Table 12.5: The Creator Pattern Larman 65 using the less detailed format.

Aside: See the sections on Aggregation 12.8.2 and Structured Classes **??**, both are concepts we'll be coming onto (but are not for right now).

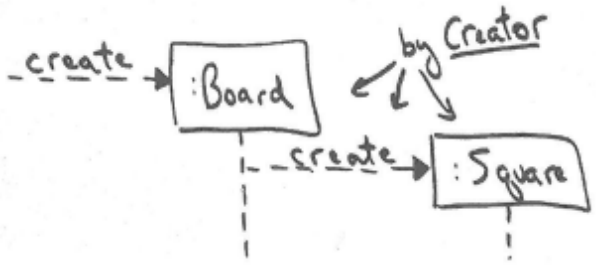
Software Objects and the Design Class Diagram - DCD .
Before applying the pattern we have to understand that we're now working with software objects 8.5.3 and not domain objects [65]. These will later be shown in a DCD (Design Class Diagram,) but where do they come from? We'll work though the process of applying the creator pattern and hopefully it will become clear.

This is the process described in Larman 65, in a simpler format with more descriptive text.

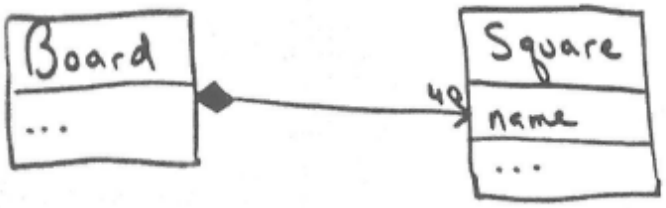
Step	In this example.
1. Identify and describe the problem:	*Who creates square?*
2. Decide on a pattern to solve the problem:	*We've chosen **Creator**.*
3. Look for existing software objects to satisfy the roles of A and B in the pattern: • **iteration *n*:** Look for existing software objects in our DCD. • **iteration 1:** We are just starting the design, using LRG (lower representational gap) identify appropriate domain classes (from the domain model,) and promote them to software classes. Add these to the DCD.	*Board and Square are made software objects, and Board (B) is chosen to be the creator of Square (A) in our example.*

We now have to consider the relation between these two objects. Larman 65 suggests that as board will create squares, they will be part of the board and when board is destroyed it will manage their destruction they should be related by composition. A quick discussion of composition can be found here 12.8.2.

Agile modelling suggests that both the dynamic (behavioural - sequence,) and static (structural - DCD) should be worked on in parallel [65]. Larmans images (see **??**) are cleaned up whiteboard images, these early stages really need whole team involvement.



Applying the Creator pattern in a dynamic model.



In a DCD of the Design Model, *Board* has a composite aggregation association with *Squares*. We are applying Creator in a static model.

Figure 12.3: The creator pattern being appiled and the start of the DCD from Larman 65. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

12.8 More Class Diagrams

In a previous section we looked at the use of raw UML to describe classes at different levels 8.5.3, we're now looking at design level class diagrams and the classes in our diagrams are moving from conceptual to design. We'll recap a little of this, before going on to consider more UML syntax pertinent to this stage.
The UML notation provides the facility to include more detail in a class diagram than shown in the conceptual classes of the domain diagram. Larman 61 provides a single diagram which shows most of the syntax, shown in **??**.

Constructors

It is normal for the constructor to be omitted, if shown it may be prefixed with a stereotype «constructor »[61].

Operations to Access Attributes

Access operations such as *getX* and *setX* are normally omitted (or filtered - in sophisticated tools) as they clutter the diagrams without adding anything interesting [61]. An exception here is any interesting attribute changing methods such e.g. *credit/debit/getBallence* for an Account object. This is perhaps essential when modelling on a whiteboard as these waste space/time an ellipsis ... can be added to show there other attributes/methods are known about but not shown [58].

12.8.1 Associations Between Classes

Much of the syntax has been covered before (when considering conceptual class models,) but we need to take this a little further to consider more sophisticated associations.

Representing Associations

Associations are relationships between instances of classes indicating that some meaningful and interested connection exists between them [56]. We'll consider associations and attributes in more detail next week, however as object diagrams without them are often incorrect we need to know a bit about them up front.
According to Larman 56 associations should be shown when one object needs to show that knowledge of a relationship needs to be preserved for some length of time (milliseconds or years). As is hopefully clear from the section above (**??**) many of these relations emerge from the application of basic principles and patterns and can be added to the diagram as these are deployed.
Representing associations is reasonably simple, we use a line with an association name [56] (see 12.4). At the end of each line we have a number or range representing multiplicity (number of) for instances of classes [56]. The association is inherently bidirectional and can be logically traversed in either direction, the association may have a reading direction arrow which suggests how the association name should be read - this has no meaning in the model just helps the reader [56].

Now we begin to consider these relations in terms of design and include restrictions on navigation, multiplicity, names for association end points (roles) within the classes.

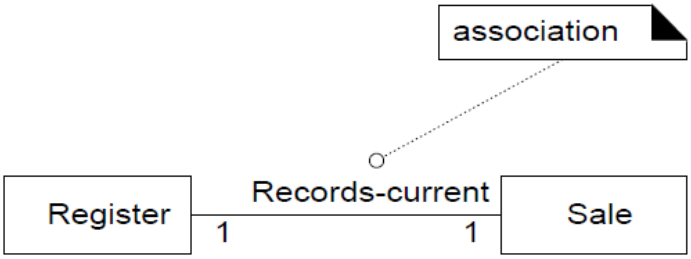


Figure 12.4: From Larman 56, a simple association with unit multiplicity.

Syntax: Associations between Classes

The diagram 12.5 shows a connection between two objects in both the domain diagram form and design class diagram. The top one shows directed association name (see 12.5) and multiplicity at each end (see 12.6). It is more common to see the role names at this stage rather than the relation name, this representation is show in the lower half of 12.5

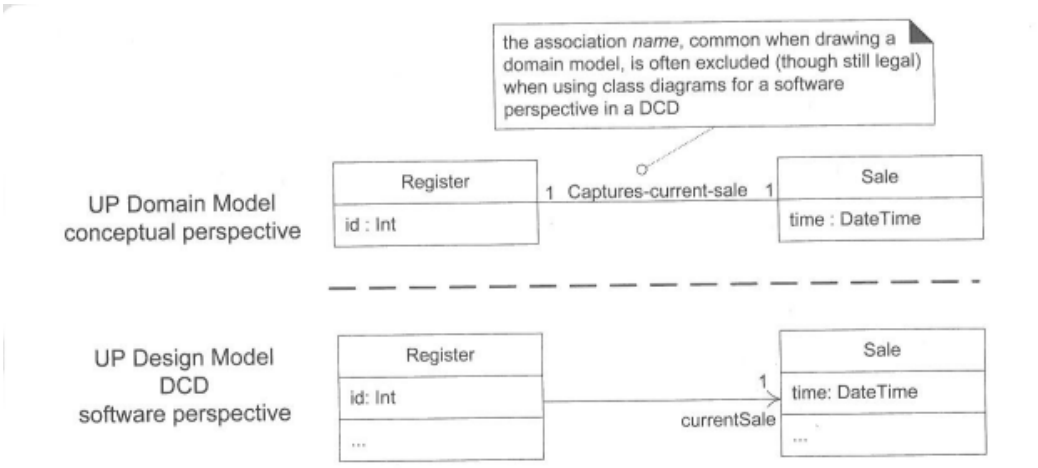


Figure 12.5: An example from Larman 56, this shows the difference in associations between domain object diagram and design class diagram.

Naming Associations

Simple names such as "has" and "uses" are a poor choice as they do not enhance our understanding of the domain [56]. In the example 9.2 "has" could have been chosen at as the association name but would have added nothing to our understanding, however "Lives-at" tells us its their home address. Lar-

man 56 suggests using a *ClassName*-***VerbPhrase***-*ClassName* format which is both informative and can be read. While his examples use a hyphenated format, camel case is also acceptable.

Roles & Multiplicity

The end of each association has a role and can have a name, multiplicity expression and navigability [56].

Multiplicity :

Multiplicity simply defines how many of each class are involved in an association at one time [56]. That is how many instances can be connected, we can improve / extend our earlier example (person-address) by considering that more than one person can live at the same address. It may be that a person lives at a number of addresses over the course of their lives, but for this system they have only one current address. The examples shown in 12.6 are reasonably representative.

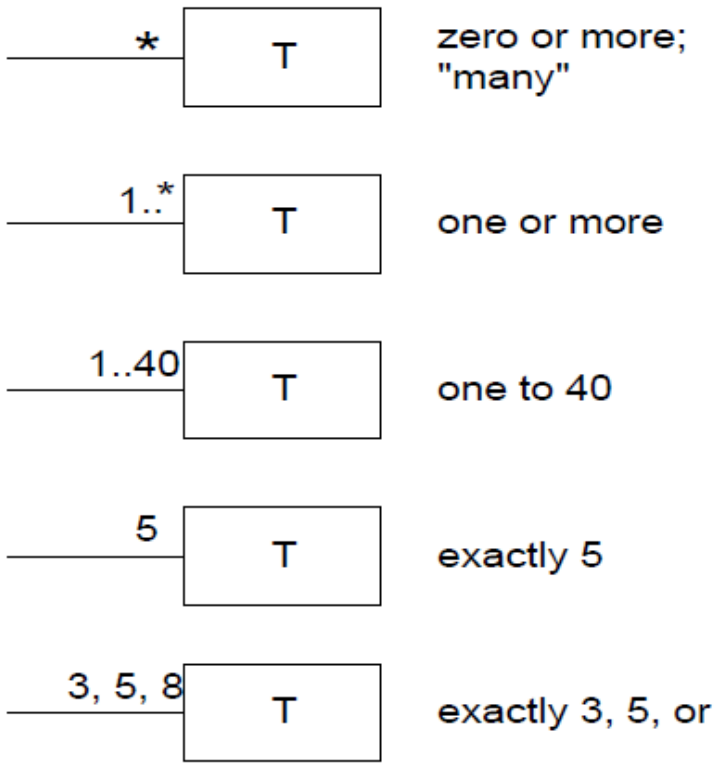


Figure 12.6: A home made example based on those in Larman 56, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image. I've had to screen shot this one to get it to work.

The multiplicity of a relationship is always based on the needs of the scenario, [56] cites Rumbaugh's example where a Person is associated with a Company. For tax purposes this will probably be a list of companies (i.e. all those who the person currently works for) for a Union probably only one.

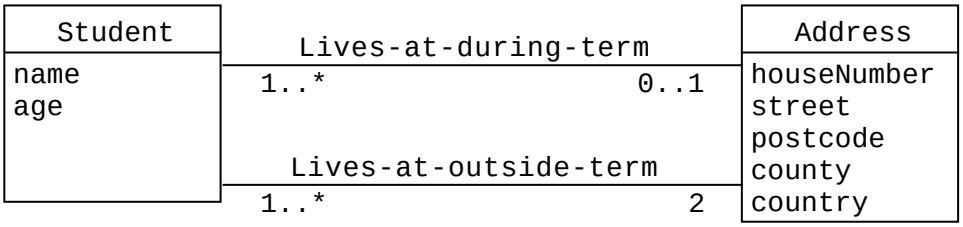


Figure 12.7: A home made example based on those in Larman 56, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image, I've not screen-shot this and its lost its arrows.

Roles :

We can have multiple many to many relations [56], this is perhaps best illustrated though an example see 12.7. Here each student has two addresses a term time address (which may or may not be present,) and a home address. Both of these are address objects, note that one or more students can live at the address in each case.

Association Syntax - Attribute Text and Association Lines

The role name is sometimes used on the association line itself, of may be omitted an included in the list of attributes or the line omitted and the attribute shown with a class type [61]. See the diagram 12.8

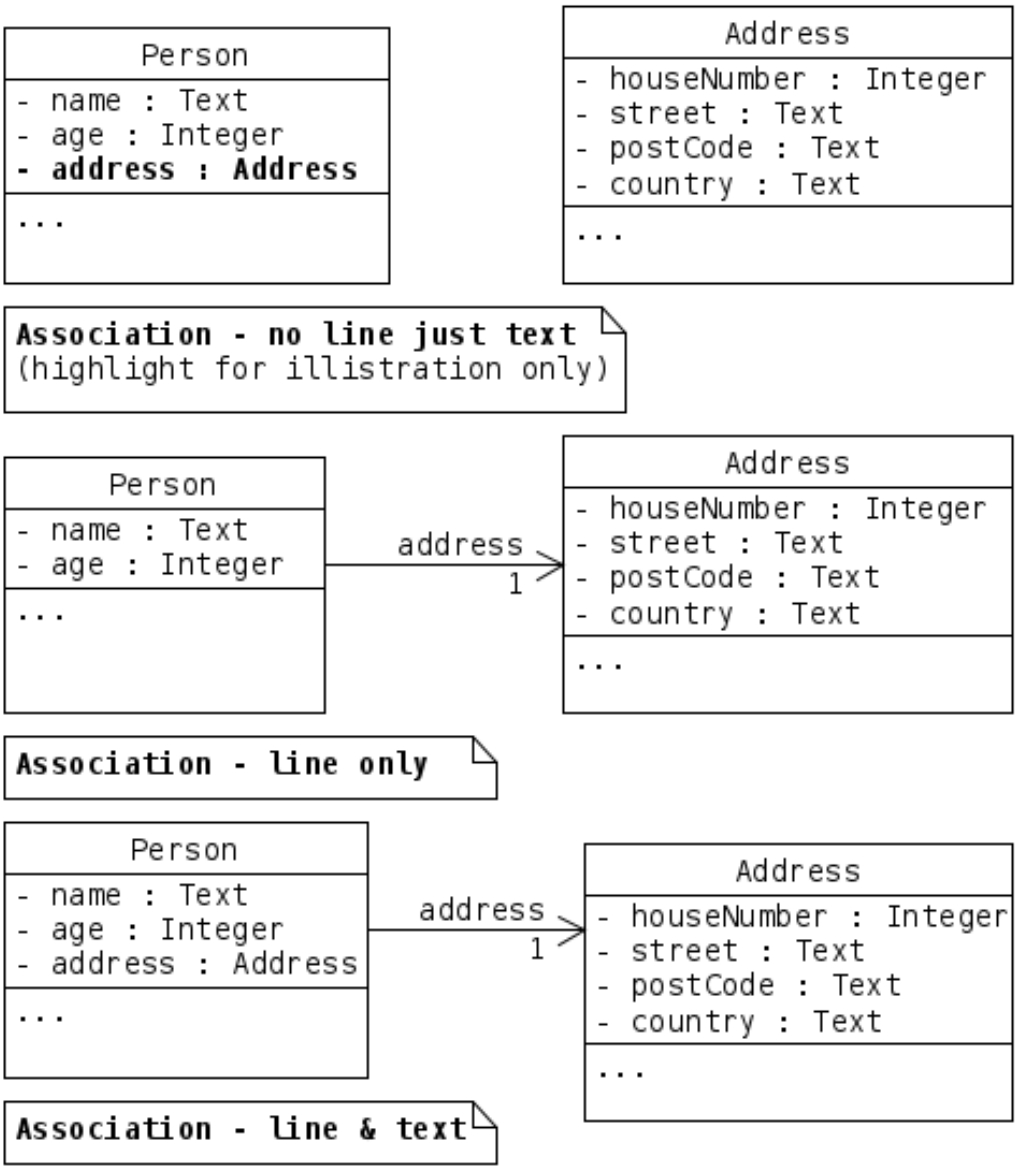


Figure 12.8: Ways in which we can show role names and associations, based on Larman 61.

Larman 58 provides the following guidance:

Type	Guidance	Example
Association Line	The visual emphasis is important, i.e. this needs to be seen. The relationship may be an important one in terms of domain logic.	*Register to Sale, Person to Address etc.*
Attribute Text	The relation would add clutter to the diagram, the relation is an obvious one. The relationship is not important in the context of the domain perhaps to a utility or support class.	*Sale to DateTime, Person to Text etc.*
Both	This is usually what tools and lecturers trying to make a point to their students do.	*Last years Introduction to Programming coursework*

We can go further and include details of multiplicity and property strings (special keywords such as ordered which tell us about the collection of objects) [61], see **??**. I've included this as its something we might want to use in the coursework, I'd not be to worried at this stage however if you omitted the property string.

12.8.2 Specialised Associations

Association is quite a general relationship between objects, we can think of this as two objects which are connected but can exist independently. Where one object is part of or owned and controlled by another we can use UML to show that this is the case.

Composition

Composition (also known as composite aggregation) is a strong kind of whole part aggregation and the most useful [61]. The implications of the relation are [58]:

1. That an instance of the part (e.g. Square) belongs to only one composite instance (e.g. Board) at a time.

2. That the part must always belong to an instance (though it may move from instance to instance).

3. That the composite part (whole) is either directly responsible for creating, managing and destroying the parts or collaborates with others to do so.

Where a class orchestrates the behaviour of a number of internal parts it becomes a structured class for which there is an alternate syntax (we'll come to these later **??**.

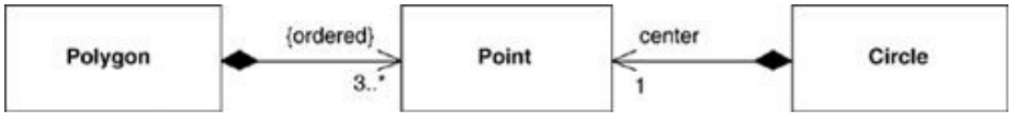


Figure 12.9: Associations (directed with multiplicity) from Larman 61.

One point noted by Fowler 43 is than the association is usually has a multiplicity of one at the part end. The syntax is filled diamond arrow head which points to the whole of the whole-part relation, this is shown in the figure 12.9.

Aggregation

Aggregation is a weaker form of Composition, it loosely suggests a whole part relation but its unclear how this differs from a more traditional associations [58]. I've always interpreted it as a part which cannot be used alone but may exist to be used by other objects. I've always taught this to my students (using the example of a car and a tow-ball,) however both Larman 58 and Fowler 43 quote Rumbaugh (one of the original creators) who suggested it exists because people though it should be there. Both authors go on to say that its probably best to stick with association and composition (or composite aggregation as Larman calls it). Its represented as a unfilled diamond but otherwise identical to composition.

12.8.3 UML Recap - Sequence Diagrams

This is a recap of 10.4 I didn't need to add anything here, so I've not bothered.

12.8.4 Participants and Messages

Sequence diagrams are one of two kinds of interaction diagram provided by UML (the other being communication diagrams,) [58], which we're going to leave to a future iteration. Sequence diagrams are more commonly used than

interaction diagrams as they emphasise sequence over structure, making it easy to find the next message (with time going down the page) [30]. A communication diagram by contrast is simply an object diagram (like a class diagram but showing individual objects,) which shows the messages passed during an interaction [30].

Sequence diagrams are the more notationally rich [58] and hence our first choice, 12.10 is based on Larman 58 but with some additional annotations of my own.

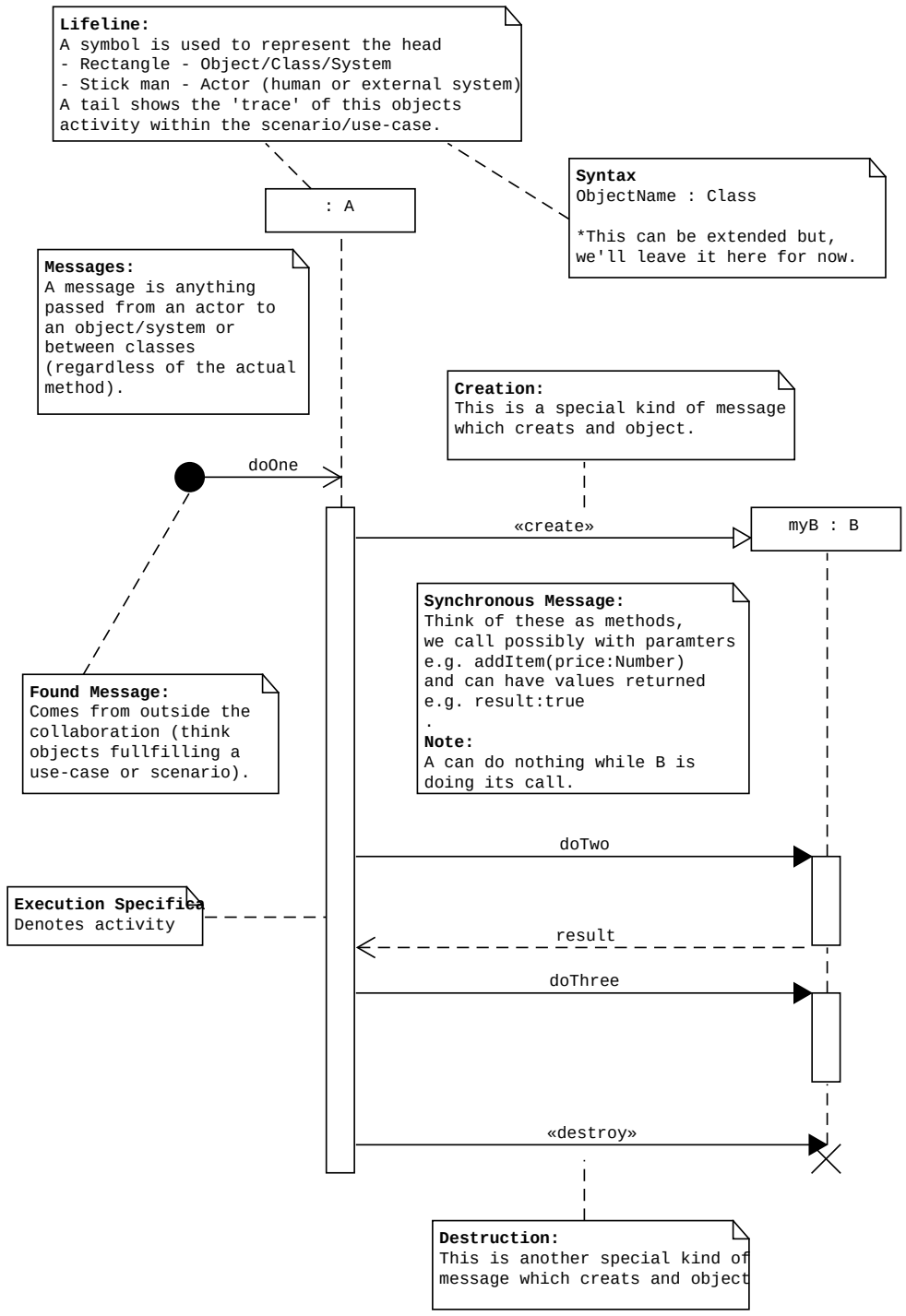


Figure 12.10: Based on Larman 57, augmented with my own annotations inspired by Fowler 42 and Douglass 30

The key elements (in 10.1) shown are:

- **Participant** - This is shown in the rectangle at the top of the lifeline, it has usually has the format **objectName:ClassName** however this can be extended to it can be used to describe multiple objects etc. objectName[n]:ClassName. The UML specification [88] provides more detail (though in a less than readable format). Common participants are show in 10.2.

- **Synchronous Message** - The sequence diagram syntax allows us to show both asynchronous and synchronous messages. The diagram above shows the latter, the best way to think of these is as one object calling a method on another, perhaps with arguments and returns (see 10.4.1 for more examples and discussion.

- **Create and Destroy Messages** These are special kinds of messages which have a stereotype associated with them i.e. «create» and «destroy». These can be though of as calling a notional constructor and destructor for the object. These are not normally shown in System Sequence diagrams.

- **Found & Lost Messages** - These are messages which leave the boundary of the current sequence diagram. Sequence diagrams are normally used to describe an interaction between classes (this could be a scenario, single use case, part of a use-case etc.) where messages enter or leave context they are represented as lost and found messages.

- **Execution Specification** - Processing within a method is often shown as a block on the lifeline, these are often used to reinforce the difference between synchronous and asynchronous messages. These are more commonly seen in diagrams created with UML tools and known as the Activation Bar in UML 1.0 [58].

NOTE: The syntax of sequence diagrams has changed over the various versions of UML, there are also a number of tools, books, authors who appear to impose their own limitations or add their own flourishes. Please make every effort to use the correct syntax, if your in any doubt check the notes. If you're writing your diagrams up in a tool and it is unable to generate the correct syntax, put a note to explain the issue. As with the other diagrams I'm not looking for perfect syntax (the logic is more important,) however there is a point at which it becomes just lines and boxes and is no longer UML.

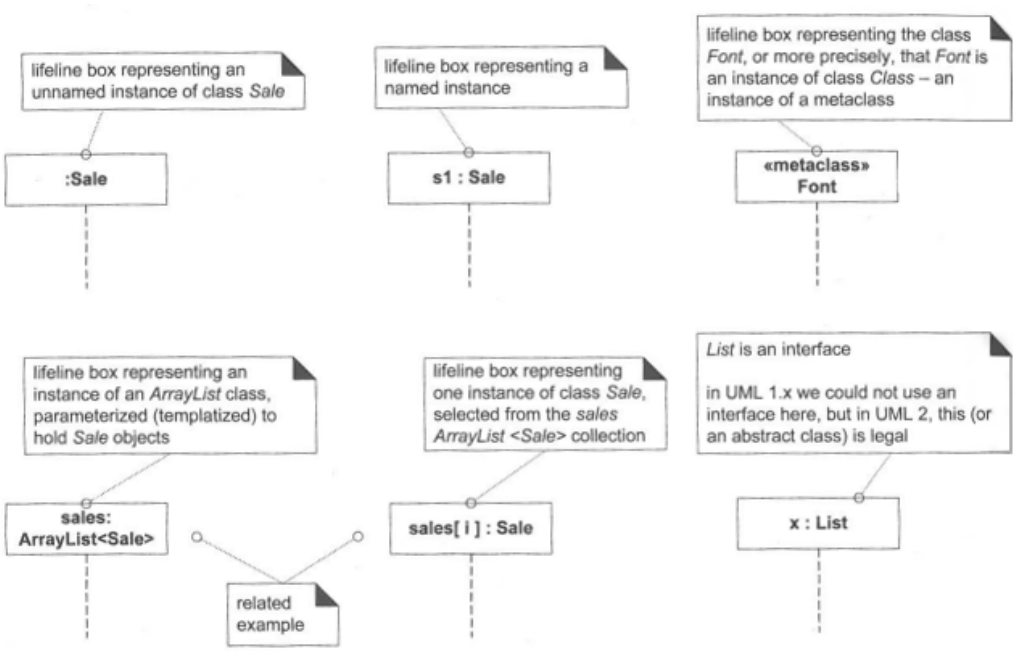


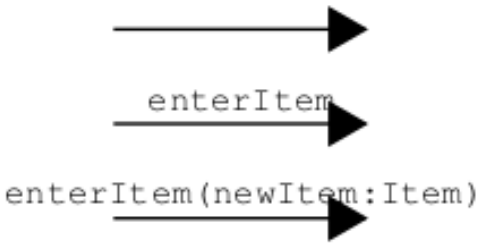
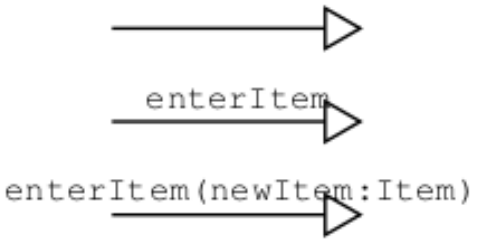
Figure 15.5 Lifeline boxes to show participants in interactions.

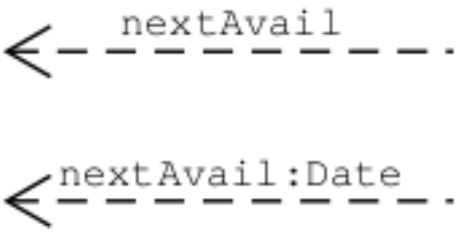
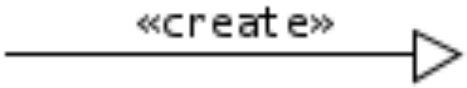
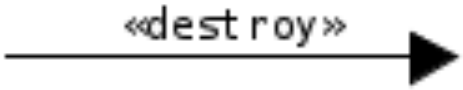
Figure 12.11: From Larman 58.

Larman 58 provides some excellent documentation on the Sequence diagram including the flowing examples of the kind of classes which may be involved at the design phase.

Messages

As mentioned above there are a number of different kinds of message which can pass between instances (or classes) in a sequence diagram, these are:

Type	Description	Symbol
Asynchronous	A message is send but the sender does not wait for a response, e.g. an event is sent to an event queue.	
Synchronous	A message is send and the sender waits until the processing is complete in the receiver before continuing. There may be a reply message involved 12.8.4	

Replies	These are messages returned as the result of a call on another object, you can think of them as method returns. There is an alternative syntax (Larman 58 provides an example, we'll stick with one way to avoid confusion.	
«create»	A special kind of synchronous message resulting in the creation of an object.	
«destroy»	A special kind of synchronous message resulting in the destruction of an object.	

Asynchronous communication is most common with real-time systems where objects have their own threads of execution, e.g. in multi-process or multi-threaded design where there is more than one active object.

All messages have the full form

return = messageName(paramter : paramterType) : return type

However it is not uncommon to see, calls for specific programming languages used here [58] this is common where the diagram has been generated from code, or is being used to generate code Model Driven Development (MDD). Larman 58 also suggests that empty parameter lists and void returns are often omitted, type information may also be omitted if obvious [58]. I'm not sure the latter is a good idea!

12.8.5 Frame Operators

In the earlier section we breifly consdiered Frame Operators here' we'll recap these there are still some we're not going to use right away, but in the next few classes we'll be introducing them as requried (probably along with more complex patterns). The following table is based on Larman 58 and Douglass 30.

Operator	Description
sd	Names an interaction fragment. This could be a system sequence diagram, a realisation of a use case (i.e. how objects from our class diagram will interact to get the job done,) a part of a sequence diagram we want to re-use in other diagrams. **Not covered here in detail, we don't need them until later see ??**.
ref	References another interaction fragment, i.e. where we have parts we wish to re-use (e.g. authenticateUser) we can refer to them with this. **Not covered here in detail, we don't need them until later see ??**.
alt	This provides a number of alternatives, one of which will be taken based on the guard conditions (in [ ]) an else guard is also provided which evaluates to TRUE if all others are false. This is a complex fragment [30] and is used in UML to express both if/else and switch/case type programming structures. **Note:** Larman 58 suggests the guard should be placed over the life-line of the participant who performs the test. See figure 12.12.
opt	A kind of simplified alt, a fragment which will only be executed if its guard is true. See figure 12.14.
loop	Specifies that a fragment should be repeated until some condition (expressed in a guard) becomes FALSE. There is no official **for** syntax though it is suggested one is being considered [58]. See figure 12.13.
par	This is similar in syntax to alt however, here all compartments execute rather than one. **Not covered here in detail, we don't need them until later see ??**.
region	Identifies a region which is atomic, and cannot be interleaved with others, i.e. the actions in this fragment must take place in this order with no possibility of interruption. **Not covered here in detail, we don't need them until later see ??**

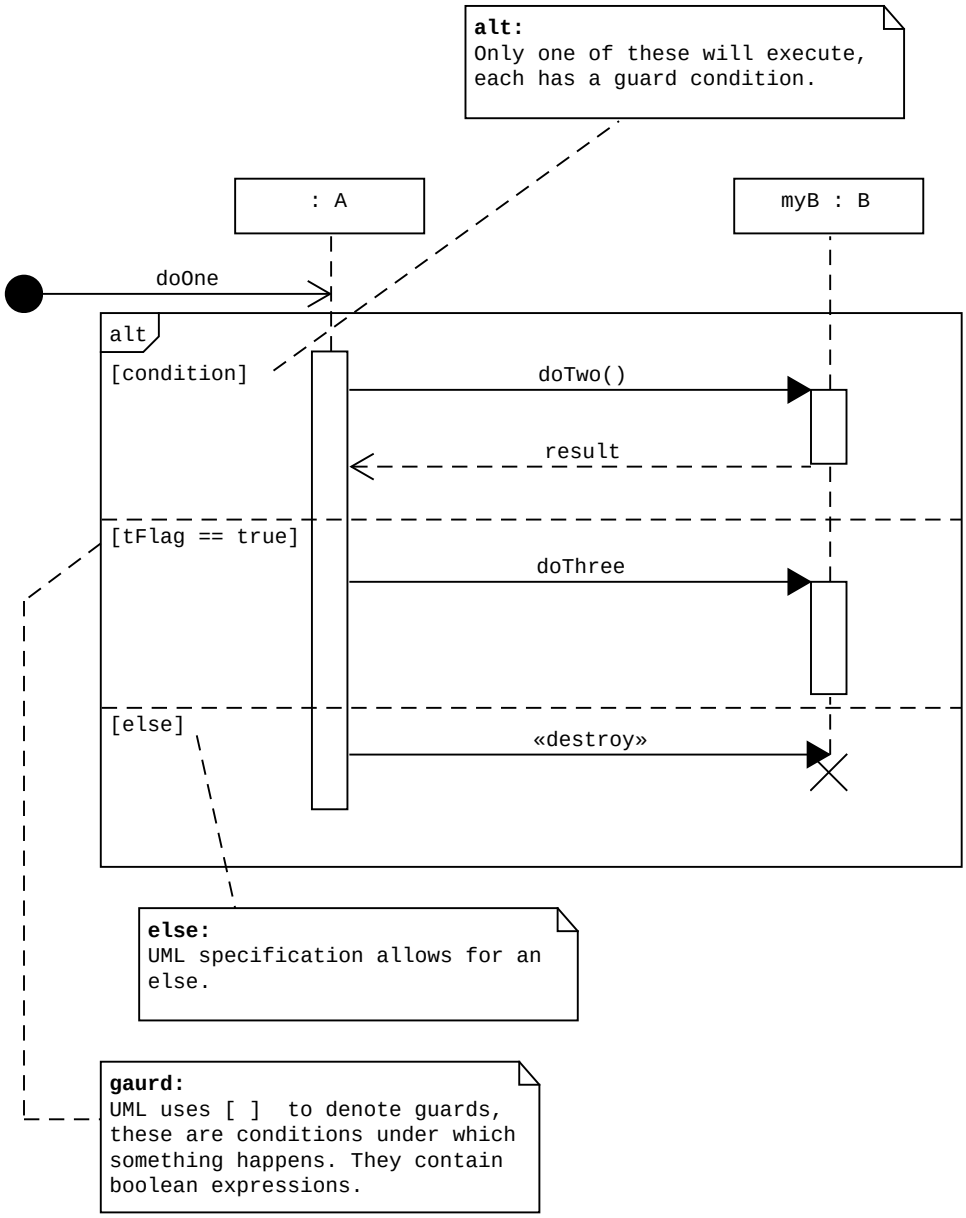


Figure 12.12: Syntax for the alt interaction fragment, this is used for both if/else and switch case type structures.

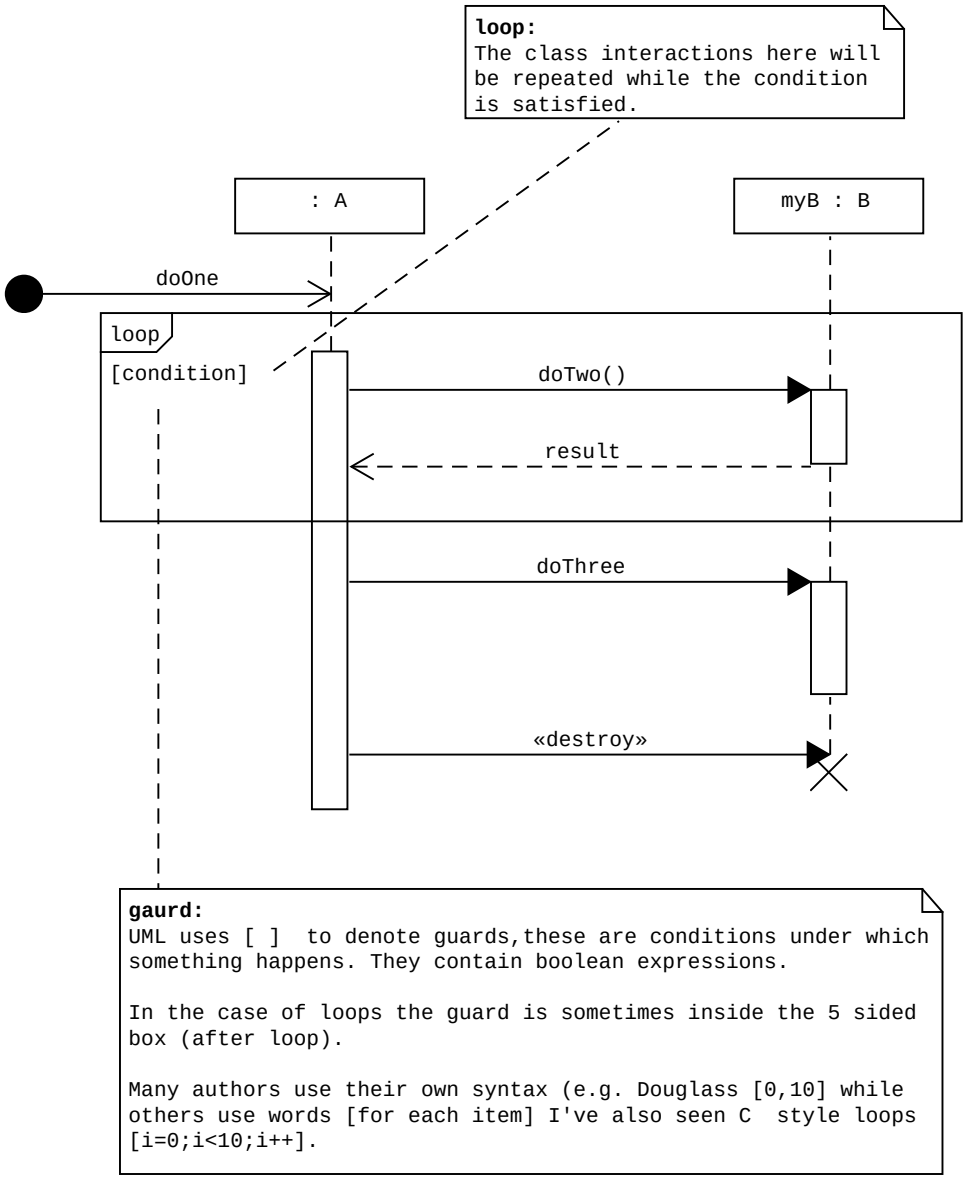


Figure 12.13: Syntax for the loop interaction fragment, this is used for both while and switch for structures.

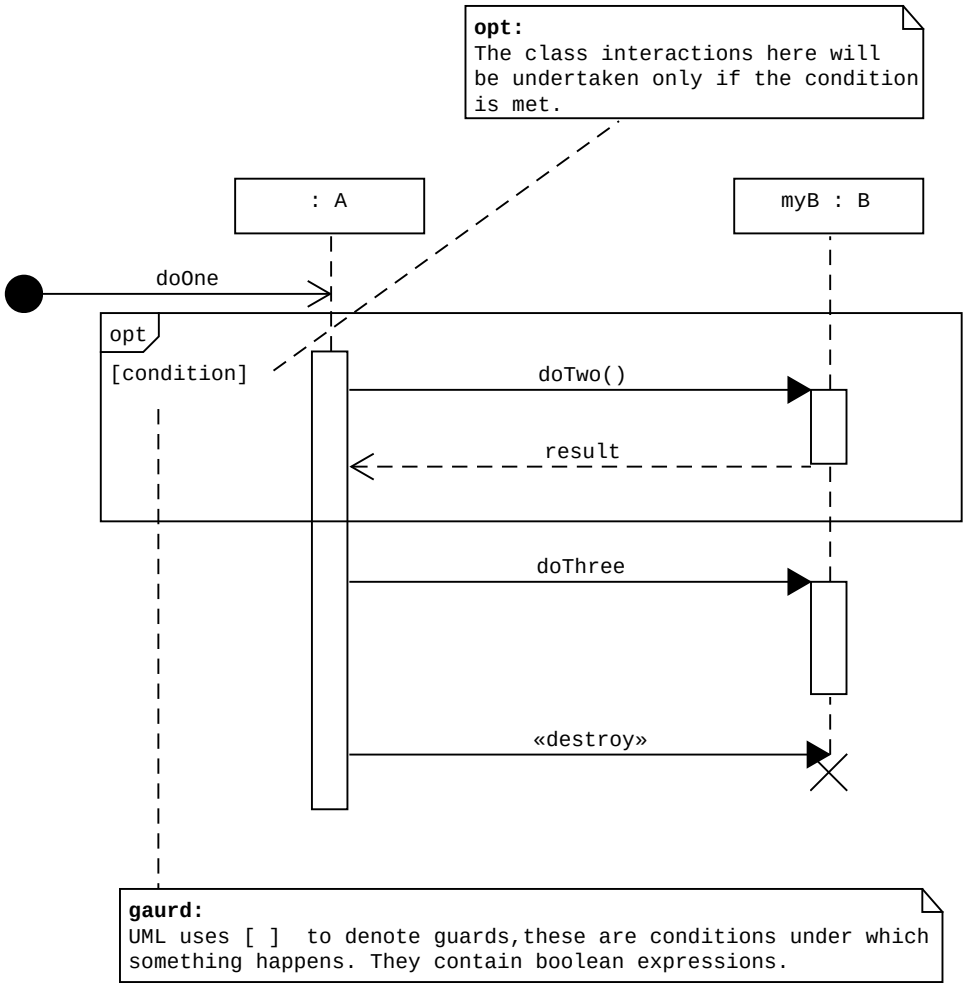


Figure 12.14: Syntax for the opt interaction fragment.

12.9 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 12.9.1, and consider the use case 12.9.1 and 11.9.3 then use an appropriate pattern to identify:
    - Who creates Unit?
    - Who creates Format?

2. Read the case study **??** the DCD provided 12.9.2 then use an appropriate pattern to identify:

    - Who creates InputPattern?
    - Who creates OutputPattern?

Questions to think about (or what kind of exam question could I get on this).

- Explain the Model-View Separation Principle and why such an approach would be used in the development of a commercial video game.

12.9.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the

default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

Use Case

The team have produced a number of use case summarised at the highest level by the following use case diagram. The names of the use case here may differ slightly from those we identified during the tutorial.

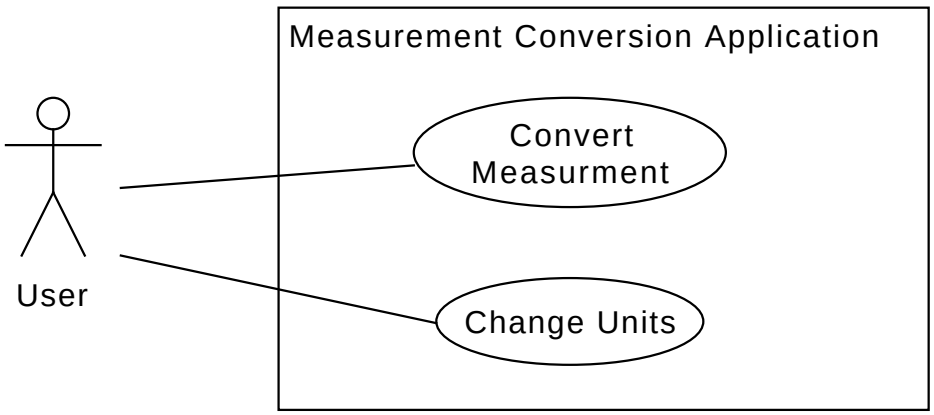


Figure 12.15: This is an incomplete use case based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough.

Use Case: Convert Measurement

UC3: Convert Measurement L12	
Scope:	Measurement Conversion Application
Primary Actor:	User
Level:	user-goal

Stakeholders and Interests:

- User: Wants to convert between imperial and metric measurements of length.

Preconditions:

- Application has launched and the interface is displayed
- Interface is prepared for user input

Postconditions (success guarantee):

- Output is presented in a suitable format

Main Success Scenario:

1. User enters a measurement in the default unit (imperial,)
2. User requests a conversion (indicating they have completed their data entry).
3. System performs the conversion
4. System displays the converted measurement

Extensions (Alternative Flows):

- 1a. User changes units before entering a measurement.
- 2a. User enters nothing
    1. System does not attempt to convert
    2. System highlights the lack of input (e.g. change background of input field to red)
- 2b. User enters an incorrectly formatted measurement for the current conversion.
    1. System Displays an error message
    2. System highlights the incorrect input (e.g. change background of input field to red)
    3. User enters a measurement

Special Requirements:

- Must be possible to enter the measurement in a number of formats, e.g. 1'6", 1.5", 18" etc.
- The output must be such that the user can easily copy the output for use in another application.

Technology and Data Variations List:

1. ...

Frequency of Occurrence:	Expect the user to perform one or more conversions in a single session

---

Open Issues:

- Given that the input can be in a number of formats, which the system is to deduce. How will this be done?
- The output may be required in a specific format, should this option be given to the user and how? This could be a property set though a menu (with a default,) or some means of selection near the output.

Domain Object Model

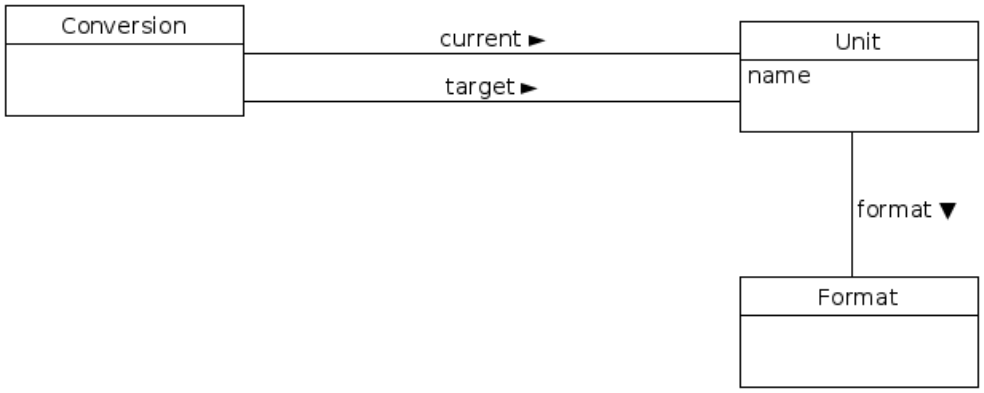


Figure 12.16: This is a domain object model based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough. I've avoided describing this as a draft (suggesting its incomplete,) or using any other term which would suggest it is final. In the spirit of Agile Modelling we are using it to inform sketches of the design so we can discuss and evaluate design options. Changes may be made as a result of this discussion and its probably that future iterations would contain some of the conceptual classes.

12.9.2 Case Study 2

A researcher at the university is looking to write a paper with a colleague they have both datasets and a new neural network to train on them. In order to do this they need to load in files from the original project so that their Neural Network will be trained and tested with the same files. Each file consists of comments (which start with a #,) some meta data (information about the file content) and the data itself (see example file), all in a CSV format. The data is in the form of input and output patterns, each contains t (a time offset). The input contains the values for parameters t, r, g, and b while the output patter has two values t and o. Due to the nature of neural network operations all parameters are floating point numbers between -1.0f and 1.0f further validation is possible as the t (which represent the time-step,) should be identical in the input and output pattern.

DCD for Case Study 2

We should be starting from a domain object diagram, but I don't have one.

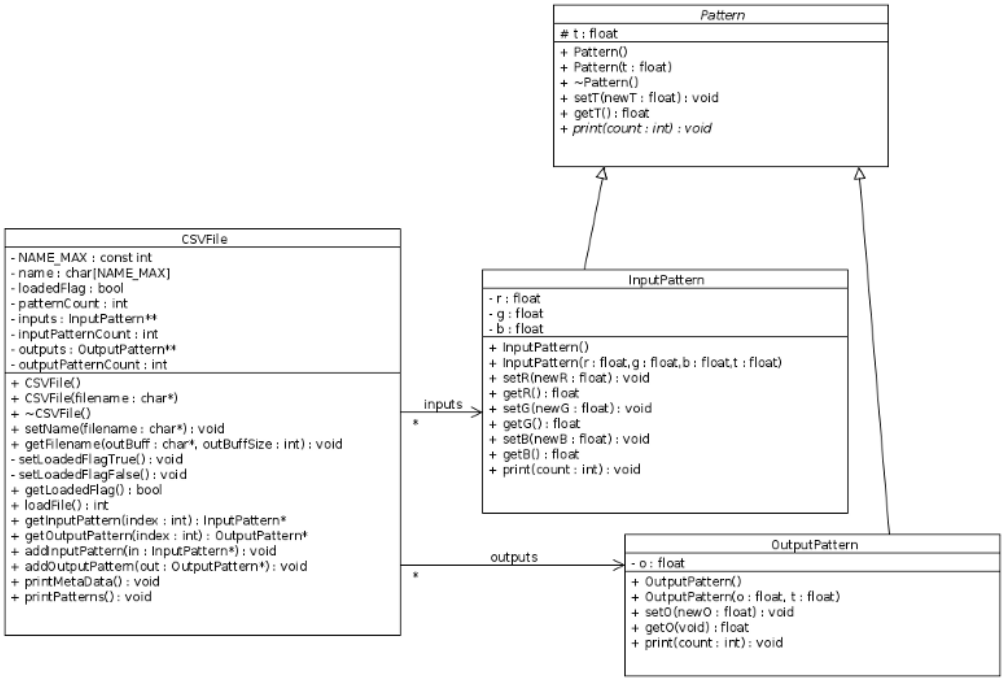


Figure 12.17: This is a the closest thing to a domain model I had to hand, hopefully it will do to give us more practicein the deployment of the creator pattern.

12.10 Summary

The next important part of the design process is the Realisation of the use case, which is the mapping of use case and individual scenarios to object behaviour. We've started small considering parts of these interactions in isolation and the low level patterns (or OO principles,) we've looked at one of these patterns so far and there are quite a few more to go!
The GRASP patterns are low level simple design patterns/principles named and documented to make them easier to remember, discuss and use. This week we've looked at Creator with some examples of its use, however we've not considered the pattern in full (which we'll do in a later class) at the moment we need to see a few more patterns so we can start using them in the realisation of use case.

12.11 Preparation for Next Week

12.11.1 Essential Reading

An introduction to GRASP and Patterns, this covers this week and next weeks class (maybe the week after).
Larman (2014q). *Applying UML and Patterns*, chapter GRASP: Designing Objects with Responsiblities, pages 312–362. Prentice Hall PTR.

12.11.2 Background Reading

In conjunction with the above you may need to refer to these chapters which provide very detailed coverage of Class and Sequence diagrams. Larman (2014j). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

Larman (2014m). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Designing with Patterns/Principles: Mechanistic Design - Part 2

13.1 Introduction

In the previous notes (12) we'd started object design and were looking to begin use case realisation. In an attempt to systematically design object orientation systems adopted a responsibility driven design approach (RDD). This suggested the use of GRASP patterns in order to assign low level responsibilities and we'd looked at one of these patterns. We'll recap some of these concepts

13.2 Recap: Use Case Realisation

Use Case Realisation or perhaps more accurately scenario realisation is the process by which we describe the design (in terms of objects,) or one or more use case [64]. The term Use Case Realisation comes from UP and is used to reinforce the connection between the use case in analysis and the design of the system [64].
Larman 64 argues that object orientated design is simply a question of assigning responsibilities to objects and designing collaborations and that both can be logically rationally explained and learned. Also that mastering OO design requires the deployment of a large number of soft principles (with many degrees of freedom,) this seems daunting but these principles can be named and learned.

13.2.1 Orientation: What's next?

The plan is as follows:

- **GRASP** We'll continue looking at the simple object orientated design patterns. These provided guidance for solving really small design problems, e.g. Who should create object X?
- **GoF Patterns** We'll the move on to look at some (or perhaps all) of the Gang of Four patterns, these are well know and well used design patterns which will give us a flavour for their use.
- **More Class Diagrams and Sequence Diagrams** Along side the principles and patterns we'll be looking at more UML, which we'll need to visualise the patterns - though we'll build some in C++ as we go!

Essentially we're working up to realisation, starting small with the GRASP - small patterns (or principles,) these should be enough to get us started.

13.3 Recap: Responsibilities and Responsibility Driven Design (RRD)

In RRD objects are attributed responsibilities, in terms of what they are responsible for knowing and what they are responsible for doing [65].
There are two types of responsibility knowing and doing [65]:

- **Doing** - Responsibilities may include:
  - Doing something itself, e.g. a calculation, creating and object etc.
  - Causing something to happen in another object, or changing its state.
  - Controlling and coordinating activities in other objects (e.g. **??**)
- **Knowing** - Responsibilities may include:
  - Holding data itself in private attributes
  - Knowing about related objects
  - Knowing about data it can derive or calculate

13.4 Responsibilities and Responsibility Driven Design (RRD)

13.4.1 Recap: What is RRD?

GRASP is a learning aid which names and describes some basic principles to assign responsibilities to objects, proving a simple first step towards RDD [65]. We're using it to provide a foundation for applying more complex patterns such as the Gang of Four (GoF) patterns which are 'larger' and deal with coordinating higher level behaviour.

Some of these (or perhaps all,) will not really be new to you, but you won't have seen them named and their pros and cons discussed. Naming them allows them to be explained and applied [65] leading to a more systematic approach to the delegation of responsibilities (and therefore object behaviour). We're going to come back to these and discuss them in more detail at a later stage, without a name we couldn't do this!

Recap: When do we assign Responsibilities?

Responsibility are assigned either during design when we draw interaction diagrams and decide where the methods which fulfil these responsibilities go or while coding [65]. The GRASP principles can be applied to guide choices about assigning responsibilities and can therefore be applied during design (while drawing interaction diagrams,) or during coding [65].

Patterns

Experienced software developers build up and deploy a "repertoire of both general principles and idiomatic solutions" (that is a generalised bits of solutions,) which if codified in a appropriate format, i.e. named, with a description of the problem and solution give a pattern [65].

According to Larman 65 named patterns provide:

- **A Memory/Learning Aid** It helps 'chunking' and incorporating that concept in to our understanding and memory. That is if we see something described as a pattern (and named,) we can think of it as a pattern and remember it, perhaps associated with other similar pattern.

- **A Communication Aid** This also facilitates communication we can talk about using the Abstract Factory Pattern, to solve a problem and other developers instantly know what we're talking about

Patterns are usually expressed in terms of a general problem and solution which can be applied in many contexts, there may also be some consideration of the trade-offs to be considered [65]. The pattern will then suggest which responsibilities should be associated with which objects (given their roles in the pattern) [65]. The GRASP patterns (or mini patterns,) do not state new ideas they name widely used principles, to an expert they should be fundamental and familiar [65]. To you (second year students,) they'll hopefully be thing which you were starting to get to grips with in your object originated coding, or even guidance for problems which are on the horizon in your coding.

13.4.2 Solving A Simple Design Problems using a GRASP Patterns - Continued

Rather than start with a full scenario (or use case,) which is where we'd normally start in terms of mechanistic design, I'm going to start with some 'parts' of a larger solution. Seeing these simple patterns deployed in a number of examples will allow us to work up to a full scenario, essentially this will be the detail behind the call to system from our SSD (see 10.3). I've stolen these examples from Larman 65, we'll work though something similar for other examples later.

Information Expert (or Expert)

Question: Who Knows about the Square object, given a key?

Larman 65 provides a very detailed discussion of this example and the case study it comes from, this is a much abbreviated version and I recommend you read the full one. Larman really explains his though process well.

Another common problem is working out who should *know* something, this is a 'knowing' responsibility which must be assigned to some object [65]. Again the example we'll use is from a scenario/use-case for the first elaboration iteration of a Monopoly game, where the game is initially set-up, a single piece is added to the board and moved around the board by dice roles (without player interaction). Larman 66 covers this in some detail but we're only looking at some elements of it anyway.

The question is:

"Who knows about Square given a key?"

First lets look back at the object domain model to see what classes we have in addition to square (see 14.4).

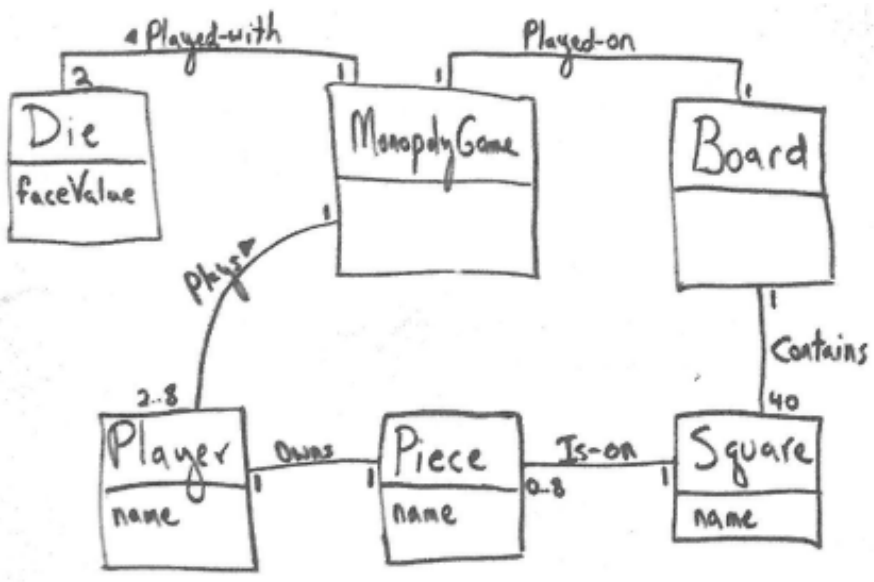


Figure 13.1: The domain model for the first elaboration iteration from Larman 65. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

There are several options according to Larman 65, I've put his discussion of each into a tabular form to aid clarity (see 12.4).

Class Name:	Patterns or Principles	Evaluation
Dog	• Lower Representational Gap (LRG) • Low Coupling (another pattern see **??**)	• An arbitrary class be responsible for knowing create squares, however this doesn't fit well with our mental model of the problem. • Informally coupling refers to a measure of connectedness between elements, minimising this reduces the impact of change on one class on others.
Board	Information Expert	This is the obvious choice (and Larman 65 suggests that of most OO developers,) for an object to find an present one square (given a name,) it must know (have the information) about all the Squares.
Piece	Information Expert	This may fit conceptually, the piece exists in our domain. However in terms of having the information required it is hopefully clear that while piece knows its own square it doesn't know any others. Therefore it cannot fulfil the responsibility.

Table 13.1: Evaulation of the candidates for creating square based on Larman 65.

Note that this is essentially responsibility assignment, we've assigned the responsibility for finding square (with a given name) to Board.

Lets look at the creator pattern (13.2) and then look at the process by which it is used.

Pattern Name:	**Information Expert**
Problem:	What is the basic principle by which we assign responsibilities to objects?
Solution: (advice)	Assign a responsibility (knowing or doing,) to the class that has the information needed to fulfil it.

Table 13.2: The Creator Pattern Larman 65 using the less detailed format.

As suggested above this pattern can apply to either knowing or doing,[65]. For example if we were looking to find the class that reset the squares (in the event of reset or new game,) we would assign the responsibility of doing this to Board, again following the Information Expert pattern.

Software Objects and the Design Class Diagram - DCD .

Again we're working with software objects in the Design Class Diagram (DCD) but we're off the starting line and have some objects already in our model, so the process changes slightly.

This is the adapted from that described in Larman 65, using the same format as we've used earl

Step	In this example.
1. Identify and describe the problem:	*Who Knows about the Square object, given a key?*
2. Decide on a pattern to solve the problem:	*We've chosen **Information Expert.***
3. Look for existing software objects to satisfy the roles of A and B in the pattern: • **iteration *n*:** Look for existing software objects in our DCD. • **iteration 1:** We are just starting the design, using LRG (lower representational gap) identify appropriate domain classes (from the domain model,) and promote them to software classes. Add these to the DCD.	*Board and Square are already software objects, and Board is chosen to be the Information Expert*

We now have to consider the relation between these two objects, fortunately the existing composition relation does not need to be changed. However the ability to fulfil this responsibility must be reflected in the relevant diagrams. Agile modelling suggests that both the dynamic (behavioural - sequence,) and static (structural - DCD) should be worked on in parallel [65]. Larmans images (see **??**) are cleaned up whiteboard images, these early stages really need whole team involvement.

NOTE: Remember this are part diagrams, no one in their right mind would draw a class diagram for these other than for illustrative purposes. Imagine these fragments as part of a complete sequence diagram for a given scenario or use case.

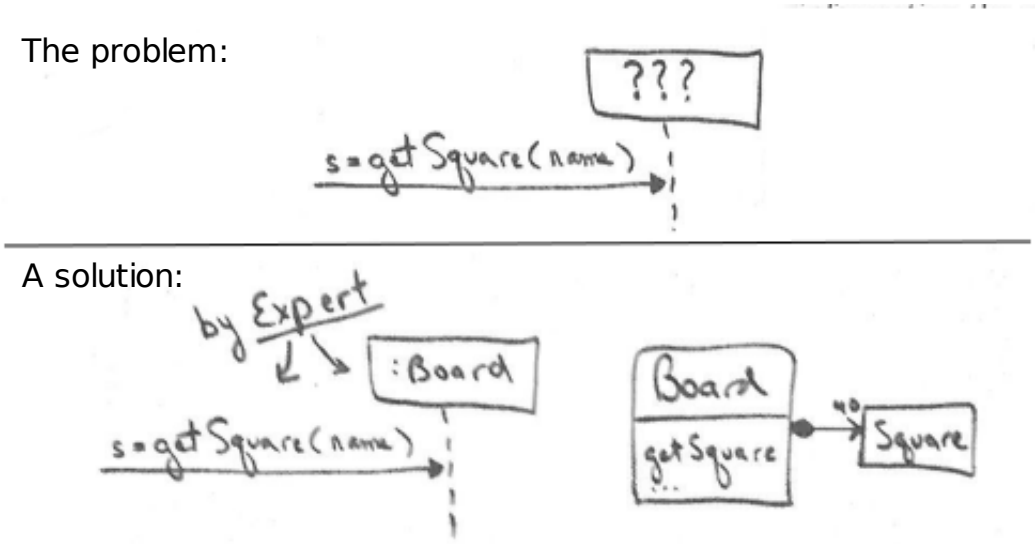


Figure 13.2: The expert pattern being appiled and updated DCD from Larman 65. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

Low Coupling

Question: Why Board over Dog?

Larman 65 provides a very detailed discussion of this example and the case study it comes from, hopefully my cut down version has left enough context for this to still make sense.

In the Creator (see **??**) and Expert (see **??**) we assigned responsibilities to Board rather than some other arbitrary class, e.g. Dog. The Expert Pattern suggested that Board which knows about all Squares, should be responsible for finding an individual Square given a key. Underlying this is the principle of Low Coupling, which we introduce though this now familiar example, but which is a central goal in the creation of software [65]. Informally coupling is a measure of how strongly one element is connected to, depends on, knows of etc., other elements [65]. For example if element A is tightly coupled (or dependent upon,) B then any change in B will have an impact (require changes in,) A. This is a very general principle which occurs at many levels of software development [65] and one you may have come across before in first year programming.
So the question is:

"Why Board over Dog?"

Lets look at the Low Coupling Principle (13.3) in the same format as the other patterns/principles we've looked at, its important to remember that these all provide advice. In this case Low Coupling supports Information Expert [65], in other cases principles will conflict and we'll need to weigh up the arguments for each.

Pattern Name:	**Low Coupling**
Problem:	How to reduce the impact of change?
Solution: (advice)	Assign a responsibility so that **unnecessary** coupling remains low. Use this principle to evaluate alternative solutions.

Table 13.3: The Low Coupling Principle Larman 65 using the less detailed format.

This is a little different to the first two patterns we looked at, we use Low Coupling to evaluate design choices (or existing designs as targets for refactoring,) unless there are external factors we favour the solution with the lowest coupling [65]. We'll start with an alternative solution to a problem we've already solved "Who knows about a Square object, given a key?", this will allow us to evaluate this against the solution suggested by the Information Expert pattern this is shown in 13.3.

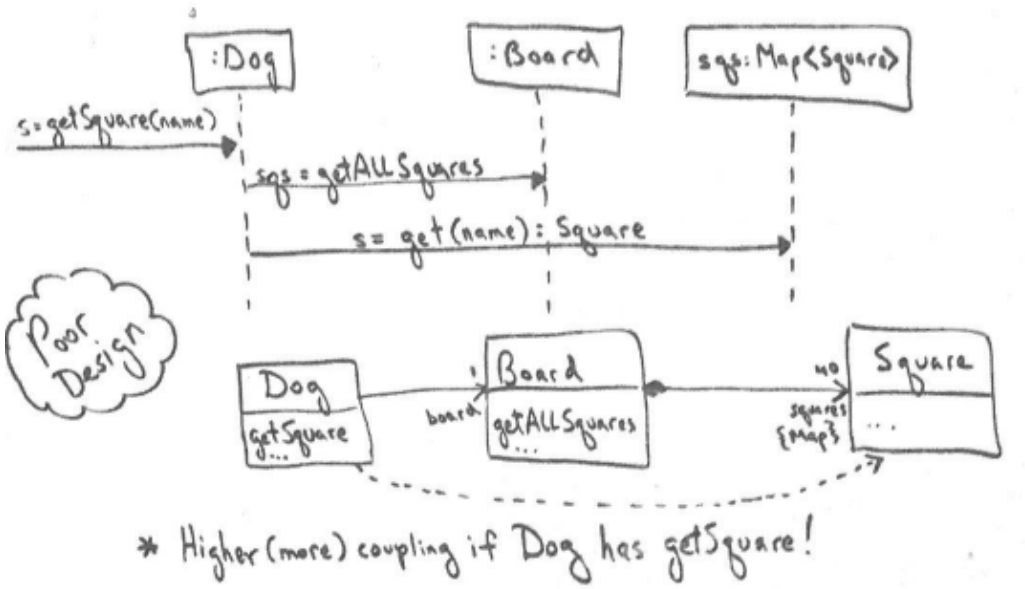


Figure 17.7 Evaluating the effect of coupling on this design.

Figure 13.3: An alternative solution to the problem of "Who knows about a Square object, given a key?" from Larman 65. The original (superior) solution is show in 13.2

Going back to our previous evalaution based on Larman 65, but using the tabular form to aid clarity (see 13.3).

Class Name:	Patterns or Principles	Evaluation
Dog	• Lower Representational Gap (LRG) • **Low Coupling**	• An arbitrary class be responsible for knowing create squares, however this doesn't fit well with our mental model of the problem. • Coupling refers to a measure of connectedness between elements, minimising this reduces the impact of change on one class on others.
Board	Information Expert	This is the obvious choice (and Larman 65 suggests that of most OO developers,) for an object to find an present one square (given a name,) it must know (have the information) about all the Squares.
Piece	Information Expert	This may fit conceptually, the piece exists in our domain. However in terms of having the information required it is hopefully clear that while piece knows its own square it doesn't know any others. Therefore it cannot fulfil the responsibility.

Table 13.4: Evaulation of the candidates for creating square based on Larman 65.

We could assign the responsibility to the Dog class (as shown in 13.3) but this would mean that Dog had to get all Squares from Board, then know enough about the Square class to find the one with the right name [65]. This requires Dog to know about (or be coupled to,) both Board and Square! The solution suggested by Information Expert (i.e. Board has findSquare(name),) requires only that Board knows about Squares i.e. Board is coupled to Square - the coupling is lower for this solution.

Software Objects and the Design Class Diagram - DCD .

This time we're not changing any objects but evaluating existing or alternative solutions [65]

There is no process described in Larman 65, using the same format as we've used earlier, I've tried to provide a framework for the process.

	Step	**In this example.**
1.	Identify and describe the problem:	*Why Board and Not Dog?*
2.	Decide on a principle to evaluate the solutions:	*We've chosen* ***Low Coupling***
3.	Consider the Coupling of each Solution: • **Dog** Requires Dog to be coupled to Board and Square. • **Board** Requires Board to be coupled to Square.	*Board and Square are already coupled (due to creator pattern,) and this solutions gives the lowest coupling Board is chosen. This also fits with Information Expert which itself aims to reduce coupling.*

The diagrams in this section introduces some new syntax (see the section on class diagrams 13.5 and sequence diagrams 13.6 for details).

Controller

Question: What is the first object after (or beyond) the UI layer should receive the message from the UI layer?

Larman 65 provides a very detailed discussion of this example and the case study it comes from. Again the example we'll use is from a scenario/use-case for the first elaboration iteration of a Monopoly game, where the game is initially set-up, a single piece is added to the board and moved around the board by dice roles (without player interaction).

In a layered architecture the UI layer handles all interactions with the user, i.e. showing windows, receiving mouse clicks and keyboard presses etc. [65]. In this case a button press results in a message being sent from the UI layer to some class '???' in the domain layer who controls/coordinates this activity.

From the Model View Principle (see **??**) we know that UI objects should not contain business logic, (e.g. working out the next square etc.,) so this must be delegated to a domain object (in the domain layer) [65].

So the question is:

"What is the first object after (or beyond) the UI layer should receive the message from the UI layer?"

First we review the SSD (see 13.4,) this shows a message generated by the human observer *playGame* [65]. Larman 65 goes on to discuss the potential sources, essentially when the code which the button runs (however it runs it - this will differ from Qt to Java, .Net etc.) this must create a call to the System. I think Larman's choice of example is an issue here, so lets think back to the other SSDs we looked at. Or jump back to 10.3, here *addItem(item, quantity)* was the message. While the conversion of an event like a gui button on the screen being converted to a call on a domain object is trivial, the *addItem* example requires some information to be collected from other sources (probably text boxes, filled from the keyboard and bar-code reader).

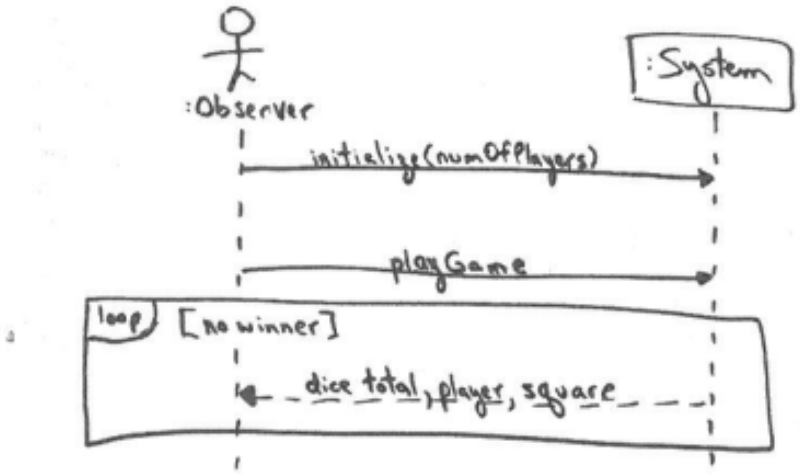


Figure 17.8 SSD for the Monopoly game. Note the *playGame* operation.

Figure 13.4: The SSD for playGame() from Larman 65.

Note the connection between the SSD and this part of the design Larman 65 is keen to point out the importance of the SSD here, later he explains that the interfacing system may not be a user, but an interface to a device (or group of devices,) a network port or some other outward facing entity.
Next lets look back at the object domain model to see what classes we have in addition to square (see 13.5).

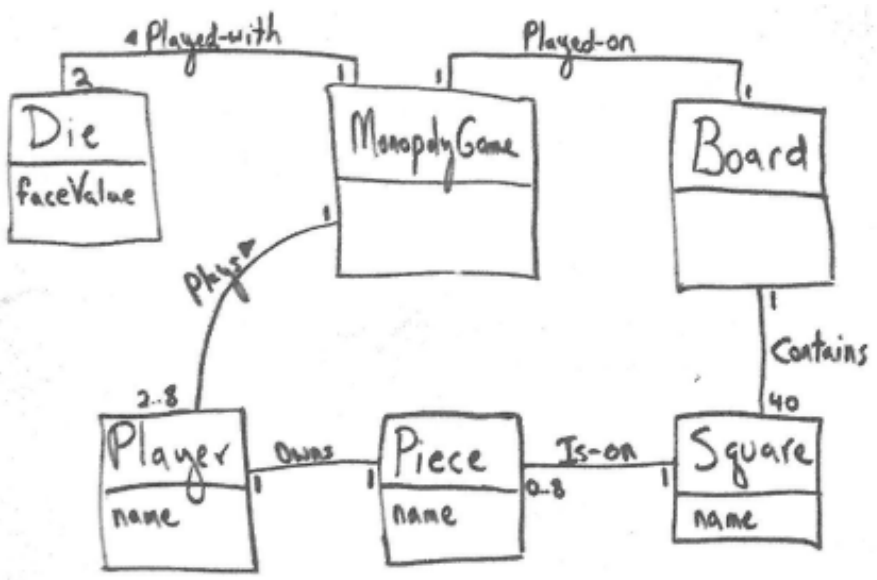


Figure 13.5: The domain model for the first elaboration iteration from Larman 65.

Its kind of obvious we're going to use the Controller Pattern here, (following the pattern from the other sections,) so lets look at the that first (13.2). We'll then evaluate possible implementation of its advice and look at some examples Larman uses to show this being applied during agile modelling.

Pattern Name:	**Controller**
Problem:	Which first object beyond the UI layer receives and coordinates (or controls) the a system operation?
Solution: (advice)	Responsibility should be assigned to an object representing one of the following: 1. An object representing the overall "system" or "root object" (this is a variation of the *facade controller*) 2. A major sub system (another variation of the *facade controller*) 3. A device the software is running within(yet another variation of the *facade controller*) 4. A use case scenario within the system within which the operation occurs (a use case or session controller).

Table 13.5: The Controller Pattern Larman 65 using the less detailed format.

As usuall there are several options according to Larman 65, I've put his discussion of each into a tabular form to aid clarity (see 13.6). One marked difference here is that the choice of object may include the creation of a new object (or objects,) this is one of those were first look at the options controller suggests then a principle (High Cohesion 13.4.2) with which to evaluated the options.

Class Name:	Patterns or Principles	Evaluation
MonopolyGame	• Controller see 13.5 • High Cohesion (another pattern see 13.4.2)	• This is a "system" or "root object" as suggested. • If the controller becomes to big we violate this see 13.4.2.
A device	Controller	This won't work - we don't have a device. If this was running on a phone or a cash machine we could use *Phone* or *BankCashMachine*.
Play Monopoly Game	Controller	This is an option, Play Monopoly Game is the use case that the playGame operation game from. When this is used we normally append "...Handler" or "...Session" to the name.

Table 13.6: Evaulation of the candidates for selecting a controller Larman 65.

Larman 65 cautiously accepts the first of these (MonopolyGame) however he notes that there are trade-offs with the patterns, in this case if the controller dose to much we get a bloated controller. We're going to consider the patterns in more detail over the next few weeks including the issues to consider when deploying them.

Software Objects and the Design Class Diagram - DCD .

Again we're working with software objects in the Design Class Diagram (DCD) but we're off the starting line and have some objects already in our model, so the process changes slightly.

This is the adapted from that described in Larman 65, using the same format as we've used earl

Step	In this example.
1. Identify and describe the problem:	*What is the first object after (or beyond) the UI layer should receive the message from the UI layer?*
2. Decide on a pattern to solve the problem:	*We've chosen* ***Controller***
4. Review the SSD, consider the message coming from the UI layer as a system operation (see 13.6)	
5. Look for existing software objects to satisfy the roles of Controller: • Look for existing software objects in our DCD. • If just starting of or if the above fails, using LRG (lower representational gap) identify appropriate domain classes (from the domain model,) and promote them to software classes. Add these to the DCD. • If none of the above fulfil the role we may consider adding a class (this is Pure Fabrication another GRASP pattern 14.3.3).	*MonopolyGame is added as our Controller, this is reasonably only if it ends up with only a few system operations (i.e. doesn't violate High Cohesion)*

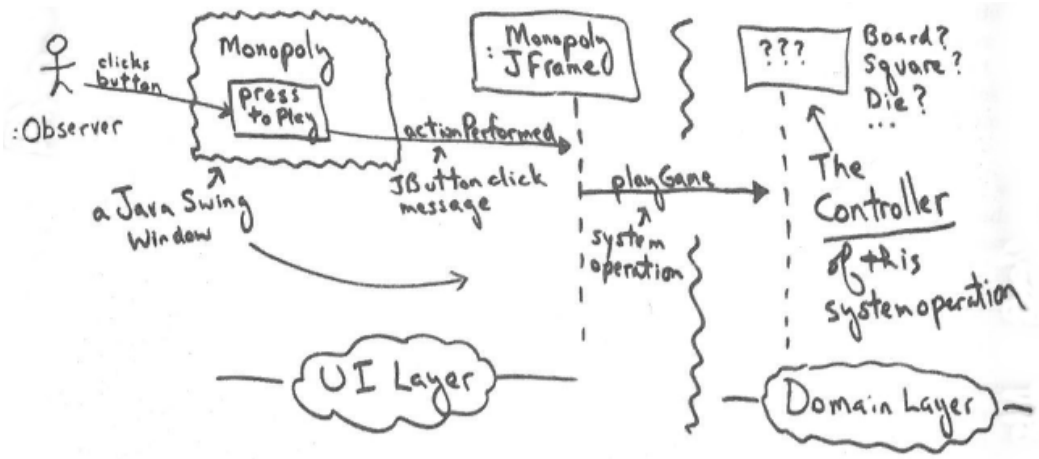


Figure 17.9 Who is the Controller for the *playGame* system operation?

Figure 13.6: Agile Modelling in Action, I like to think this is the moment someone said "who should be the controller?" during the agile modelling session. Actually its taken from Larman 65 discussion so its probably just illistrative.

We now have to consider the relation between these two objects, we add the class to our DCD, we can't connect it up just yet as we need to think about what playGame involves and how it impacts on other objects. Agile modelling suggests that both the dynamic (behavioural - sequence,) and static (structural - DCD) should be worked on in parallel [65]. Larmans image of the completed controller, is a kind of after image of the earlier one 13.7.

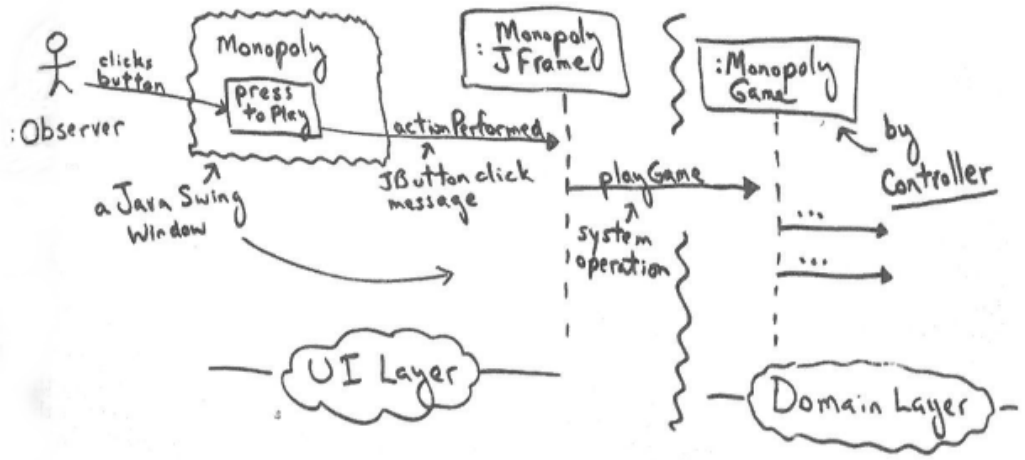


Figure 17.10 Applying the Controller pattern—using *MonopolyGame*. Connecting the UI layer to the domain layer of software objects.

Figure 13.7: This is the controller pattern being applied (taken from [65].

NOTE: This is only a partial scenario, the next step is to work out what objects need to do what. We're not going to go into this in any detail but we will consider High Cohesion a principle which helps us to evaluate our choice (and in part one we've just made).

High Cohesion

The next step is to consider how the domain objects are directed will follow in a few classes time, but one of the key principles we can consider now [65]. This is something we've considered before in the first year - Cohesion. Lets review the problem and possible solutions, we're looking produce the mechanistic design for playGame (that is how do the domain classes collaborate to throw the dice, move the piece etc.) Larman 65 summarises this as a part-complete sequence diagram 13.8.

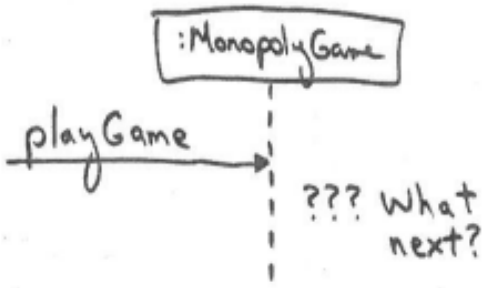


Figure 13.8: Agile modelling - what do we do next?! (taken from [65].

The possible solutions are again show as partial sequence diagrams (see 13.9) but these differ in the level of cohesion.

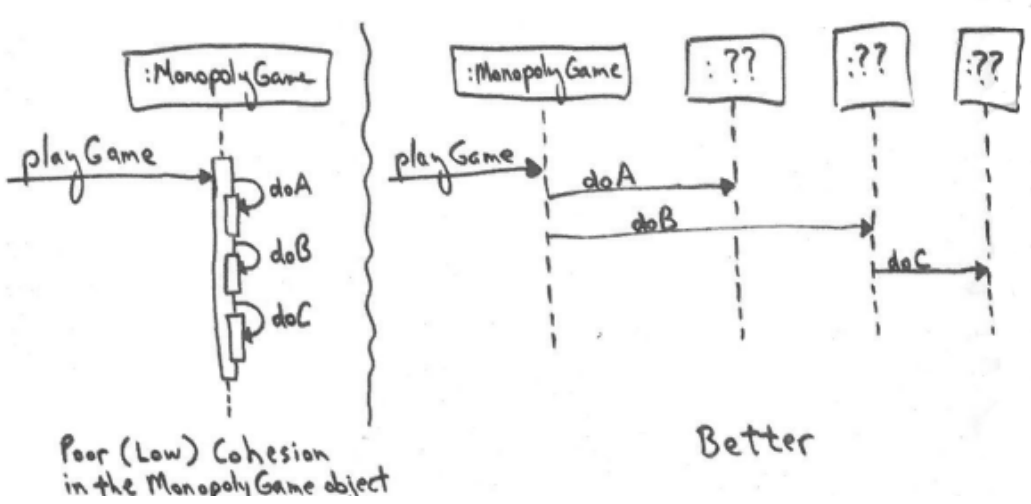


Figure 13.9: High and low coupling and example taken from [65].

Cohesions provides an informal measure of how functionally related operations of a software element are, in this case we're talking about the methods (which fulfil responsibilities) of an object [65]. It also measures the amount of work done by a software element [65]. The book goes for a software based example to begin with but I'll start with an analogy, a reasonably size builders merchants has around 30 staff spread over a head office and several yards. A member of staff Mr. A is responsible for working on the counter at Yard B, organising the payroll for the whole company,

completing the tax return and Health and Safety at yard C. This is not a cohesive set of responsibilities (and hopefully the various issues with this are obvious). So lets re-organise the company, so we have staff at each yard responsible for counter sales, stock control, deliveries and Health and Safety only, in the office we have staff responsible for payroll, finance, etc. Mr C. the manager is responsible for overseeing the office staff, finance, payroll, deliveries, sales, stock control ... in one morning he may authorise the purchase of some bricks from a stockist for Yard A, the transfer of some paint from Yard A to Yard B, a repair to a truck at Yard C, annual leave for a member of the sales team at Yard A etc. Again this is Clearly this is also not quite right, we'd expect to see a layer of management between the overall manager Mr. C and the front line staff. Perhaps a yard manager who took responsibility for the day to day running of the yard?

So the overriding principle here is that an object shouldn't do unrelated jobs and should not do too many jobs (i.e. we should delegate to other objects if there are a large number of related responsibilities).

Going back to the example 13.9 we can see that the right hand version is much more cohesive [65] despite not know what A, B and C are in this context it doesn't matter. If A, B and C relate to different responsibilities its cohesion is low, if they are all related to one responsibility and its just big or complex this is still low cohesion and we need to divide it up. Going a step further if doA() etc. all require at least two collaborating classes (A1 and A2 say) then the left hand version is highly coupled. This leads us to another key point High Cohesion and Low Coupling often go together [65].

Finally we introduce the principle, (and this is more a principle than a pattern,) High Cohesion 13.7 which can be used to evaluate solutions to find the best alternative.

Pattern Name:	**High Cohesion**
Problem:	How to keep objects focused, understandable, manageable and as a side effect support Low Coupling?
Solution: (advice)	Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

Table 13.7: The High Cohesion principle Larman 65 using the less detailed format.

Note: Looking back a little we notice that our use of the controller pattern may cause low cohesion, if we try to use the same controller for everything this is called bloated controller [65].

13.5 More Class Diagrams

13.5.1 Dependency

Associations (and their specialised forms - aggregation and composition,) indicate that one object has an attribute of a type which is another object [61]. Dependency is used to indicate that while the object does not have an attribute whose type is another object, it uses one in some other way [61]. The dependency arrow in UML is an open headed (crows foot,) arrow with a dotted line which may have a stereotype to indicate the nature of the dependency [61].

This shows a kind of coupling between classes [61], is useful when we're analysing solutions for Low Coupling!

Guideline: When to Show Dependency

Larman 58 suggest that we should only show dependencies in the following circumstances (note UML describes the dependent class as the client and the depended upon as the supplier):

- Accessing a local or global variable of the supplier type, this could mean accessing a static constant of a class in C++ or creating a temporary object inside a class.
- Calling a static method of the supplier type, this could mean accessing a static constant of a class in C++, see example 13.10(A)
- Receiving a parameter of the supplier type, see the example 13.10(B)

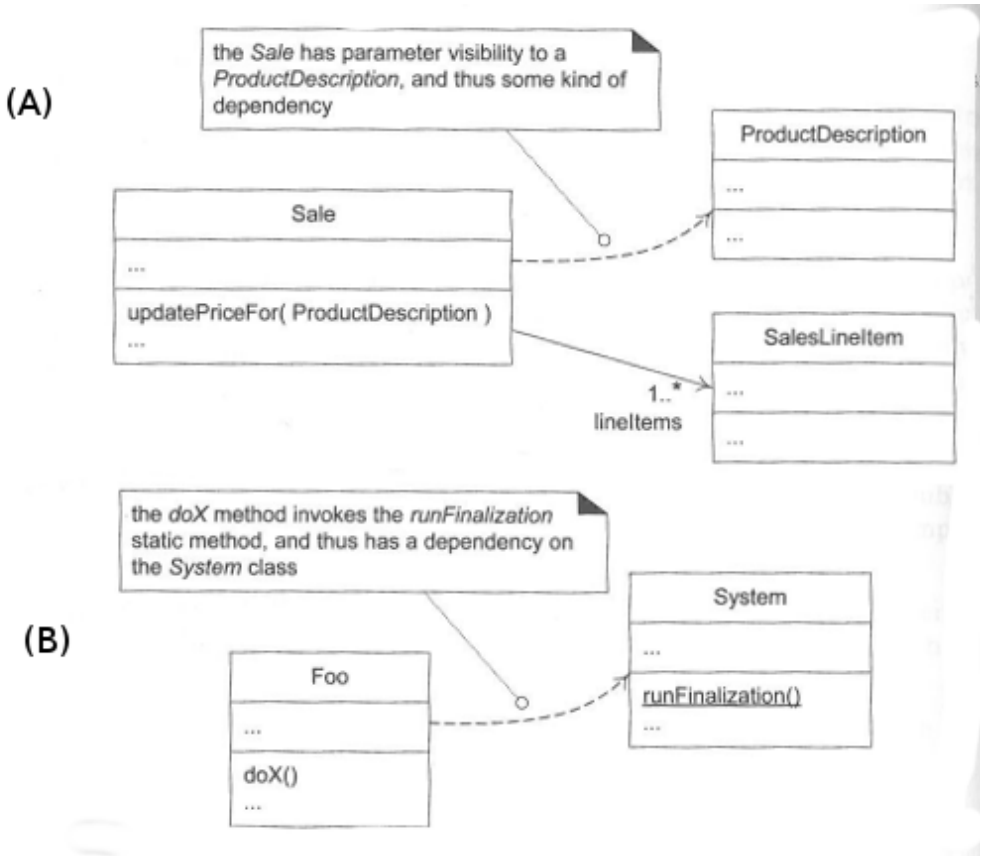


Figure 13.10: Showing dependencies (which are not association/inheritance) in UML class diagrams adapted from[61].

UML provides a number of keywords to help clarify the dependency line (The specification I have (2.5) dates after Larman's book and gives the following) [88]:

- **«instantiate»** - the client creations an instance of the supplier.
- **«use»** - the client uses an instance of the supplier.

13.6 More Sequence Diagrams

13.6.1 Collections and Return Variables

The examples shown above (in the section on Low Coupling13.3) use an alternative syntax for return variables to the one we've used previously (to save whiteboard space). Larman 65 notes the following elements:

- The return variable *sqs* from *getAllSquares* is the same as that named in the 3rd lifeline. This is an object of type Map<Square> a type of map that holds Squares. This referencing of a return variable in the message (sent to it,) rather than as a return is common.
- Similarly the variable *s* in which begins the *getSquare* message and the variable s in the later get message refer to the same object.
- The formatting of the message *s = get(name) : Square* indicates that a Square is returned with the name *s*.

An additional example is show in 13.11.

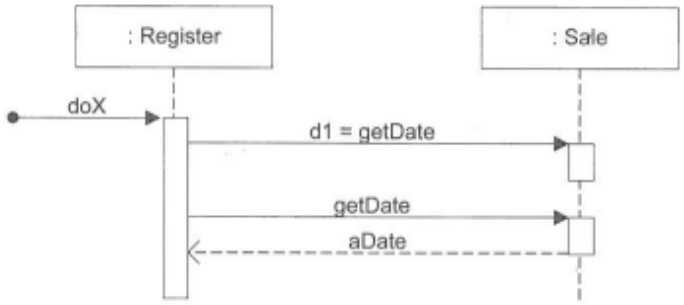


Figure 13.11: Showing return values in sequence diagrams from[58].

13.6.2 Messages to Self

The example for High Cohesion(see 13.9) shows a number of self calls in a sequence diagram, this is used to show an object calling methods from itself (e.g. calling a private helper method). This is shown as a message arrow point back to the originating lifeline with a nested activation bar [58].

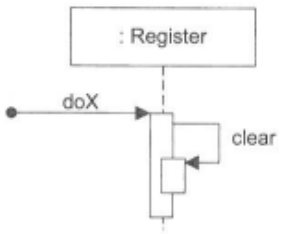


Figure 13.12: Showing self calls in sequence diagrams from[58].

An additional example is show in 13.12.

13.7 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 13.7.1, and consider the use case 13.7.1, domain object model 13.14 and SSD 13.15 then use an appropriate pattern to identify:
    - Who is the first object beyond the UI layer who receives the message in the SSD?
    - Who knows how to convert a measure between a given unit and another unit?
    - Who knows how to create a formatted output for a given unit?
2. Read the case study 13.18 see the ODD 13.16 and DCD provided **??** and the SSD **??** then use an appropriate pattern to identify:
    - Who creates InputPattern?
    - Who creates OutputPattern?
    - Who can find an Input Pattern given an index?
    - Who can find an Output Pattern given an index?
    - Who can load Input and Output Patterns from file, given a filename?

Questions to think about (or what kind of exam question could I get on this).

- What are the GRASP patterns and why are they useful in the development of object orientated software?

13.7.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in

an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

Use Case

The team have produced a number of use case summarised at the highest level by the following use case diagram. The names of the use case here may differ slightly from those we identified during the tutorial.

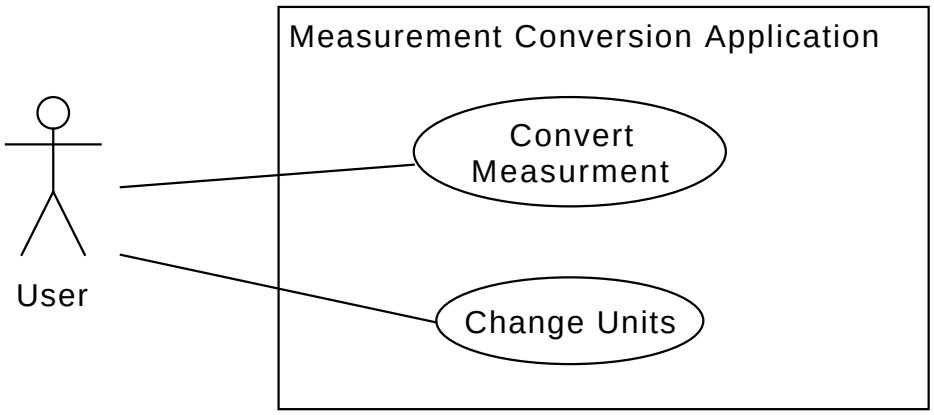


Figure 13.13: This is an incomplete use case based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough.

Use Case: Convert Measurement

UC4: Convert Measurement L13	
Scope:	Measurement Conversion Application
Primary Actor:	User
Level:	user-goal

Stakeholders and Interests:

- User: Wants to convert between imperial and metric measurements of length.

Preconditions:

- Application has launched and the interface is displayed
- Interface is prepared for user input

Postconditions (success guarantee):

- Output is presented in a suitable format

Main Success Scenario:

1. User enters a measurement in the default unit (imperial,)
2. User requests a conversion (indicating they have completed their data entry).
3. System performs the conversion
4. System displays the converted measurement

Extensions (Alternative Flows):

- 1a. User changes units before entering a measurement.
- 2a. User enters nothing
    1. System does not attempt to convert
    2. System highlights the lack of input (e.g. change background of input field to red)
- 2b. User enters an incorrectly formatted measurement for the current conversion.
    1. System Displays an error message
    2. System highlights the incorrect input (e.g. change background of input field to red)
    3. User enters a measurement

Special Requirements:

- Must be possible to enter the measurement in a number of formats, e.g. 1'6", 1.5", 18" etc.
- The output must be such that the user can easily copy the output for use in another application.

Technology and Data Variations List:

1. ...

Frequency of Occurrence:	Expect the user to perform one or more conversions in a single session

---

Open Issues:

- Given that the input can be in a number of formats, which the system is to deduce. How will this be done?
- The output may be required in a specific format, should this option be given to the user and how? This could be a property set though a menu (with a default,) or some means of selection near the output.

Domain Object Model

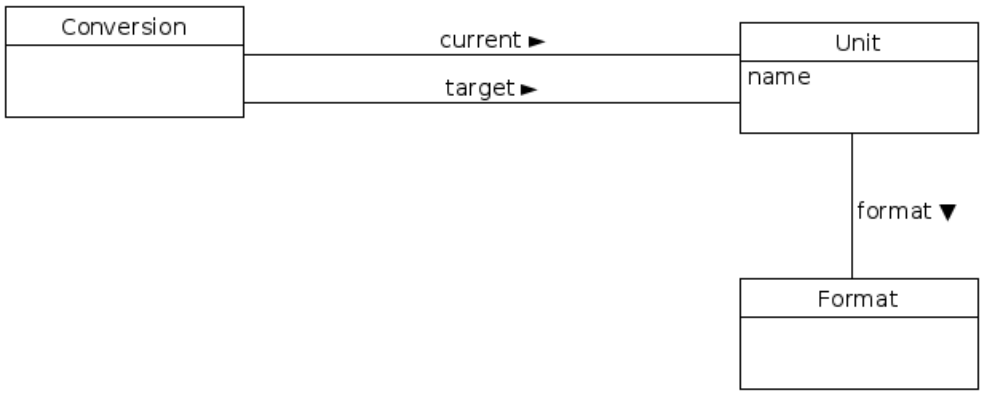


Figure 13.14: This is a domain object model based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough. I've avoided describing this as a draft (suggesting its incomplete,) or using any other term which would suggest it is final. In the spirit of Agile Modelling we are using it to inform sketches of the design so we can discuss and evaluate design options. Changes may be made as a result of this discussion and its probably that future iterations would contain some of the conceptual classes.

System Sequence Diagram

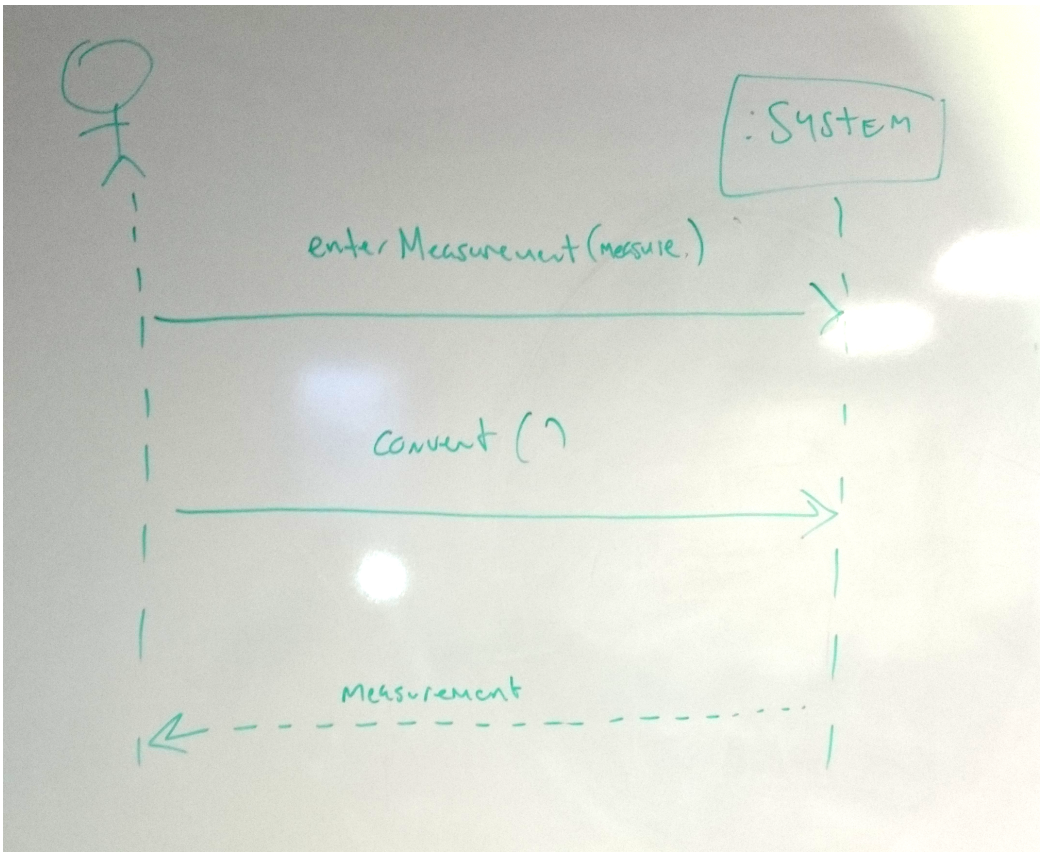


Figure 13.15: This is the one we did in class, I'm not sure if its going to be usable!

13.7.2 Case Study 2

A researcher at the university is looking to write a paper with a colleague they have both datasets and a new neural network to train on them. In order to do this they need to load in files from the original project so that their Neural Network will be trained and tested with the same files. Each file consists of comments (which start with a #,) some meta data (information about the file content) and the data itself (see example file), all in a CSV format. The data is in the form of input and output patterns, each contains t (a time offset). The input contains the values for parameters t, r, g, and b while the output patter has two values t and o. Due to the nature of neural network operations all parameters are floating point numbers between -1.0f

and 1.0f further validation is possible as the t (which represent the time-step,) should be identical in the input and output pattern.

13.7.3 Domain Object Diagram for Case Study 2

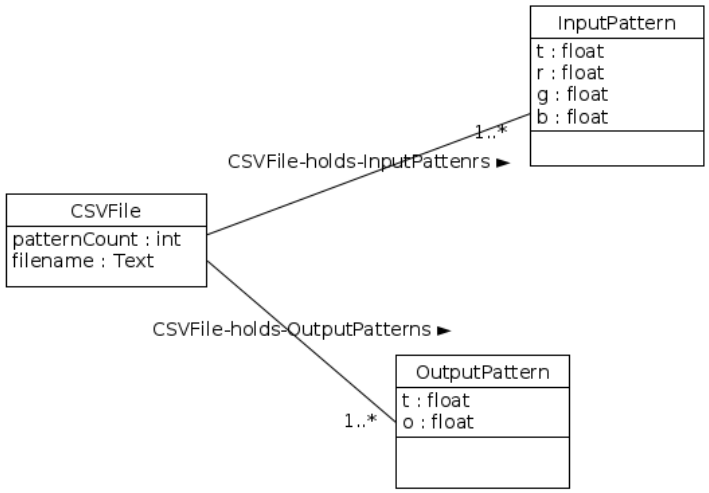


Figure 13.16: Domain Object Model for Case Study 2.

13.7.4 System Sequence Diagram for Case Study 2

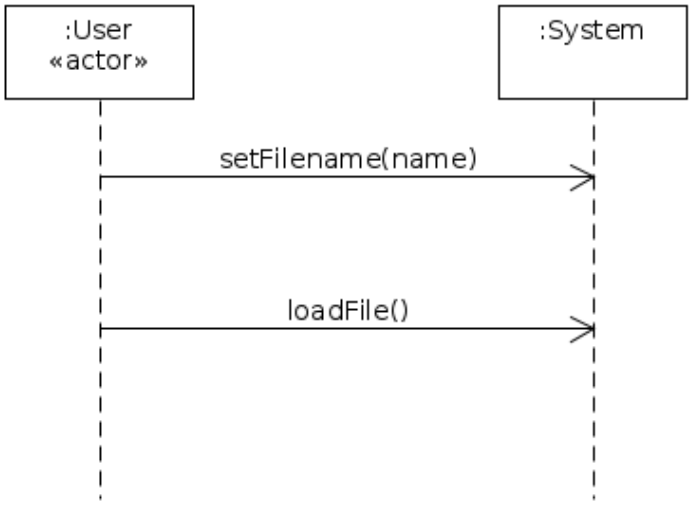


Figure 13.17: SSD for Case Study 2.

DCD for Case Study 2

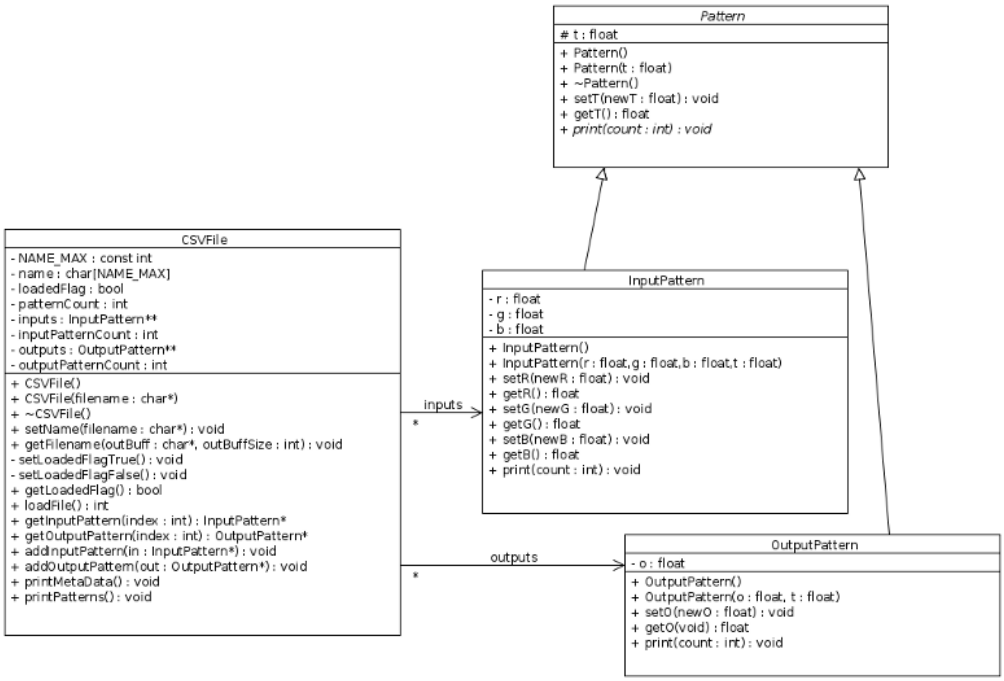


Figure 13.18: I happen to have this as I set is as a coursework, to end up with this we'll need one pattern I've not covered yet!.

13.8 Summary

The next important part of the design process is the Realisation of the use case, which is the mapping of use case and individual scenarios to object behaviour. We've started small considering parts of these interactions in isolation and the low level patterns (or OO principles,) we've looked the most essential of the patterns (including creator we did last week). There are only a few more to do, but we're going to use these to realise some use case these before looking at them.

The GRASP patterns are low level simple design patterns/principles named and documented to make them easier to remember, discuss and use. This week we've looked at Information Expert, Low Coupling, Controller and High Cohesion with some examples of their application, however we've not considered the pattern in full (which we'll do in a later class) at the moment we

need to see a few more patterns so we can start using them in the realisation of use case.

13.9 Preparation for Next Week

13.9.1 Essential Reading

An introduction to GRASP and Patterns, this covers this week and last weeks material..
Larman (2014q). *Applying UML and Patterns*, chapter GRASP: Designing Objects with Responsiblities, pages 312–362. Prentice Hall PTR.
For next week read as much of chapter 18 as you can! I'll not be going into as much detail (or doing as many examples as Larman)!
Larman (2014p). *Applying UML and Patterns*, chapter Object Design Examples with GRASP, pages 312–362. Prentice Hall PTR.

13.9.2 Background Reading

In conjunction with the above you may need to refer to these chapters which provide very detailed coverage of Class and Sequence diagrams. Larman (2014j). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

Larman (2014m). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Designing with Patterns/Principles: Mechanistic Design - Part 3

14.1 Introduction

In the previous notes (13) we'd started object design and were looking to begin use case realisation. In an attempt to systematically design object orientation systems adopted a responsibility driven design approach (RDD). This suggested the use of GRASP patterns in order to assign low level responsibilities and we'd looked at a few of these patterns. We're going to add two more (of the most useful,) then look at some examples, I'm going to provide a chapter to read from the book rather than a set of notes and spend much of the class

14.2 Use Case Realisation

We're moving on to Use Case Realisation or perhaps more accurately scenario realisation is the process by which we describe the design (in terms of objects,) or one or more use case [64]. The term Use Case Realisation comes from UP and is used to reinforce the connection between the use case in analysis and the design of the system [64].
Larman 64 argues that object orientated design is simply a question of assigning responsibilities to objects and designing collaborations and that both can be logically rationally explained and learned. Also that mastering OO design requires the deployment of a large number of soft principles (with many degrees of freedom,) this seems daunting but these principles can be named and learned (as we've started to do with GRASP).

14.2.1 Orientation: What's next?

The plan is as follows:

- **GRASP** We'll look at two more patterns from GRASP and one extra I like. These give us a bit more scope (and we need one for our case studies 14.2.1).
- **Realising Use Case with GRASP** I had originally planned to to straight on to look at GoF but I think we'd be better placed to use what we've done so far in anger first. So we're going to use GRASP to realise some use case.
- **GoF Patterns** We'll the move on to look at some (or perhaps all) of the Gang of Four patterns, these are well know and well used design patterns which will give us a flavour for their use.
- **More Class Diagrams and Sequence Diagrams** Along side the principles and patterns we'll be looking at more UML, which we'll need to visualise the patterns - though we'll build some in C++ as we go!

Essentially we're working up to realisation, starting small with the GRASP - small patterns (or principles,) these should be enough to get us started.
NOTE: The need for extra patterns is due to my plans for the case studies being designed around exercising as much OO/UML as possible (including inheritance,) Larman in his wisdom avoids this in the early examples.

14.3 Responsibilities and Responsibility Driven Design (RRD)

14.3.1 Recap: What is RRD?

In RRD objects are attributed responsibilities, in terms of what they are responsible for knowing and what they are responsible for doing [65].
There are two types of responsibility knowing and doing [65]:

- **Doing** - Responsibilities may include:
  - Doing something itself, e.g. a calculation, creating and object etc.
  - Causing something to happen in another object, or changing its state.
  - Controlling and coordinating activities in other objects (e.g. **??**)

- **Knowing** - Responsibilities may include:
  - Holding data itself in private attributes
  - Knowing about related objects
  - Knowing about data it can derive or calculate

14.3.2 Recap: GRASP - Design Principles (mini-patterns)

GRASP is a learning aid which names and describes some basic principles to assign responsibilities to objects, proving a simple first step towards RDD [65]. We're using it to provide a foundation for applying more complex patterns such as the Gang of Four (GoF) patterns which are 'larger' and deal with coordinating higher level behaviour.

As we've noted previously these patterns shouldn't be new but naming them allows them to be learned, explained and applied [65] leading to a more systematic approach to the delegation of responsibilities (and therefore object behaviour). We're going to come back to these and discuss them in more detail at a later stage, without a name we couldn't do this!

Recap: When do we assign Responsibilities?

Responsibility are assigned either during design when we draw interaction diagrams and decide where the methods which fulfil these responsibilities go or while coding [65]. The GRASP principles can be applied to guide choices about assigning responsibilities and can therefore be applied during design (while drawing interaction diagrams,) or during coding [65].

Note: Recently I've spoken to two members of staff who stated that second year students were struggling to develop OO solutions. Applying these principles/patterns when your writing code for them should help.

14.3.3 Solving A Simple Design Problems using a GRASP Patterns - Continued

Before moving on to a full scenario (or use case,) which is where we'd normally start in terms of mechanistic design, I'm going to continue with some 'parts' of a larger solution.
We'll see these deployed in a number of examples and this will will allow us

to work up to a full scenario. Essentially this will be the detail behind the call to system from our SSD (see 10.3). For context in this case the question arises as part of *throwDice* scenario.

I've stolen these examples from Larman 65 and Larman 67, we'll work though something similar for other examples later.

Something to note here is that these patterns are like the controller pattern moving much more into the design of the system and the use of LRG seems to decrease as we become more interested in creating well designed software and less so in the modelling of the domain.

Pure Fabrication

Question: What object should have the responsibility, when you don't want to violate High Cohesion and Low Coupling, or other goals, but the solution offered by Expert (for example) are not appropriate?

Larman 67 provides a very detailed discussion of this example as this is from a later section it introduces the pattern in a more formal way.

There are instances where the solutions proposed by simple patterns such as expert will lead to solutions which violate high cohsions or low coupling [67]. For example in the Monopoly problem the dice are associated with the Player object and undertakes some dice related operations (e.g. working out the total). In a more complex (and perhaps tangible) example an entity in a data driven system (e.g. a Sale being recored,) needs to be saved in a database. Information Expert suggests that as Sale has the relevent data it should undertake the storing [67]. However Larman 67 notes the following complications:

- A large number of database-orientated operations are required inside of sale to support this (e.g. connect, build query etc.,) this impacts on **High Cohesion** i.e. the class becomes incohesive.
- Sale becomes coupled to the interface to the database, **Low Coupling** is impacted and coupling increases. Worse than this the coupling is to a specific database interface rather than other domain objects.
- Saving objects is a general thing which lots of classes will do. Placing these responsibilities in Sale will lead to duplication in other classes

which must also be stored.

The question is:

"Who can we assign this responsibility to without violating High Cohesion and Low Coupling?"

First lets look back at the object domain model to see what classes we have in addition to square (see 14.4). In this case it doesn't help.

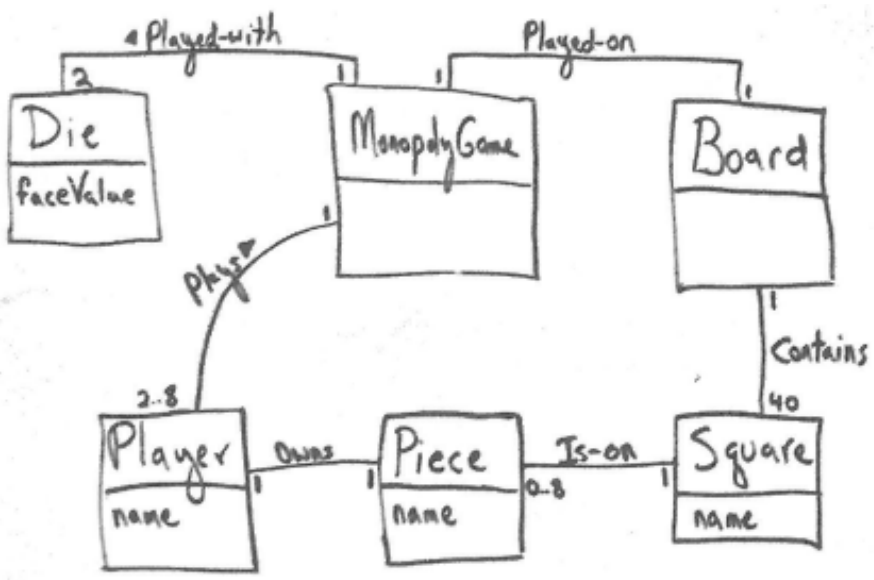


Figure 14.1: The domain model for the first elaboration iteration from Larman 65. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

In this case we're going to add a class, this will be something which doesn't exist in our domain (hence *Fabrication*) and hopefully something which gives us high cohesion and low coupling (hence *pure*) [67]. As in other patterns we'll try to maintain LRG (Low Representational Gap,) by choosing something sensible [67] in the case of the Monopoly Game *cup*. The performs the dice rolls for a number of dice and knows (and remembers,) the resulting number, many games use a cup when rolling dice so it fits with the domain, it could also be re-used in another game [67].

Note that this is essentially responsibility assignment, we've assigned the responsibility for finding and knowing the total of the dice to the cup class.

Lets look at the Pure Fabrication pattern (14.1) and then look at the process by which it is used.

Pattern Name:	**Pure Fabrication**
Problem:	What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?
Solution: (advice)	Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept - i.e. made up to support high cohesion coupling and reuse.

Table 14.1: The Pure Fabircation pattern Larman 67 using the less detailed format.

As in previous examples this pattern can apply to either knowing or doinig, [67].

Software Objects and the Design Class Diagram - DCD .

Again we're working with software objects in the Design Class Diagram (DCD) but we're off the starting line and have some objects already in our model, so the process changes slightly.

This is the adapted from that described in Larman 67, using the same format as we've used earlier

	Step	In this example.
1.	Identify and describe the problem:	*What object should be responsible for performing the dice role, and knowing the total while maintaining Low Coupling and High Cohesion?*
2.	Decide on a pattern to solve the problem:	*We've chosen **Pure Fabrication** (over Information Expert)*
3.	Think of (fabricate) a class which will perform the role, ideally this should either mean something in the domain developing the system	In this case we chose *cup* to be our new class, this supports LRG.

We now update the diagrams (DCD and relevant sequence digram,) I'm showing the diagram from the first iteration 14.2 and the updated one from the second iteration 14.3, Larman didn't need to solve this problem in the first iteration so he left it until much later in the book.

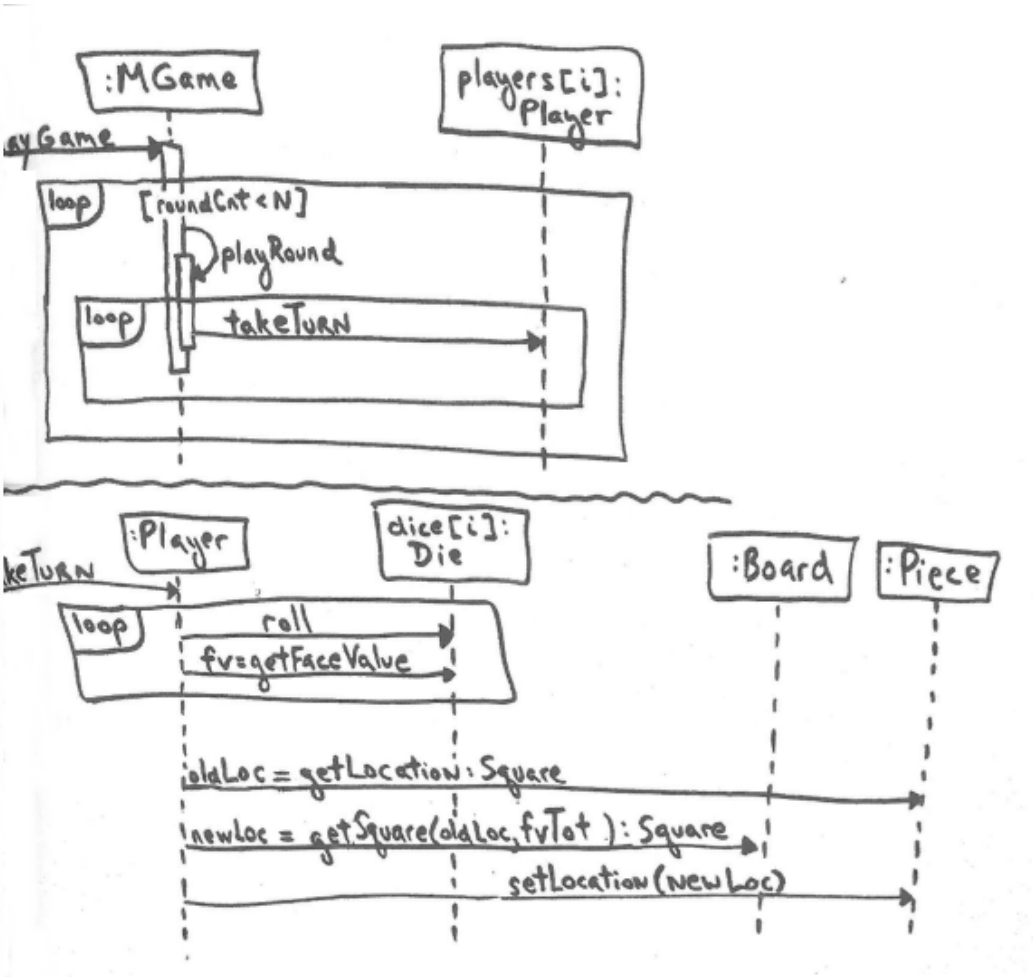


Figure 14.2: This is the original design for roll dice, the issue with this is that player is responsbile for rolling the group of dice and working out the total - these are dice related rather than player related - hence cohesion is low. This image is from Larman 67.

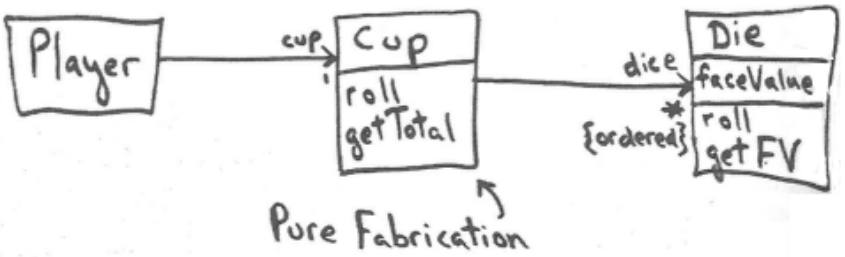


Figure 25.8 DCD for a *Cup*.

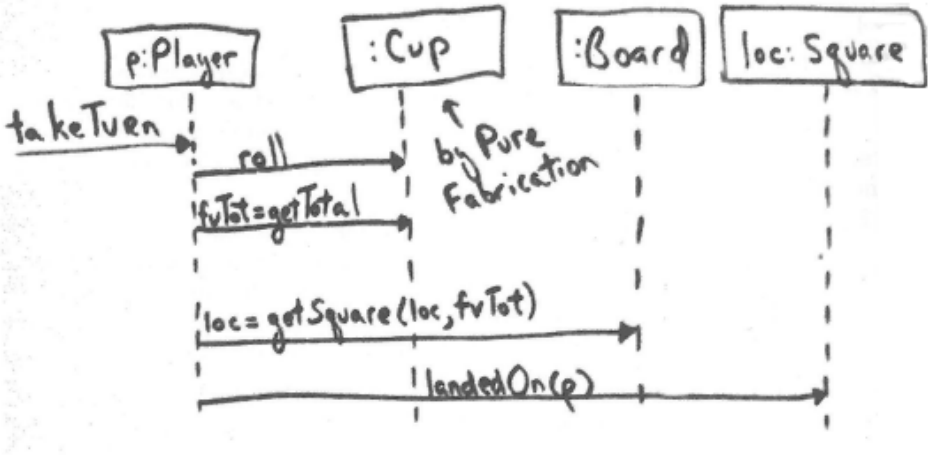


Figure 25.9 Using the *Cup* in the Monopoly game.

Figure 14.3: The Pure Fabrication pattern has been appiled to the above resulting in a updated DCD and Sequence Diagram from Larman 67. Note that all the responsibility for knowing about and doing things with dice is now managed by cup.

Something Larman 67 does pick up in the discussion which is worth mentioning is that the objects selected in design can be broadly divided into:

1. Those chosen by representational decomposition, where entities are responsible for representing things in our domain.

2. Those chosen by behavioural decomposition, where responsibility is assigned that to group behaviours.

Sale, etc. are all based on representational decomposition and this is a common design strategy which supports LRG [67]. Examples of classes which are groupings of associated behaviours include *TableOfContentsGenerator* which

contains a collection of methods associated with the generation of tables of contents [67]. This helper class was created by the developer without any concern for the domain vocabulary, it is a convenience class created by the developer to group together some related behaviour/methods. There are a number of similar patterns which provide behavioural (function-centric) partitioning e.g. Adapter, Strategy, Command etc. [67].

NOTE: Remember this are part diagrams, no one in their right mind would draw a class diagram for these other than for illustrative purposes. Imagine these fragments as part of a complete sequence diagram for a given scenario or use case.

Polymorphism

Question: How to handle different types of square? Or to allow for new types of Square in future variations?

Larman 67 provides a very detailed discussion of this example but as this is from a later section it introduces the pattern in a more formal way.

I've kept the above related to Square, however the more general questions are the question is (really):

"How to handle alternatives based on type? How to create pluggable software components?"

The above requires a little explanation, we often require conditional variation in our programs (i.e. we have something with an if-then-else or case statement) [67]. However taking this approach may mean the same condition being repeated in a number of places, this makes extension to include more variations as changes are required in several places [67]. Viewing software entities as clients and servers (or producers and consumers,) i.e. something which is coupled, we must consider how the server can be replaced without changing the client [67].

I suspect as with a good deal of the other questions, we kind of know the answer. We solved similar problems on the board in introduction to programming by using polymorphism. So lets lets look back at the object domain model,Larman has updated this to include squares for the second interation (see 14.4).

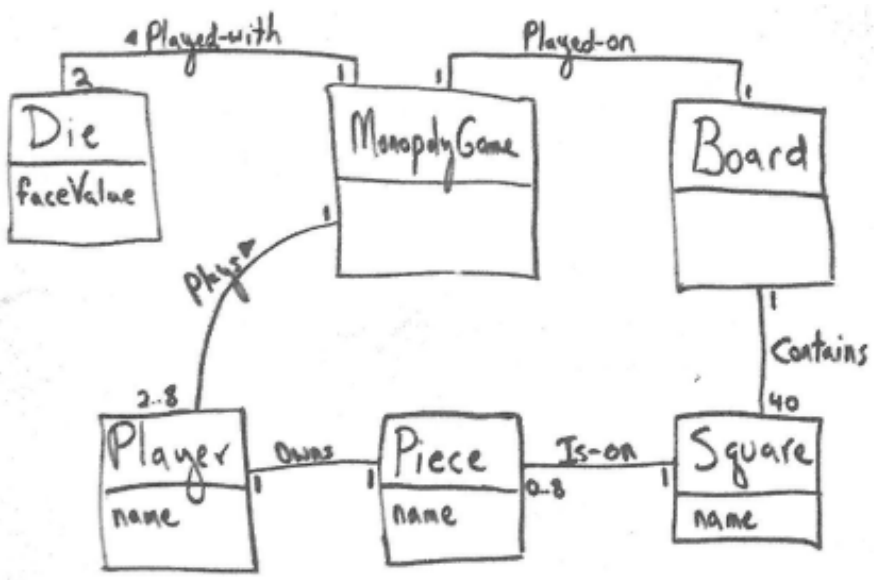


Figure 14.4: The domain model for the second elaboration iteration from Larman 67. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

Lets explore a few options, I'm using the format Larman does in his early examples and adapting the later ones to fit, this leaves out some of the discussion (please read the relevant chapter of [67]). I think the table format used in 14.2 still OK for this.

Class Name:	Patterns or Principles	Evaluation
Square	None - Switch Statement in Square	This is poor design the pattern explicitly advises against this.
GoSquare IncomeTaxSquare etc.	None - Separate classes	This is poor design we end up repeating lots of common code across a number of classes. The implications are hopefully obvious.
Square & children	Polymorphism	Create an abstract square class and a number of specific concrete derived classes with specialised behaviours as required.

Table 14.2: Evaulation of solutions to the problem Larman 67.

So we make Square abstract (allowing it to retain all the common information,) and create several specialised versions with distinct behaviour (for the landedOn() method). [67].

Note that this is essentially responsibility assignment, we've assigned the responsibility for doing something when the player lands on a square to Square and its subclasses.

Lets look at the Polymorphism pattern (14.3) and then look at the process by which it is used.

Pattern Name:	**Polymorphism**
Problem:	How to handle alternatives based on type? How to create pluggable software components?
Solution: (advice)	When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour – using polymorphic operations – to the types for which the behaviour varies. It follows (reversing the advice,) that we should not test for the type of object and use conditional logic to perform varying operations based on type.

Table 14.3: Polymorphisum principle/pattern based on Larman 67 using the less detailed format.

Its less clear whether this is a principle or a pattern, though the distinction is often blurred [65].

Software Objects and the Design Class Diagram - DCD .

Again we're working with software objects in the Design Class Diagram (DCD) this time we've got both classes in the DCD and some new ones from the current iteration of the object domain model.
This is the adapted from that described in Larman 67, using the same format as we've used earlier.

Step	In this example.
1. Identify and describe the problem:	*How to handle different types of square?*
2. Decide on a pattern to solve the problem:	*We've chosen* ***Polymorphism***
3. What happens here largely depends on when we spot the need for specialisation. • **From the object domain model – 1** we can promote square and its children (which show the different types). • **From the object domain model – 2** If we didn't spot the need for specialisations of square in the domain model (i.e. we're looking at taking out common behaviours/attributes) to create a parent we'll need to update this. • **From the DCD** we may have created a square in an earlier iteration and need to refactor (change,) or make this abstract and create child objects.	

We now update the diagrams (DCD and relevant sequence digrams,) I'm showing the diagrams from the first iteration and the updated one from the second iteration, Larman didn't need to solve this problem in the first iteration so he left it until much later in the book. First the before 14.5 and after 14.6 DCD.

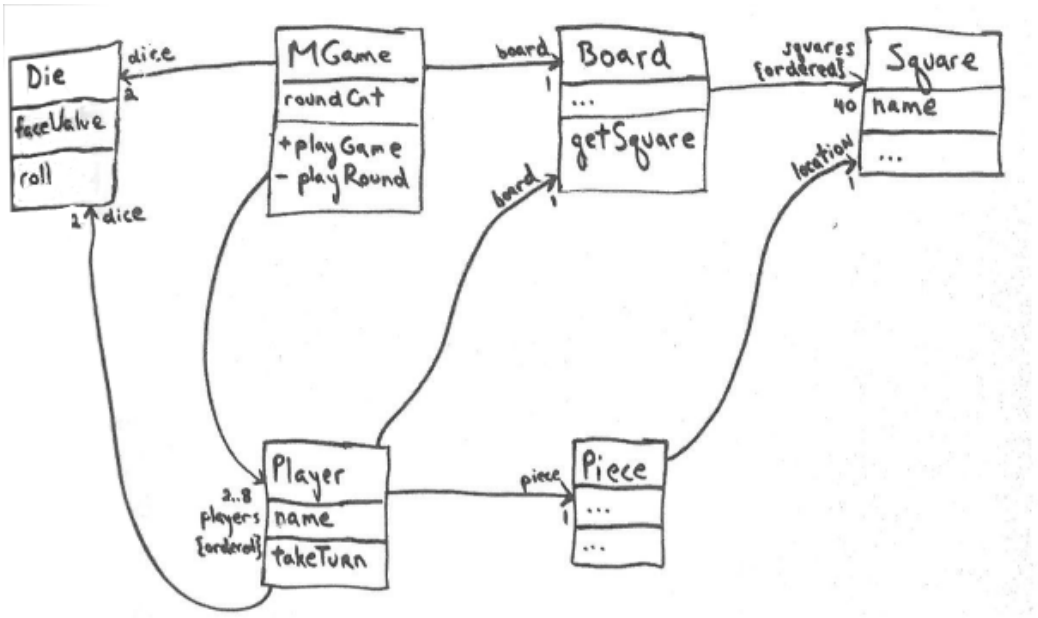


Figure 18.26 Static design for *playGame*.

Figure 14.5: This is an early version of the design class diagram (during its construction,) as the playGame use case is being realised, at this stage Larman has not considered what will happen when the Player lands on a Square [64].

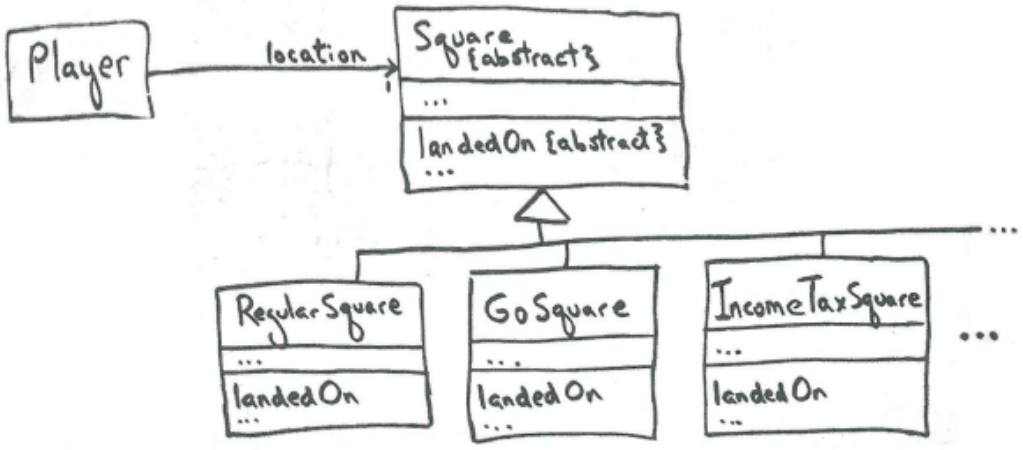


Figure 25.2 Applying Polymorphism to the Monopoly problem.

Figure 14.6: This is an later version of the design class diagram Larman has considered the need for Square to have different behaviours based on the type of Square and deployed Polymorphism to resolve this [67].

There are a few ways that we could consider the behaviour of the different types in our sequence diagrams, Larman 67 suggests that the clearest when modelling is to show the polymorphic behaviour of each derived (child) class in a different sequence diagram. The other option for this is an alt fragment.

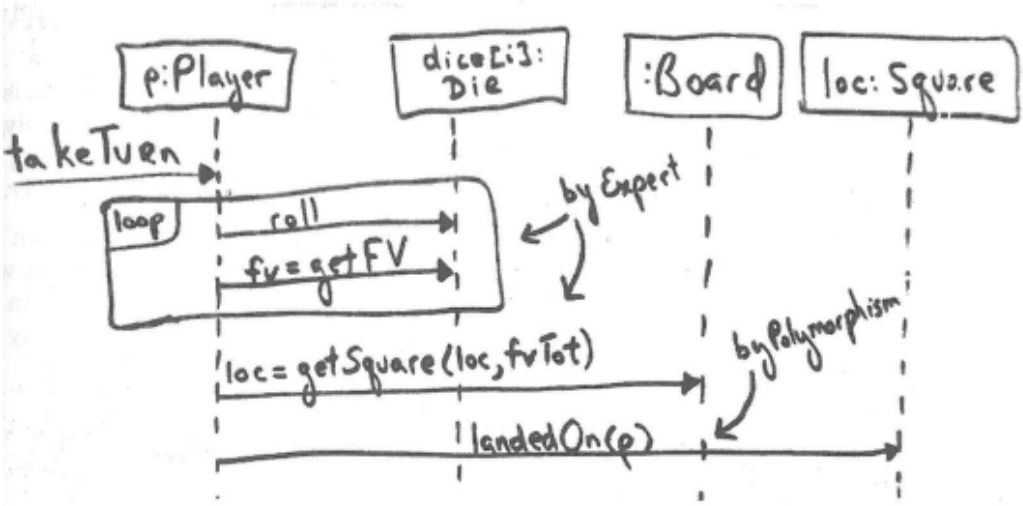


Figure 25.3 Applying Polymorphism.

Figure 14.7: A sequence diagrams showing the base class [67].

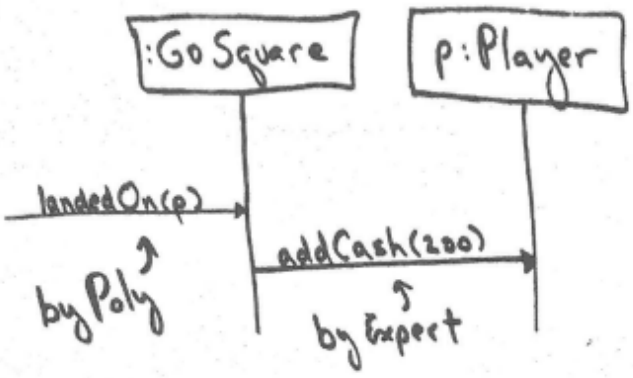


Figure 25.4 The *GoSquare* case.

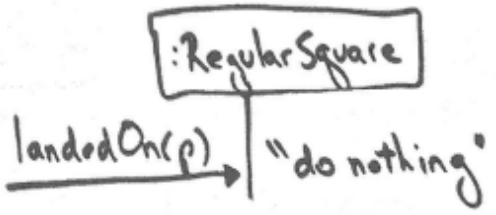


Figure 25.5 The *RegularSquare* case.

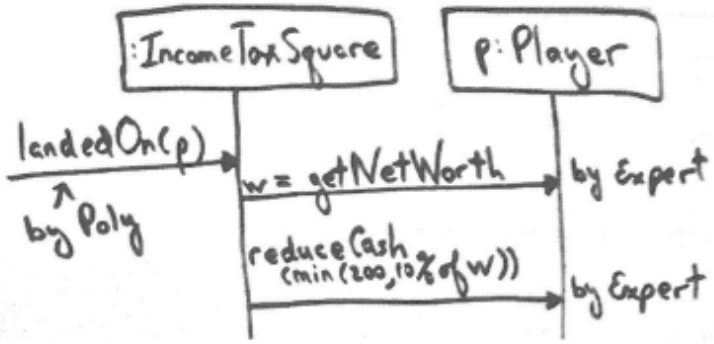


Figure 25.6 The *IncomeTaxSquare* case.

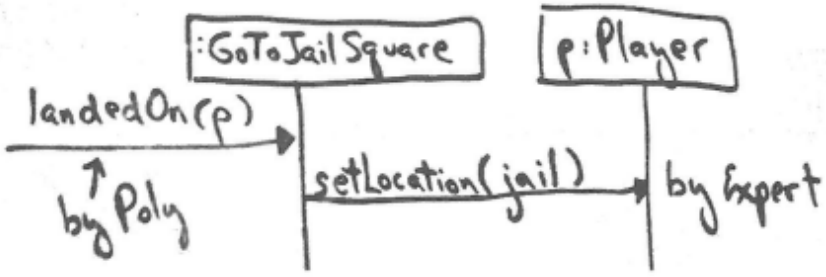


Figure 25.7 The *GoToJailSquare* case.

Figure 14.8: A sequence diagrams showing the base class [67].

14.3.4 The Command-Query Separation Principle

This is not a GRASP pattern, but turns up mid way though Chapter 18 of Larman's [64]. Its something we kind of know because its part of Low Coupling at the method level but seeing it described like this is really powerful and provides a justification for many of our earlier design decisions.
Larman 64 introduces this example at the end of the first iteration when considering the dice, he separates roll from getFace value and this leads to a discussion of the above principle. The listing 14.1 show roll() (a command) as separate to getFaceValue() which query the class for the result.

Listing 14.1: Command-Query Seperation - Good Example

```
//Assume these are correctly declared in the class
//and the relevant includes for rand are present.

void Die::roll()
{
    this->faceValue = (rand()%6)+1;
}

int Die::getFaceValue()
{
    this->return faceValue;
}
```

The Command-Query Separation Principle states that [64]

- A command method that performs an action (updating, coordinating, etc.,) often has side effects such as changing the state of an object, and is void (no return value); or
- A query that returns data to the caller and has no side effects should not permanently change the state of any objects (this one or those associated with it).

These are mutually exclusive, a method should never be both a Command and a Query [64]. The following sample 14.2 shows a badly designed version of the above (14.1).

Listing 14.2: Command-Query Seperation - Bad Example

```
//Assume this is correctly declared in the class
```

```
//and the relevant includes for rand are present.

int Die::roll()
{
    this->faceValue = (rand()%6)+1;
    return this->faceValue;
}
```

The final note here is that in this case with out this separation we cannot go back and check what the last roll of the dice was (should we need to).

14.4 More Class Diagrams

14.4.1 Generalisation

Generalisation is shown using a solid line with a large triangular arrow head pointed towards the superclass (or base class) [61]. The UML definition suggests inheritance, essentially stating that a taxonomic relationship exists and that each instance of the more specific classifier is indirectly an instance of the more general classifier [58]. This is not strictly inheritance, in the domain diagram it implies one concept is a more specific subset of another (the superset), however in the DCD (software-perspective,) it implies inheritance [58]. The example provided by [58] shows all the relations on one digram, I've created my own in 14.9.

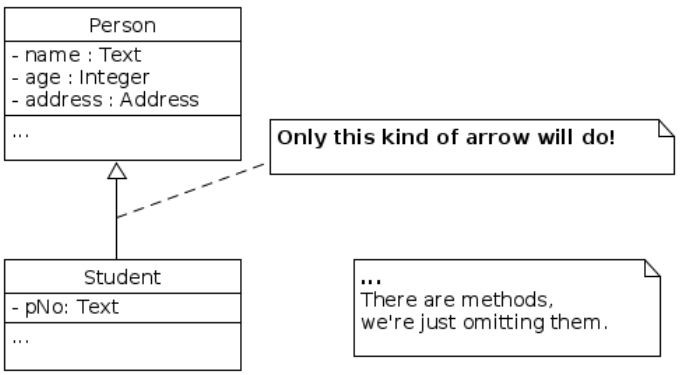


Figure 14.9: Inheritance in the Design Class Diagram.

Terminology .

There are a large number of terms used for this relation, I've tried to summarise the most common below.

More Generalised	More Specialised
Parent	Child
Base	Derived
Superclass	Subclass

Abstract Classes and Abstract Operations

Abstract classes can be shown using the tag *{abstract}* or by italicizing the name the former is clearly more useful while sketching while the latter is more common in tools [58]. UML (and some languages such as C#) allow for the opposite case, final classes and operations that can't be overridden in sub classes, these are shown with the *{leaf}* tag [58]. These are shown in the diagram below 14.10.

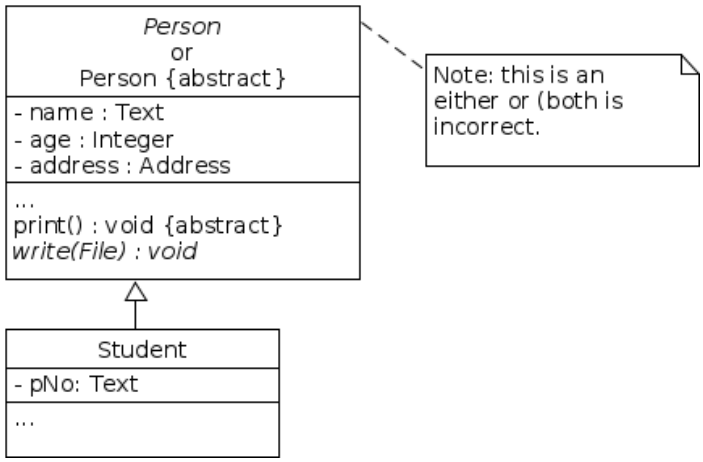


Figure 14.10: Abstract classes / methods in the Design Class Diagram.

14.5 More Sequence Diagrams

14.5.1 Interactions with Polymorphic Instances

Following on from the section on generalisation (**??**,) we need to consider how the interactions between these classes (notably when their behaviour deviates from the parent class,) will be represented. Larman 58 recommends showing these as separate diagrams each beginning with the found polymorphic

message [58]. This means the method call which is abstract in the superclass but overloaded in the subclass. This approach is shown in 14.11.

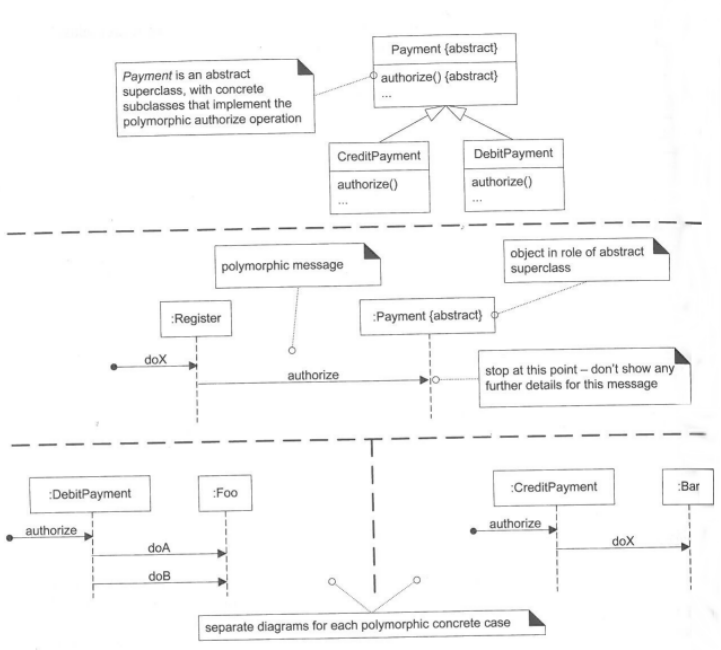


Figure 14.11: This is the what Larman 64 suggests, its especially good for sketching.

Larman 64 suggests an alternative is to make use of *ref* and *sd* frames see 14.12.

14.5.2 Relating Interaction Diagrams

Occasionally we need to relate interaction diagrams, this is usually to factor out common behaviour or simplify a complex diagram [58]. They are probably more common in UML tools which allow the diagrams to be related [58] however I've found them handy for spreading things across whiteboards or leaving something for later.

They are composed of two parts:

1. A **sequence diagram** (*sd*) in a box with a label in a five sided box (e.g. *AuthenticateUser*).

2. A **reference** (*ref*) that refers to another named sequence diagram; this is the actual interaction occurrence.

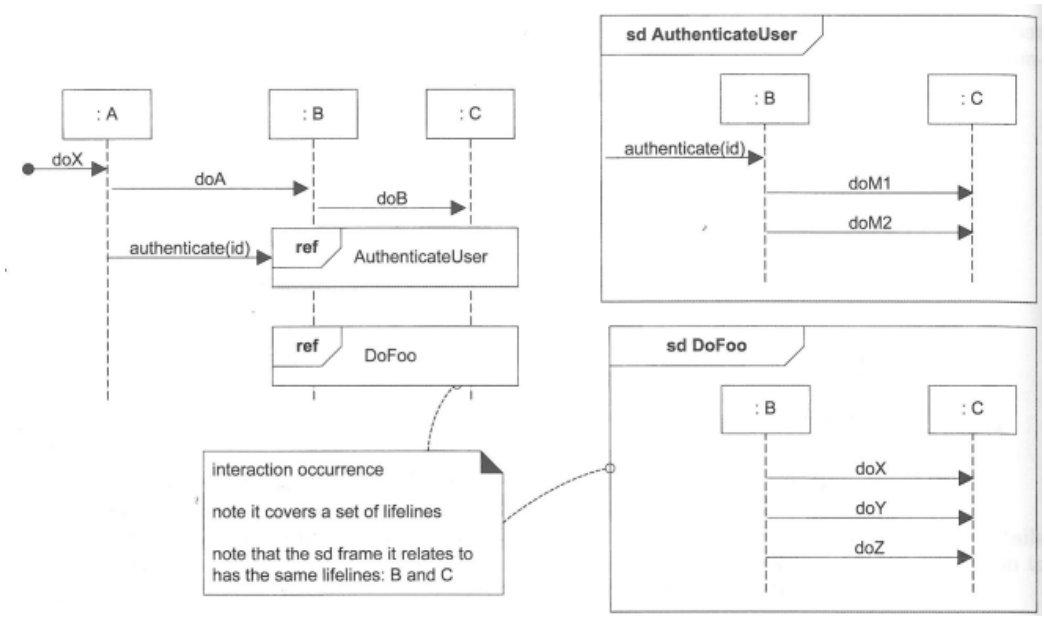


Figure 14.12: Example from Larman 58 showing the use of sd/ref.

Note: Larman 58 suggests only using the sd frame for sequence diagrams when you expect to reference it in another frame. I can see this is very agile, but other sources suggest that the correct syntax for a sequence diagram is to always have the box (and some tools enforce it).

Alternatively we may need to represent a polymorphic operation and achieve this by showing the operations is separate fragments (I've re-written Larman's example - shown in 14.11 using the alternate style 14.13

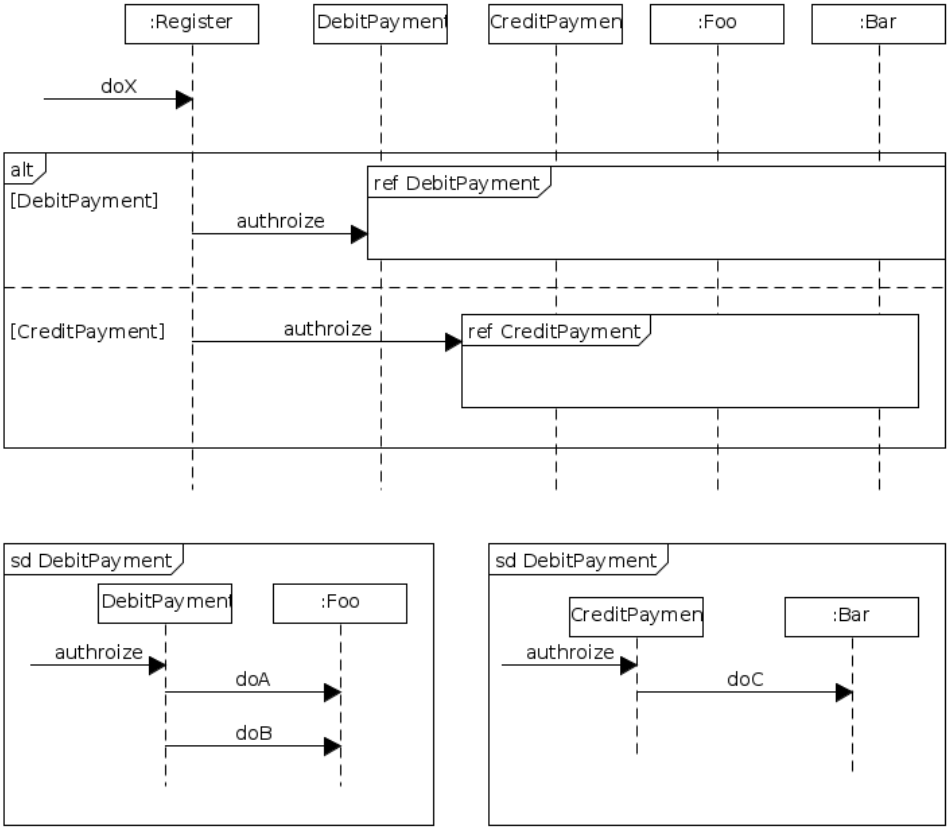


Figure 14.13: This is my copy of Larman 58 example shown in 14.11 using the alternative syntax.

Interaction Overview Diagrams

Interaction overview diagrams are made up of a large number of references to other sequence diagrams (interaction occurrences,) these can be used to orchestrate complex behaviour in large systems [58].

14.6 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 14.18 see the ODD 14.16 and DCD provided **??** and the SSD **??** then use an appropriate pattern to identify:

    - Who creates InputPattern?

- Who creates OutputPattern?
- Who can find an Input Pattern given an index?
- Who can find an Output Pattern given an index?
- How can we get print to work differently for input and output patterns but allow both to be treated as patterns?
- Who can load Input and Output Patterns from file, given a filename?

2. Read the case study **??**, and consider the use case **??**, domain object model **??** and SSD **??** then use an appropriate pattern to identify:
   - Who is the first object beyond the UI layer who receives the message in the SSD?
   - Who knows how to convert a measure between a given unit and another unit?
   - Who knows how to create a formatted output for a given unit?

Questions to think about (or what kind of exam question could I get on this).

- What are the GRASP patterns and why are they useful in the development of object orientated software?

14.6.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

Use Case

The team have produced a number of use case summarised at the highest level by the following use case diagram. The names of the use case here may differ slightly from those we identified during the tutorial.

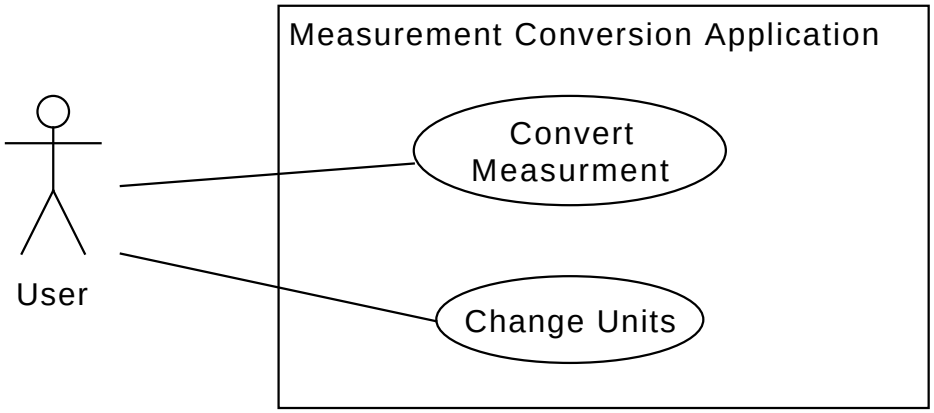


Figure 14.14: This is an incomplete use case based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough.

Use Case: Convert Measurement

UC5: Convert Measurement L14	
Scope:	Measurement Conversion Application
Primary Actor:	User
Level:	user-goal

Stakeholders and Interests:

- User: Wants to convert between imperial and metric measurements of length.

Preconditions:

- Application has launched and the interface is displayed
- Interface is prepared for user input

Postconditions (success guarantee):

- Output is presented in a suitable format

Main Success Scenario:

1. User enters a measurement in the default unit (imperial,)
2. User requests a conversion (indicating they have completed their data entry).
3. System performs the conversion
4. System displays the converted measurement

Extensions (Alternative Flows):

- 1a. User changes units before entering a measurement.
- 2a. User enters nothing
    1. System does not attempt to convert
    2. System highlights the lack of input (e.g. change background of input field to red)
- 2b. User enters an incorrectly formatted measurement for the current conversion.
    1. System Displays an error message
    2. System highlights the incorrect input (e.g. change background of input field to red)
    3. User enters a measurement

Special Requirements:

- Must be possible to enter the measurement in a number of formats, e.g. 1'6", 1.5", 18" etc.
- The output must be such that the user can easily copy the output for use in another application.

Technology and Data Variations List:

1. ...

Frequency of Occurrence:	Expect the user to perform one or more conversions in a single session

Open Issues:

- Given that the input can be in a number of formats, which the system is to deduce. How will this be done?
- The output may be required in a specific format, should this option be given to the user and how? This could be a property set though a menu (with a default,) or some means of selection near the output.

Domain Object Model

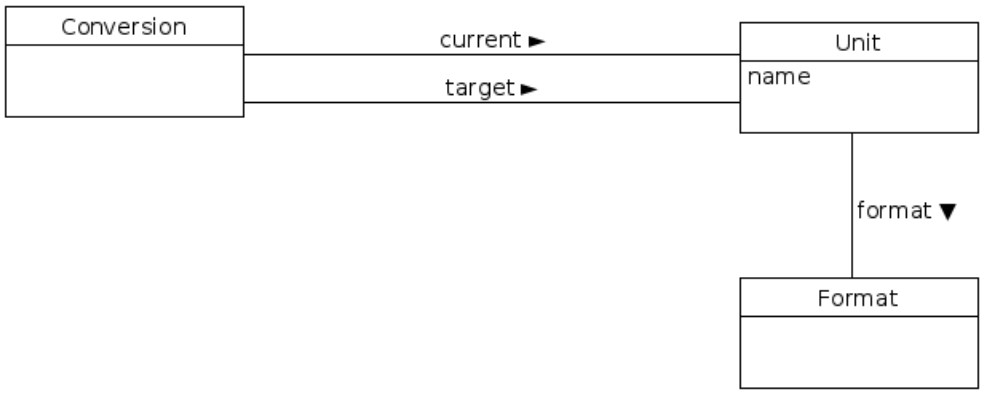


Figure 14.15: This is a domain object model based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough. I've avoided describing this as a draft (suggesting its incomplete,) or using any other term which would suggest it is final. In the spirit of Agile Modelling we are using it to inform sketches of the design so we can discuss and evaluate design options. Changes may be made as a result of this discussion and its probably that future iterations would contain some of the conceptual classes.

14.6.2 Case Study 2

A researcher at the university is looking to write a paper with a colleague they have both datasets and a new neural network to train on them. In order to do this they need to load in files from the original project so that their Neural Network will be trained and tested with the same files. Each file consists of comments (which start with a #,) some meta data (information about the file content) and the data itself (see example file), all in a CSV format. The data is in the form of input and output patterns, each contains t (a time offset). The input contains the values for parameters t, r, g, and b while the output patter has two values t and o. Due to the nature of neural network operations all parameters are floating point numbers between -1.0f and 1.0f further validation is possible as the t (which represent the time-step,) should be identical in the input and output pattern.

14.6.3 Domain Object Diagram for Case Study 2

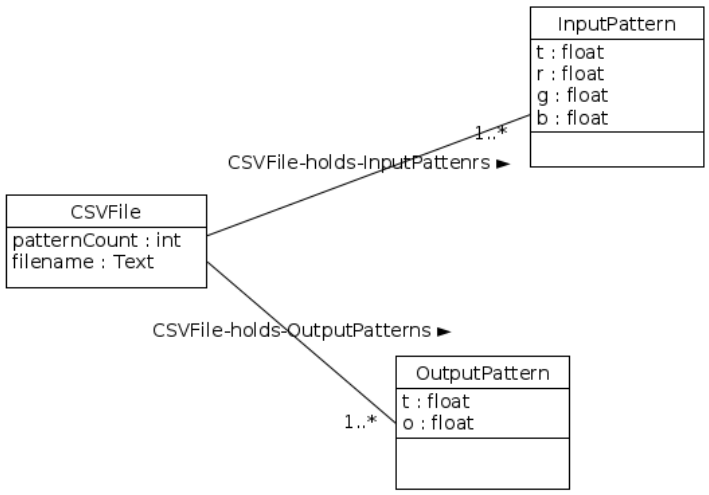


Figure 14.16: Domain Object Model for Case Study 2.

14.6.4 System Sequence Diagram for Case Study 2

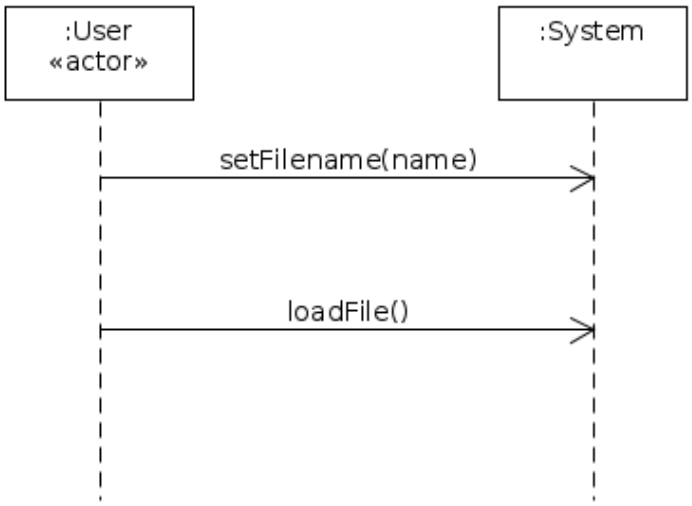


Figure 14.17: SSD for Case Study 2.

DCD for Case Study 2

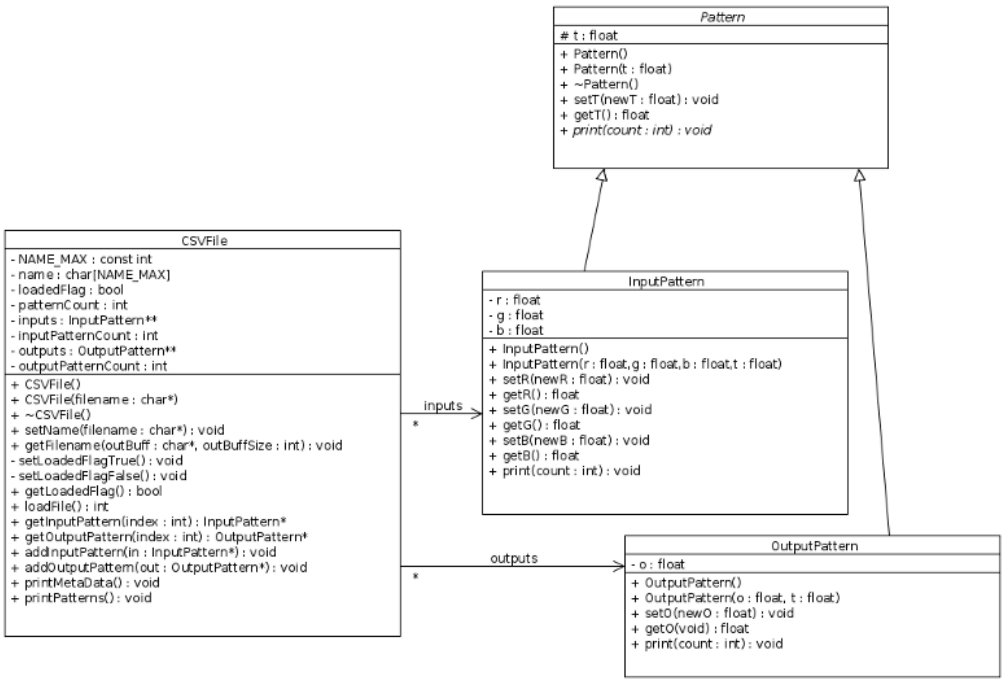


Figure 14.18: I happen to have this as I set is as a coursework, to end up with this we'll need one pattern I've not covered yet!.

14.7 Summary

The next important part of the design process is the Realisation of the use case, which is the mapping of use case and individual scenarios to object behaviour. We've started small considering parts of these interactions in isolation and the low level patterns (or OO principles,) we've now considered all the most useful patterns. We'll come back to the remaining ones if there is time.

The GRASP patterns are low level simple design patterns/principles named and documented to make them easier to remember, discuss and use. This week we've looked at Pure Fabrication and Polymorphism with some examples of their application, however we've not considered the pattern in full (i.e. with a full discussion of their pros and cons) at the moment we need to see a few more patterns so we can start using them in the realisation of use case. I've also included the Command-Query Separation Principle, this is not part of GRASP but is another useful pattern which I think will help.

14.8 Preparation for Next Week

14.8.1 Essential Reading

An introduction to GRASP and Patterns, this covers this week and last weeks material..
Larman (2014q). *Applying UML and Patterns*, chapter GRASP: Designing Objects with Responsiblities, pages 312–362. Prentice Hall PTR.
For next week read as much of chapter 18 as you can! I'll not be going into as much detail (or doing as many examples as Larman)!
Larman (2014p). *Applying UML and Patterns*, chapter Object Design Examples with GRASP, pages 312–362. Prentice Hall PTR.

14.8.2 Background Reading

In conjunction with the above you may need to refer to these chapters which provide very detailed coverage of Class and Sequence diagrams. Larman (2014j). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

Larman (2014m). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Designing with Patterns/Principles: Mechanistic Design - Part 4

15.1 Introduction

In the previous notes (14) we were considering object design and were started (more by accident than design,) the use case realisation. In an attempt to systematically design object orientation systems we adopted a responsibility driven design approach (RDD). This suggested the use of GRASP patterns in order to assign low level responsibilities and we'd looked at almost all of these these patterns, again more by accident than design as they turned out to be required by the 'simple' case studies I'd created.

15.2 Use Case Realisation

We're moving on to Use Case Realisation or perhaps more accurately scenario realisation, the process by which we describe the design in terms of objects and their interactions [64]. The term Use Case Realisation comes from UP and is used to reinforce the connection between the use case in analysis and the design of the system [64].

Larman 64 argues that object orientated design is simply a question of assigning responsibilities to objects and designing collaborations and that both can be logically rationally explained and learned. Also that mastering OO design requires the deployment of a large number of soft principles (with many degrees of freedom,) this seems daunting but these principles can be named and learned.

15.2.1 Orientation: What's next?

The plan is as follows:

- **Realising Use Case with GRASP** We're currently looking at deploying what we've covered so far with GRASP in the realisation of some use case. This will give us some functioning code.
- **Implementation considerations** We'll look at how we write code as a collective, including version control and automated testing (maybe some continuous integration) if I can get the server set-up in time.
- **GoF Patterns** We'll the move on to look at some (or perhaps all) of the Gang of Four patterns, these are well know and well used design patterns which will give us a flavour for their use.

We've been working up to this bit realisation, starting small with the GRASP - small patterns (or principles,) these should be enough to get us started.

15.3 Applying GRASP to Realise a Scenario/Use Case

This example is based heavily on that of Larman 64 its one of the few sections I'd have happily given you without modification but only the old version is available as a sample chapter. If you'd like to read the un-edited version it starts on page 349.

15.3.1 Monopoly Case Study: Reminder

This case study is based on Monopoly, in this case the initial version allows players to watch only without interaction the game plays itself for a set number of rounds [64].

The first iteration of elaboration implements the use case **Play Monopoly Game** it has only two system operations, startUp (which is implicit for all systems,) and playGame. Larman 64 suggests that it is good practice to leave initialisation until last. Exploring the system other operations first gives us a DCD and therefore a clear picture of what we need to initialise.

15.3.2 The Design of playGame()

The playGame system operation occurs when the human player (though the UI) requests that game simulation (its not interactive in this iteration,) begins so it can be observed [64]. Larman omits much of the analysis for the Monopoly case study as the mode of play is well known.

NOTE: What follows is a huge amount of discussion and reasoning about the design of a very small chunk of system behaviour. Its meant to show how possible design decisions are evaluated in the mind of a developer.

Before going on lets remind ourselves of the SSD for this system operation 15.1.

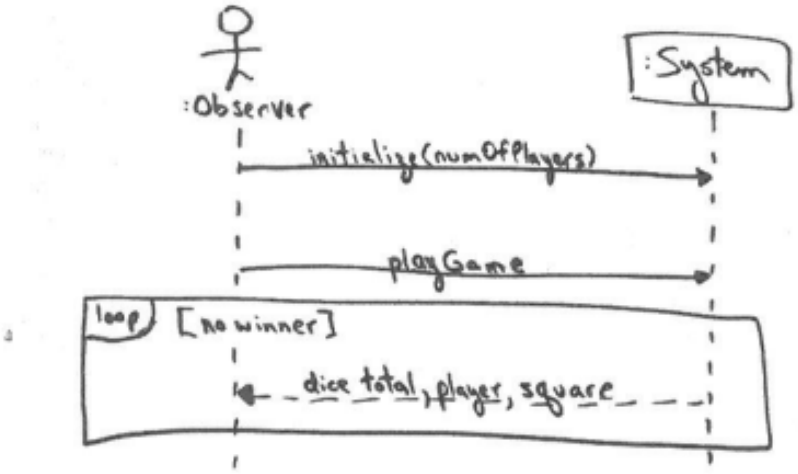


Figure 17.8 SSD for the Monopoly game. Note the *playGame* operation.

Figure 15.1: The SSD for playGame() from Larman 65.

Also lets look back at the domain object model, we'll refer to this a lot 15.2.

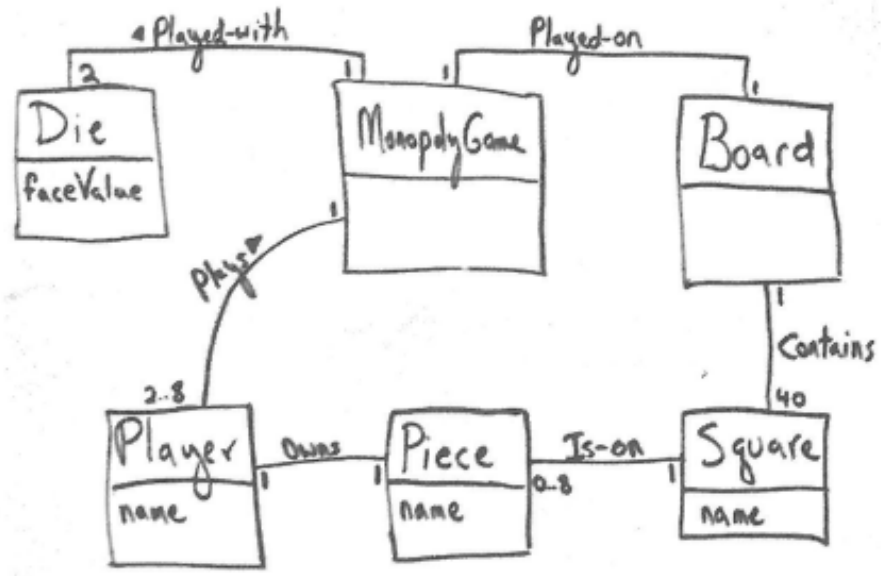


Figure 15.2: The domain modelfrom Larman 68.

Choosing the Controller Class

Our first design choice is involves selecting the controller for the *playGame* message entering the domain layer from the UI [64].

The **Controller** Pattern suggests:

Advice	***Candidate* and Explanation**
An object representing the overall "system" or "root object" (this is a variation of the *facade controller*)	*MonopolyGame* A kind of root object we think of most other objects as being contained within the MonopolyGame.
A major sub system (another variation of the *facade controller*)	*MonopolyGameSystem* A name suggested the overall system
A use case scenario within the system within which the operation occurs (a use case or session controller).	*PlayMonopolyGameHandler* *PlayMonopolyGameSession* A name constructed from the pattern <use-case-name>Handler or <use-case-name>Session

Choosing a root-object facade controller like MonopolyGame is satisfactory only if this does not violate High Cohesion (i.e. there are only a few system

operations and these are reasonably small in scale) [64].

We may represent this decision in our whiteboard sketch as shown in 15.3.

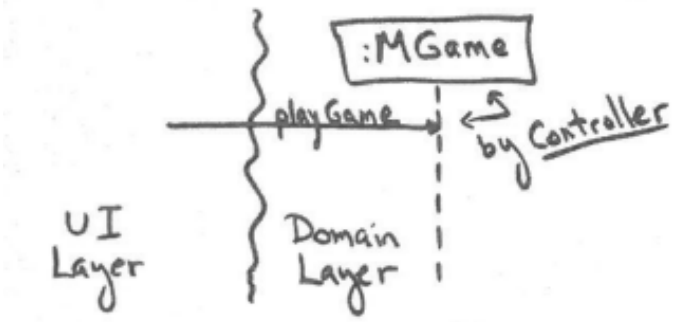


Figure 15.3: Assigning the responsiblity for PlayGame by Controller Larman 64.

The Game-Loop Algorithm

For this iteration the mode of play is greatly simplified, we have N rounds where each player takes a turn (rolling the dice and taking a turn) [64].

Who is Responsible for Controlling the Game Loop?

As we're doing RDD (Responsibility Driven Design,) we need to work out where who takes responsibility for this important logic [64]. This is clearly a doing rather than a knowing responsibility, its also not a creation or a controller problem [64]. Expert is the obvious choice, for this we need to work out the information required and who has this information to identify candidate classes [64]. We can then evaluate these in terms of cohesion and coupling (if more than one alternative exists), the following table **??** is based on that provided by [64].

Information Required	**Who (which object,) Has the Information?**
The current round count	None yet, but by LRG, assigning this to MonopolyGame would seem reasonable.
All the players, so that each can take a turn	Taking inspiration from the domain model 15.2 MonopolyGame is a good candidate.

Larman 64 concludes that MonopolyGame is a good candidate, (see 15.4), this also shows of a private helper method playRound.

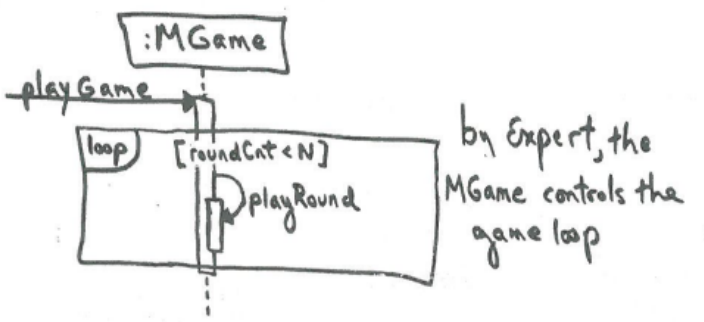


Figure 15.4: Assigning the responsiblity for PlayRound by Expert Larman 64.

Private Helper Methods .

The use of private helper methods has two key benefits [64]:

- Can be used to organise cohesive chunks of behaviour, e.g. playRound in the above. This supports high cohesion at the method level.
- Can connect these cohesive chunks to the domain vocabulary, e.g. playRound.

Who Takes Turn?

As mentioned earlier taking a turn involves rolling the dice (working out the total,) and moving the piece which represents the player. So this is another behavioural ('doing') responsibility "Which object should be responsible for taking the turn of a player?" [64].

Again Expert will apply but this time the situation is more complex.

Guideline: DO NOT assign responsibilities to objects as they are assigned to people. Larman 64 suggests that the naive reaction is to go for player, as in the real world its player who takes a turn. He goes on to stress that in OO the designs are not necessarily a one-to-one simulation of the real world, this tends to lead to very large objects as in the real-world lots of things are done by people. This often leads to very large objects as in most systems the user is the driving force and the objects who should represent them can take over, a violation of high cohesion (leading to a

violation of low coupling) [64].

In this example, player turns out to be a good choice. However, lets again evaluate the candidate objects and the information they have, the table below is taken from Larman 64 with with some formatting tweaks.

Information Required	**Who (which object,) Has the Information?**
Current location of player i.e. the starting point of a move.	*Piece* knows its *Square* and *Player* knows its Piece. Player could therefore know its location by LRG.
The two die objects, to roll them and calculate their total.	From the domain model, MonopolyGame is a candidate, as we think of dice as being part of the game.
All the squares - their organisation (so we can calculate the correct new square i.e. given the dice roll).	By LRG Board is the obvious candidate.

NOTE: We've been using this example to look at GRASP so we've got some insight into the application of other patterns to these objects. We know for example that Board will create Square (by Creator 12.7.4) and that Board can find Squares (by Expert 13.4.2). We also know that part of the above (calculating the total of dice,) was factored out of player in a later iteration by applying Pure Fabrication 14.3.3. We'll ignore that for now and working though the reasoning as if its for the first time.

So going back to **??**, we have a problem, there are a number of partial experts! Reasoning about these requires some additional guidance and (an issue for us,) some experience, however Larman 64 provides some guidance note to help with the decision.

Guideline: When faced with multiple partial information experts:

1. Choose the one with the most information, i.e. the dominant expert. Doing so should support low coupling.

2. If no clear dominant expert, consider alternative design choices in terms of cohesion and coupling.

3. If no clear winner from the above consider future evolution, i.e. how will future iterations effect the information objects have, cohesion and coupling.

The following evaluation is based on Larman 64, I've tried to put it into a less verbose format.

Guideline	**Evaluation**
1.	They are unfortunately all pretty equal.
2.	Monopoly game is pretty busy, where as the other two classes have little to do. This suggests we should select either Player or Board.
3.	In future iterations taking a turn will involve buying properties, checking for properties of this colour already owned etc., all of which by LRG would be stored in player.

The conclusion from this is to that reduce future coupling we should select the player object as being responsible for taking a turn, 15.5 summarises the emerging dynamic design [64].

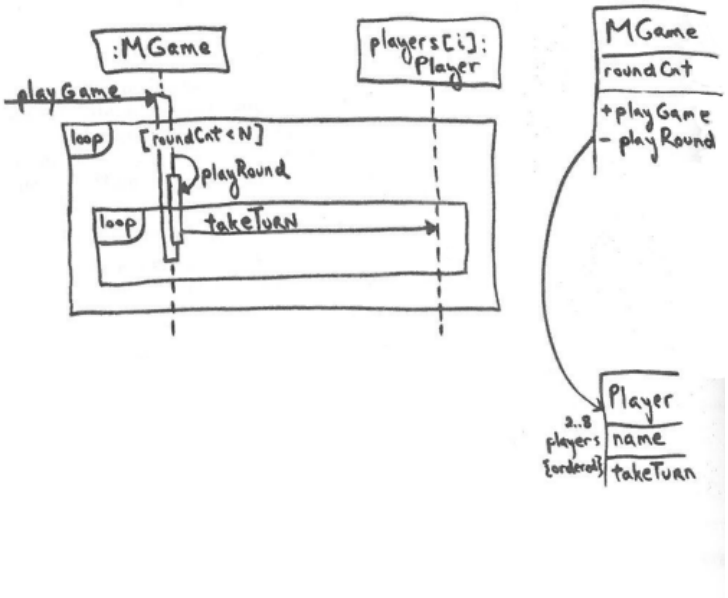


Figure 15.5: Assigning the responsiblity for takeTurn by Expert Larman 64.

Taking a Turn

Earlier we noted that playing a round involved N turns, Larman 64 decomposes turn into the following steps:

- Calculate a random number between two and 12 (the range of two dice).
- Calculate the new square location (based on the current square).
- Move the players piece from the current square to the new one.

There are clearly a few problems here [64]:

The Random Number Problem: By **LRG**, we'll promote the *Die* object from the domain model to the class model to represent the *Die* object and hold the faceValue attribute [64]. By **Information Expert** we assign the roll behaviour to *Die* (this is essentially the generation of a random number, by **LRG** we use the domain language roll) [64]. Essentially we have made *Die* responsible for knowing and generating (doing the generation of,) its face value.

The New Square Location Problem: By **LRG** its reasonable that *Board* knows all its *Squares* so by **Expert** its reasonable a *Board* will be responsible for calculating a new square given an old one and some offset (the dice total) [64].

The Piece Movement Problem: By **LRG** it seems reasonable that *Player* will know its *Piece* and *Piece* its *Square*, alternativly *Player* could know its *Square* [64]. By **Expert** piece will have to change its location but this may be calculated by its owner *Player* [64].

The Coordination Problem: All this activity has to been coordinated, the question here has already been answered "Who coordinates all this?" we decided *Player* takes a turn so he coordinates [64]. Note at this stage if we were further along and *Player* had started to do a large number of things, we may be looking to use **Pure Fabrication** and create a class to include this and other similar behaviour.

The Visibility Problem: Wait, what's visibility? **?** **?** dedicates a whole chapter to visibility design, essentially there are three flavours:

- Attribute Visibility - B is an attribute of A.
- Parameter Visibility - B is a parameter of a method of A.
- Local Visibility - B is a (non-parameter) local object in a method of A.

In UML we the first of these as an association (see 12.8.1) and the others as dependencies (see 13.5.1) [58].

Back to the problem at hand, the *Player* coordinating the activities of Board, Die and Piece suggests that at start up we can usefully initialise Player with permanent references (associations) to these at start-up [64].

Final design of playGame - at this stage

Based on the above design decisions the following dynamic 15.6 and static 15.7 diagrams emerge (from Larman 64).

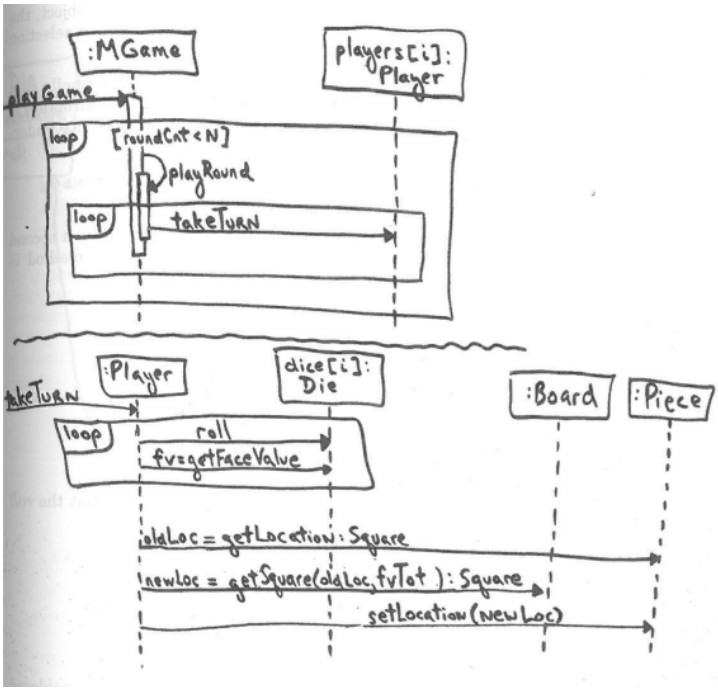


Figure 15.6: The final dynamic design for playGame Larman 64.

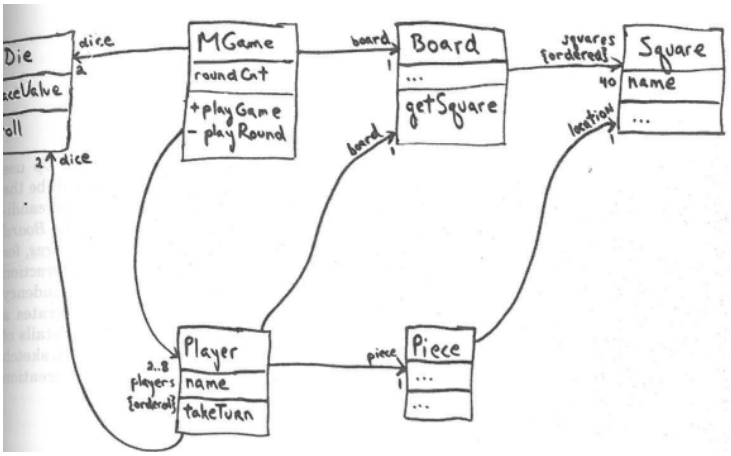


Figure 15.7: The final staic design for playGame Larman 64.

Note: The key point here (and one which [64] also notes,) is that we've methodically assigned responsibilities using the GRASP principles to make decisions about the design and to evaluate alternatives. **There is no magic here, its a methodical process which we get better at with experience.**
Note: There are two further improvements to be made which we've already discussed as examples when considering **PureFabrication** 14.3.3 and **Polymorphism** 14.3.3

15.3.3 Initialisation and the Start-up Use Case

In an earlier section **??** we mentioned an implicit Start Up use case which takes responsibility for creating the class structured for the current iteration. This will be continually augmented to include the creation of who whole system structure as we progress. While working on this iteration we create a creation dependency diagram (deploying the Creator 12.7.4 pattern), to record our decisions as to who creates who this is shown below 15.8.

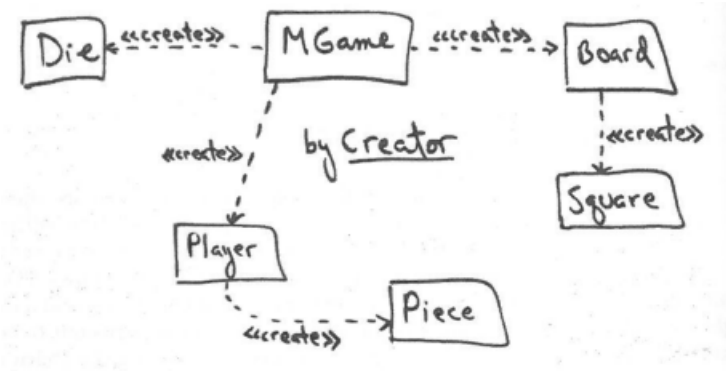


Figure 15.8: The creation hirarchy, created by deploying the creator pattern for the various objects in our DCD Larman 64.

15.4 Additional UML

15.5 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Read the case study 15.13 see the ODD 15.11 and DCD provided **??** and the SSD **??**, the use case being realised is "Load Training File".

    - Produce a design for the system operations shown in the SSD (note we leave the implicit "Start Up" until last).

2. Read the case study **??**, and consider the use case **??**, domain object model **??** and SSD **??**, the use case being realised is "Convert Measurement":

    - Produce a design for the system operations shown in the SSD (note we leave the implicit "Start Up" until last).

Questions to think about (or what kind of exam question could I get on this).

- At this point your likely to see questions like the above in the exam, or parts of them at least.

15.5.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from

meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is negative. The application reports a formatting error, but does not clear the input field allowing the user to make corrections.

Use Case

The team have produced a number of use case summarised at the highest level by the following use case diagram. The names of the use case here may differ slightly from those we identified during the tutorial.

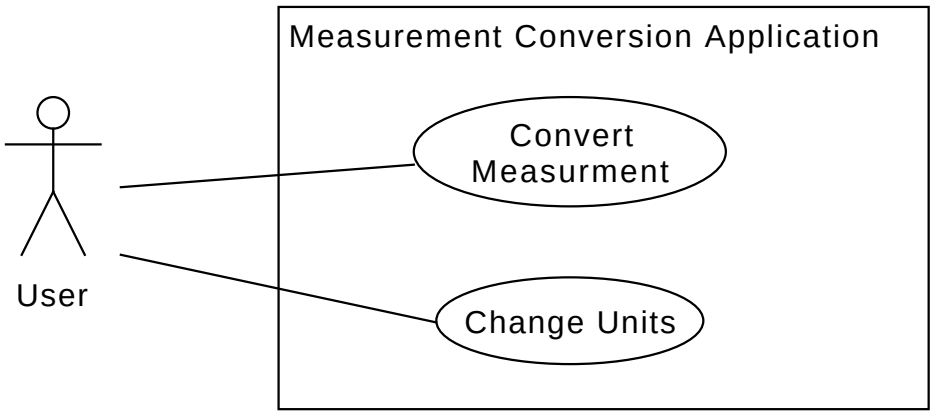


Figure 15.9: This is an incomplete use case based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough.

Use Case: Convert Measurement

UC6: Convert Measurement L15	
Scope:	Measurement Conversion Application
Primary Actor:	User
Level:	user-goal

Stakeholders and Interests:

- User: Wants to convert between imperial and metric measurements of length.

Preconditions:

- Application has launched and the interface is displayed
- Interface is prepared for user input

Postconditions (success guarantee):

- Output is presented in a suitable format

Main Success Scenario:

1. User enters a measurement in the default unit (imperial,)
2. User requests a conversion (indicating they have completed their data entry).
3. System performs the conversion
4. System displays the converted measurement

Extensions (Alternative Flows):

- 1a. User changes units before entering a measurement.
- 2a. User enters nothing
    1. System does not attempt to convert
    2. System highlights the lack of input (e.g. change background of input field to red)
- 2b. User enters an incorrectly formatted measurement for the current conversion.
    1. System Displays an error message
    2. System highlights the incorrect input (e.g. change background of input field to red)
    3. User enters a measurement

Special Requirements:

- Must be possible to enter the measurement in a number of formats, e.g. 1'6", 1.5", 18" etc.
- The output must be such that the user can easily copy the output for use in another application.

Technology and Data Variations List:

1. ...

Frequency of Occurrence:	Expect the user to perform one or more conversions in a single session

---

Open Issues:

- Given that the input can be in a number of formats, which the system is to deduce. How will this be done?
- The output may be required in a specific format, should this option be given to the user and how? This could be a property set though a menu (with a default,) or some means of selection near the output.

Domain Object Model

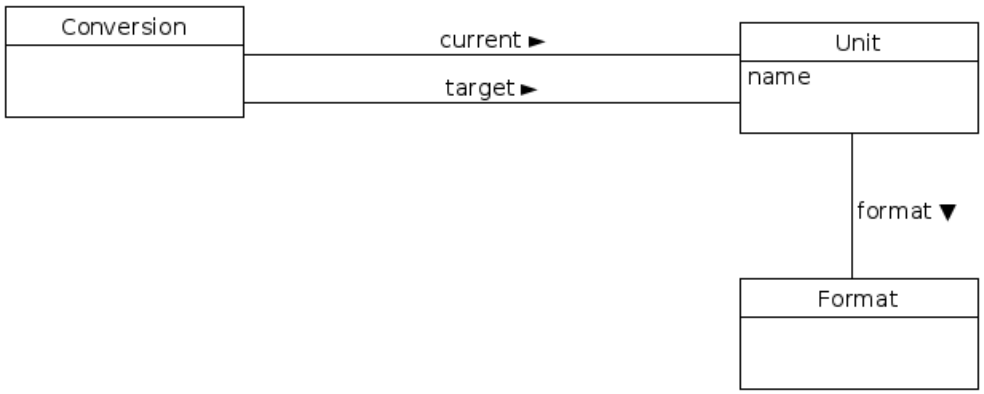


Figure 15.10: This is a domain object model based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough. I've avoided describing this as a draft (suggesting its incomplete,) or using any other term which would suggest it is final. In the spirit of Agile Modelling we are using it to inform sketches of the design so we can discuss and evaluate design options. Changes may be made as a result of this discussion and its probably that future iterations would contain some of the conceptual classes.

15.5.2 Case Study 2

A researcher at the university is looking to write a paper with a colleague they have both datasets and a new neural network to train on them. In order to do this they need to load in files from the original project so that their Neural Network will be trained and tested with the same files. Each file consists of comments (which start with a #,) some meta data (information about the file content) and the data itself (see example file), all in a CSV format. The data is in the form of input and output patterns, each contains t (a time offset). The input contains the values for parameters t, r, g, and b while the output patter has two values t and o. Due to the nature of neural network operations all parameters are floating point numbers between -1.0f and 1.0f further validation is possible as the t (which represent the time-step,) should be identical in the input and output pattern.

15.5.3 Domain Object Diagram for Case Study 2

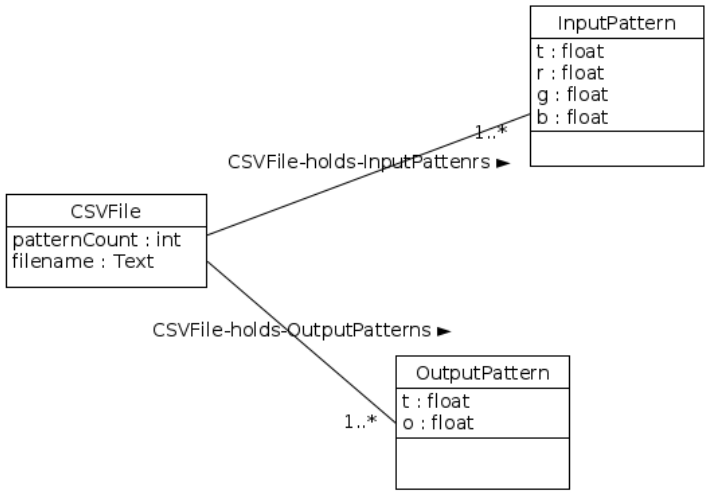


Figure 15.11: Domain Object Model for Case Study 2.

15.5.4 System Sequence Diagram for Case Study 2

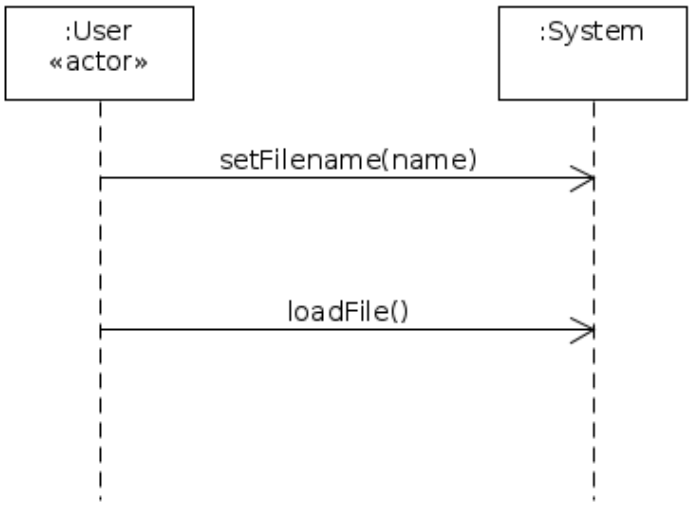


Figure 15.12: SSD for Case Study 2.

DCD for Case Study 2

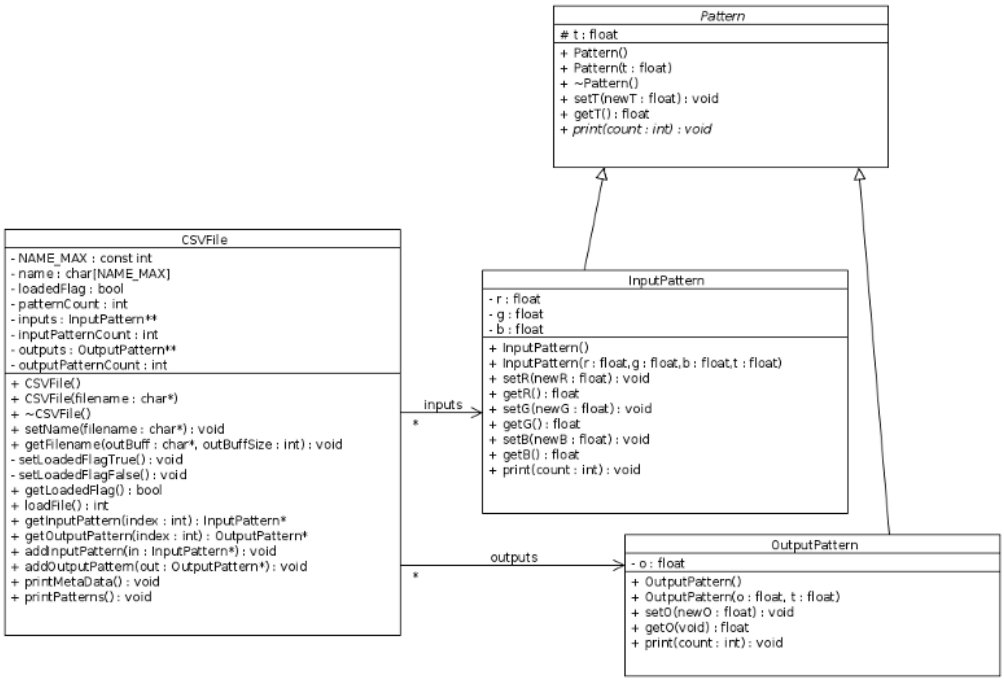


Figure 15.13: I happen to have this as I set is as a coursework, to end up with this we'll need one pattern I've not covered yet!.

15.6 Summary

Realisation of the use case (the mapping of use case and individual scenarios to object behaviour) has been introduced in terms of methodically deploying very simple principles and patterns (GRASP).

The GRASP patterns are low level simple design patterns/principles named and documented to make them easier to remember, discuss and use. I've introduced almost all of these along with a few others I think are worth knowing (while trying not to overload you). We're now going to look at how we turn these designs into code and testing, this will allow us to complete an iteration (primarily for the group project). We'll then take a break from the low level activities to consider professional issues in software development before entering another iteration - adding to the first - with more complicated patterns (the GoF).

15.7 Preparation for Next Week

15.7.1 Essential Reading

Read Larman's more detailed descriptions of the GRASP patterns
Larman (2014q). *Applying UML and Patterns*, chapter GRASP: Designing Objects with Responsiblities, pages 312–362. Prentice Hall PTR and Larman (2014s). *Applying UML and Patterns*, chapter GRASP: More Objects with Responsiblities, pages 401–343. Prentice Hall PTR.
Read the other case study (POS) from:
Larman (2014p). *Applying UML and Patterns*, chapter Object Design Examples with GRASP, pages 312–362. Prentice Hall PTR.

15.7.2 Background Reading

In conjunction with the above you may need to refer to these chapters which provide very detailed coverage of Class and Sequence diagrams. Larman (2014j). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

Larman (2014m). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Building the Implementation Model

16.1 Introduction

In the previous notes (11 to 14) we were considering object design and use case realisation. To systematically design object orientation systems we adopted a responsibility driven design approach (RDD). This suggested the use of GRASP patterns in order to assign low level responsibilities. We're moving on to consider the code itself, the UP describes this as the **Implementation Model** which refers to all implementation artefacts [69]. This includes code, database definitions, JSP/XML/HTML pages etc. The input for these are the artefacts created during the design work-flow i.e. DCDs and Interaction (sequence) diagrams.

The plan is for a short class, focusing on turning the design into code and the practical issues surrounding programming as a team.

16.2 Where are we?

In the development case is shown in 16.1 (from Larman 49,) we're now moving to the Implementation discipline (work-flow).

16.3 Programming and OOD

The analysis and design work which has proceeded this section may suggest that there has been no programming to date, this is not the case and prototyping and design-while-programming may have been employed [69]. Modern programming environments and external tools provided excellent facilities to explore and refactor to alternative solutions [69]. I reiterate

Discipline	Artifact Iteration-*	Incep. I1	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 2.1 Sample Development Case of UP artifacts, s - start; r - refine

Figure 16.1: The Sample Development Case from Larman [49].

here that when we write code without first sketching a design we are still designing, we're working out how it works before we write any code.

The motivation for using OOA/D (Object Orientated Analysis / Design) is that it provides an end to end road map from requirements to design, with artefacts feeding into later artefacts allowing traceability [69].

16.3.1 Creativity and Change

During the design work a number of key decisions will be made and to some extent (for simple examples,) the conversion to code is fairly mechanical [69]. In more complex real world examples this is rarely the case and the design models generated are more a incomplete first step, programming and testing reveal uncover problems which are resolved though changes in the design [69].

This seems to undermine the need for design however, its important to note that the ideas and understanding generated during the design will provide a sound and extensible base for development [69]. There will be a fair amount of change during programming, which we'll need to manage however our work in the design phase will hopefully minimise these [69]. As students you should expect to make some changes, however, the assignment is reasonably simple and I'd expect you're design to get you fairly close to a working prototype.

16.4 Mapping Designs to Code

We've previously looked at writing object orientated code and should be familiar with defining the class, its attributes and methods and implementing these (year one Introduction to Programming). This is should be reasonably straight forward, however Larman 69 makes some important observations which I'll highlight.

16.4.1 Creating Class Definitions from DCDs

The DCD should provide us with class names, attributes and operation signatures sufficient to create a basic class definition, we may also include in our DCD interfaces, superclasses etc. [69]. Most UML modelling tools will generate a basic class structure from this information, [69] some of the more advanced ones will keep this diagram up to date as edit the code [70].

16.4.2 Creating Methods from Interaction Diagrams

Sequences of messages in the interaction diagram provide us with methods which which the class will provide (along with its parameters and returns) [69]. Private methods can be derived from self calls, while overloaded constructors are suggested by create/make type messages being passed to a class. An interesting note here is that where we have the potential for failure a method may be a better option than an unguarded constructor (as error handling is more robust).

16.4.3 Misc. Guidance

Larman 69 mentions a few other points in this section which are worthy of note.

Collection Classes

One to many relationships can be implemented in a number of ways including the use of collection classes (e.g. List, Map, Vector etc.) [69]. Thinking this though we are essentially increasing cohesion, the responsibility of manipulating the group of elements is being given to another class.

Order of Implementation

We need to implement and fully test our classes and to do so effectively we attempt to test code we've written as soon and as often as possible. Lar-

man 69 suggests moving from least-coupled to most-coupled, UML to some extend makes this easy as the arrow (inheritance or directed association) points towards the parent/associated class. So we start with the classes who have only arrows pointing to them (and the fewer the better,) implementing them and testing them, then move to those who use only the classes we've implemented. In terms of inheritance I suggest base class (and if abstract,) a single derived class in order to fully test the base before continuing.

16.5 Test Driven Development

Test-driven (or TDD,) or test-first development was originally promoted as part of the Extreeme Programming method (as proposed by Beck,) [69], its also a recommeded practice in Crystal Clear (developed by Cockburn,) and common practice in many Scrum projects [44]. It can be applied to Agile UP as well as a number of other iterative methods [69]. The essential work-flow is that the test is written before the code, so a number tests which check that the addItem(prod, quantity) system operation works as it should for various products and quantities would be produced, e.g. checking that the items are now part of the sale and can be accessed etc. Then the code for the operations is written, if helper (private,) methods are required these are also generated.

This is a deceptivly simple interpretation of TDD as it transforms the testing process and the role it plays in the creation of software [44]. Rather than focusing on keeping software free from defects testing becomes more about understanding what makes the system correct and delivering those features [44]. Writing the tests first essentially makes the tests part of the analysis process, we work out what the system has to do and make a test to see that is is done and that all the possible deviations are considered [44]. This allows the development of a suite of regression tests which allow additions, modifivations, changes etc. to be tested as the system is developed [44]. Building on what Larman suggests above Freeman and Price 44 suggest that the TDD work-flow is - write a failing unit test, write the code to make it pass, improve the code (re-factor,) until your happy with it.

I'm not going to consider TDD in detail at this stage but we'll return to some elements of it later. I suggest anyone who has an interest reads the relevant section of Larman's book ... actually everyone should probably read the bit on refactoring as it shows how to use the tools to reduce coding effort.

Larman (2014w). *Applying UML and Patterns*, chapter Test Driven-Development and Refactoring, pages 385–395. Prentice Hall PTR.

16.6 Team Based Development

Developing software in the real world often necessitates working in teams on the same code base. Tools to support this have been in development almost as long as software itself, the history is quite interesting and current techniques and practice an essential for anyone looking for employment as a developer.

16.6.1 Source Code Management

I was going to write some lengthy notes here, but I found a free on-line book which covers the material really well. Please read "About Version Control" a section of the chapter Chacon and Straub 20.

Here is the full reference to save you scrolling down:

Chacon, S. and Straub, B. (2014a). *Pro Git*, chapter Getting Started, pages 27–42. Apress, Berkely, CA, USA, 2nd edition.

The full-book is available from the git website here. I suggest you bookmark the page or get the pdf for reference purposes, there's a fairly good chance you'll need to look something up.

16.6.2 Git

Again Pro Git book covers this really so read the following sections of Chacon and Straub 20:

- A Short History of Git
- Git Basics
- The Command Line
- First-Time Git Set-up

16.6.3 Git Basics - A crash course in using git

This should be enough to get you going with your own little code repository, you'll need to read this before you do the tutorial. Else your going to spend the tutorial trying to read this as your going.

Chacon, S. and Straub, B. (2014b). *Pro Git*, chapter Git Basics, pages 43–81. Apress, Berkely, CA, USA, 2nd edition.

16.7 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Sign up for a Bitbucket Account (if you don't have one).
2. Sign up for a Bitbucket Cloud Academic Account (if you don't have one).
3. Work though this online tutorial using BitBucket.
4. Finally setup another Bitbucket repository and clone an empty **private** repository of your own to practice with. I suggest you put a current coursework here, e.g. SSUMAE or GCE and try and use the repository in anger. This kind of real world experience with the tool can't be simulated in a tutorial!

Note: There has been no playing with the case study this week, next week it will return with a vengeance when I'll be getting you to work on an implementation of the first iteration as a collective!
Questions to think about (or what kind of exam question could I get on this).

- What do we mean by distributed version control? Provide a detailed description with examples.
- How could a distributed version control system help:
  (a) A lone developer.
  (b) A small development team located in a single office.
  (c) A large development team spread over a number of countries.

16.8 Summary

We've started looking at implementation, this may feel a little premature as we've not covered any detailed design (internal object behaviour,) for the most part object orientated systems are more about object interaction than individual object behaviour however there are exceptions. We're going to do a little more on Git before moving on to look at detailed design, the final stage for us will be to look at some of the wider professional issues in Software Engineering.

This week we've looked at moving from paper/whiteboard designs to object orientated code and at the most basic level how we cooperate to implement this particularly the use of a distributed version control system (git).

16.9 Preparation for Next Week

16.9.1 Essential Reading

Read Larman's detailed explanation of the object implementation:
Larman (2014u). *Applying UML and Patterns*, chapter Mapping Designs to Code, pages 369–385. Prentice Hall PTR.

In addition you need to read the sections of the git book on branching in preparation for next weeks class:

Chacon, S. and Straub, B. (2014c). *Pro Git*, chapter Git Basics, pages 81–125. Apress, Berkely, CA, USA, 2nd edition.

16.9.2 Background Reading

TDD is very popular at present and good understanding will be useful:
Larman (2014w). *Applying UML and Patterns*, chapter Test Driven-Development and Refactoring, pages 385–395. Prentice Hall PTR.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - More Version Control

17.1 Introduction

Last week we started looking at the conversion of the UML sketches into production code and how at the most basic level we can manage a code base shared by a number of developers, this week we'll continue this. The focus however is on getting to grips with distributed version control and git, the importance of which cannot be stressed enough.

17.1.1 Recal: Source Code Management

There were no extensive notes on this last week, we're going to work from the online git book ("Pro Git"), if you missed last week or need to recap please read "About Version Control" a section of the chapter Chacon and Straub 20.

Here is the full reference to save you scrolling down:

Chacon, S. and Straub, B. (2014a). *Pro Git*, chapter Getting Started, pages 27–42. Apress, Berkely, CA, USA, 2nd edition.

The full-book is available from the git website here. I suggest you bookmark the page or get the pdf for reference purposes, there's a fairly good chance you'll need to look something up.

17.1.2 Recap: Git Basics

Last week you should have read the following before attempting the tutorial, if you not done so start here this week (and do last weeks tutorial) **you will find it impossible to start this week without doing so**.

Read this:

Chacon, S. and Straub, B. (2014b). *Pro Git*, chapter Git Basics, pages 43–81. Apress, Berkely, CA, USA, 2nd edition.

Do this: 16.7.

17.1.3 Branching and Merging

Still working from the on-line book please read:

Chacon, S. and Straub, B. (2014b). *Pro Git*, chapter Git Basics, pages 43–81. Apress, Berkely, CA, USA, 2nd edition.

17.2 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. You need to have done last weeks tutorial to have any idea what is going on here!
2. Work though this online tutorial using BitBucket.
3. Set up a shared Bitbucket repository and invite your team members (if your team have abandoned you, joint another this is just a tutorial) start work on implementing the file loader case study.

Note: The focus of this is working with git, version control is big in the industry right now and this will feed into other topics such as continuous development.
Questions to think about (or what kind of exam question could I get on this).

- What do we mean by distributed version control? Provide a detailed description with examples.
- How could a distributed version control system help:
  (a) A lone developer.
  (b) A small development team located in a single office.
  (c) A large development team spread over a number of countries.

17.3 Summary

More work on version control and hopefully some insight into the use of git to manage larger projects. Looking forward we'll be working with more git in the future when (or if) we start looking at continuous integration.

Next week we'll be going back to look at the use of activity and state chart diagrams to for detailed design.

17.4 Preparation for Next Week

17.4.1 Essential Reading

Read next weeks notes and the relevant chapters of Larmans book: Larman (2014t). *Applying UML and Patterns*, chapter UML Activity Diagrams and Modeling, pages 477–485. Prentice Hall PTR.
Larman (2014x). *Applying UML and Patterns*, chapter UML State Machine Diagrams and Modelling, pages 485–493. Prentice Hall PTR.

17.4.2 Background Reading

TDD is very popular at present and good understanding will be useful: Larman (2014w). *Applying UML and Patterns*, chapter Test Driven-Development and Refactoring, pages 385–395. Prentice Hall PTR.

YOU will need to do the preparatory work ahead of next weeks class!

Agile UP: Elaboration - Testing - part 1

18.1 Introduction

Having introduced some group development practices we're now moving to testing. This is a huge topic and we're only going to introduce it in this module. We'll start by introducing testing as a topic and some of the key ideas before moving on to more practical considerations.

18.2 What is Testing?

18.3 Psychological and Economic Considerations

Software testing is a technical task but involves some important psychological and economic considerations [82].

18.3.1 Attitude to Testing

Myers 82 suggests that the correct attitude to testing is essential, humans are goal orientated and establishing a proper goal has an important psychological effect. If the goal is to show that a program has no errors then subconsciously tests are selected to reduce the chance of finding an error [82]. This is to some extent against human nature, which is on the whole constructive (rather than destructive,) and affable rather than confrontational [82]. The implication here is that testing should be a distructive and sadistic process, which aims to destroy. This is partly true but according to Ratzmann and De Yong 89 this is also unhelpful as it causes friction

between developer and tester, they suggests a change of thinking. The same wrong way round view of testing is seen in the description of tests, a test which finds a bug/flaw is a 'failed' test while one which doesn't find a bug has 'passed' [82]. However the test that finds the bug improves our code (the passed test adds nothing,) suggesting perhaps it was a better test [82].

Ratzmann and De Yong 89 agree that a key factor in software testing is the attitude of the tester, but they suggest an alternative mindset for developers/testers. They quote Cockburn who has suggested that software development is more akin to a game, while its possible to discuss tactics and set pieces prior to the game what happens on the day will depend on the opposition and conditions. Therefore the goal of the whole team is to produce high quality (bug free,) software, this means finding bugs is a productive and valuable contribution. This is even more critical where there is no dedicated test team and developers move between roles (as developer/tester,) here changing mindset and looking to create probing tests is an essential for a developer [89]. Interestingly developers faced with re-testing are keen to automate the process [89]

18.3.2 Economic Factors

Even for trivial programs it is impossible to test a program to find all errors, this has implications for the economics of testing, the assumptions the tester will have to make about the program and how the tests are designed [82]. In order to understand why exhaustive program testing is impossible, we consider in general terms the prevalent strategies [82] in software testing black-box and white-box testing.

Black-Box Testing

Black-box testing (also known as data-driven or input/output driven testing) views the system as a black box, ignoring the internal structure and attempting to find circumstances where the program does not produce the expected behaviour [82]. To use this approach to find all errors, you would need to make use of every possible input condition as a test case [82]. Myers 82 provides some illustrative examples, the first is a program (or single function) which attempts to detect different types of triangle (equilateral etc.) using the lengths of the three sides. The only way of checking that all possible combinations is to create test cases for all valid triangles up to the maximum integer size of the the language. This is not exhaustive, we'd need to test all possible invalid inputs also (e.g. 2, A, 2 or -3, 5, 2). An

exhaustive test would require a virtually infinite number of test cases, which is impossible [82]. Second a compiler, would need to be tested with all the possible valid programs and invalid programs. Finally any system which have a memory (i.e. who have a state dependent on their current state,) such as operating systems or database applications will need to be tested with every valid/invalid transaction and every sequence of transactions!

If programs cannot be tested to guarantee that the program is error free then (i.e. exhaustive testing is impossible,) the object of testing is to maximise the yield [82]. That is attempt to find the maximum number of errors found by a finite number of tests cases, this requires we make assumptions about similar cases e.g. a program which detects 2,2,2 as an equilateral triangle will do the same for 3,3,3. [82]. We'll return to this later when we consider test case design **??**.

18.3.3 White-Box Testing

An alternative testing strategy is that of white-box testing (also knows as logic-driven testing,) in which tests are derived from the logic of the application [82]. An important note here is that while this kind of testing can be shown to test the code, it won't ensure the code meets its specification [82]. In my opinion bottom up testing (testing smallest part first, then progressively bigger chunks composed of these parts,) can lead to a kind of tunnel vision. We focus getting the code to work and loose sight of what its supposed to do. Modern approaches including TDD suggest starting with a system test (based on the specification) which fails [44], this will hopefully prevent us becoming overly focused on testing low level code.

Exhaustive testing of the code implies that each statement is tested, this is analogous to exhaustive path testing (testing each execution path functions called, if/else branches taken etc.) [82]. Myers 82 suggests that while such testing would appear to be exhaustive there are two problems. Firstly, the number of logical paths would be astronomically large. His example is a do while loop which iterates 20 times containing two nested if/else statements would result in 10^14 or 100 trillion paths. If an individual could write a test, execute it and evaluate the result in one second it would take 3.2 million years to test each path. In the real world not all branches are independent of each other so the actual total would be less, however, real programs are considerably more complex. Myers 82 also highlights a second problem, despite executing all the paths the program may not meet its specification. Missing paths will not be tested (only those we have in place,) some path

errors may be data sensitive (i.e. they give an incorrect result) we may take both paths in the if/else but this doesn't test that the comparison is being done correctly [82].

18.4 Myers Principles of Testing

Myers 82 outlines the following key principles in his work:

1. **A necessary part of a test case is a definition of the expected output or result.** Plausible but incorrect results can be accepted unless detailed input/output data or start/end state information are provided. Essentially there is chance of us "seeing what we want to see".

2. **A programmer should avoid testing their own program**. There are a number of psychological issues, we don't want to find (and fix bugs in our own code,) we don't want to break what we've spent time creating and finally if we misunderstood the problem in the first place we can only test against our own metal model of the system.

3. **A programming organisation should not test its own programs**. As for the above, however more generally finding errors increases project time/cost and may reduce profit.

4. **Thoroughly inspect the result of each test**. Symptoms of errors are often overlooked in low level testing.

5. **Test cases must be written for conditions that are invalid and unexpected, as well as those that are valid and expected**. We have a tendency to prove that the program works as expected i.e. with valid data in expected conditions. Rather than more generally trying to prove that it performs as expected with any combination of data and conditions.

6. **Examining a program to see if its does not do what it is suppose to do is only have the battle; the other half is see whether the program does what it is not suppose to do**. This extends the above, a program may do what is expected and nothing else, i.e. it shouldn't do anything we don't expect it to do. For example, a program which loads a file may load the file correctly but if the file is over-written or corrupted by the process it should not pass testing.

7. **Avoid throwaway test cases unless the program truly a throw-away program**. Running a program and working though some tests invented 'on the fly' represents an investment of time in useful test cases. If the software has to be fixed, these tests (or similar ones,) should be re-run to ensure that the fixed has not caused and error (re-gression testing **??**). Storing these tests or incorporating them into an automated testing suite removes this problem.

8. **Do on plan a testing effort under the tacit assumption that no errors will be found.** A common mistake often founded on an incorrect definition of testing, testing does not simply show the program runs correctly for a pre-determined set of values. Testing aims to find errors in the program.

9. **The probability of the existence of more errors in a section of program is proportional to the number of errors already found in that section**. In terms of classes this means if class A has been found to contain more errors than class B. There is a greater chance of finding more errors in class A. Essentially errors tend to cluster, the reason for this is unknown but it helps focus our testing effort.

10. **Testing is a extremely creative and intellectually challenging task**. It takes more creativity to think of a dozen ways something can go wrong than one way it can go right. Writing effective tests is difficult it challenges us to be creative in destroying something, which often doesn't come naturally.

18.5 Designing Test Cases

In the past I've always started with the phases of testing 18.6 then considered the different types of test. However having read Myers 83 who introduces test planning first, this seems a more sensible order. At this stage I'd like to make clear that testing includes tests of individual units (a unit test **??**), tests of units being combined (an integration or incremental test **??**) and tests of the system as a whole (function and system tests) [**?**]. The following section considers general black box and white box testing strategies which are employed at the lower (smaller parts of the system,) levels.

Given the constraints on time and cost the key issue of testing becomes finding a subset of all possible test cases which has the highest possibility of

finding the most errors [84]. The lest effective methodology is selecting random values to enter into the system, however more effective methodologies (for both black and white box testing,) have been developed [84].

We're going to cover a selection of methodologies, normally rigours testing would be undertaken with black box methods first then supplemented with white-box methods to ensure the logic of the program is fully exercised [84]. These are covered in more detail in Myers 84.

18.5.1 Black Box Testing Methodologies

There are several key black box testing methodologies, including Equivalence partitioning, boundary-value analysis, cause-effect graphing and error guessing [84]. These will be considered in the sections below.

Equivalence Partitioning

As previously discussed we're looking to find a subset of all possible tests which will find the most errors. This requires we appreciated that firstly a good test set should reduce the number of further tests to be developed and secondly it should cover a large set of other possible values (i.e. it suggest the code works for other values) [84]. The second of these properties implies that the input domain can be partitioned into a finite number of equivalence classes (such that it is possible to assume - but not be sure,) that the result would be the same for any other member of the set [84]. Having identified our set of equivalence classes we aim to identify the minimal set of test cases to cover those classes [84].

Guidance: Finding Equivalence Classes .
Myers 84 suggests that we take each input to the system and partition it into at least two classes (valid/invalid). He suggests that in most cases at least one valid and two invalid cases will be required, for example a input with a valid range 0-99 will require one above, one below and one in the valid region. His final advice is that test cases should never be combined, so for example if the input to a function under test was BookType and price, we'd avoid entering "XYZ" and -1 as the program would respond correctly rejecting only the first parameter as an error.

Examples .

Myers 84(page 56) provides a FORTRAN example.

The following simplified example is based on one found online. Suppose a password field allows characters, numbers and symbols as input. The maximum length of a password is 11 characters and the minimum is 6.

Equivelence Classes and Test Cases

Class No.	Description	Expected Output	Test Case
1.	length 0-5	invalid	"abc"
2.	length 6-11	valid	"abcdefgh"
3.	length 11-99	invalid	"abcdefghijklmnop"

Table 18.1: Simple Equivalence Classes.

Boundary-value Analysis

Boundary conditions (as their name suggests,) are those on around the boundary of equivalence classes [84]. Myers 84 suggests that test cases which explore the boundaries (of both input and output equivalence classes,) often have a better chance of finding errors that those that do not. He notes two key differences between this and equivalence partitioning, firstly rather than selecting cases to be representative we look for cases at the edge of a class (i.e. above, below and on the boundary). Secondly rather than focusing on the input space we also consider the output (or result) space.

Guidelines for Writing Myers 84 acknowledges that is more difficult to undertake boundary value analysis as it requires more creativity, knowledge of the problem and a certain state of mind. The following general guidance (see 18.3 is provided [84]:

	Guidance	Example
1.	Where are range of inputs is valid, write test cases for the ends of the range and invalid input cases for just beyond the ends.	For a range of -1.0 to 1.0 test for -1.0, 1.0, -1.001 and 1.001.
2.	Where a number of values are allowed write test cases for the minimum and maximum number of values along with one over and one under these values.	If a polygon can have a minimum of 3 sides and maximum of 20 sides, we'd test for 2 and 3 at the lower end and 20 and 21 at the top end.
3.	Use 1. for each output condition.	For example if the range of valid outputs is 0 to 10, write test cases to cause each of these outputs and which attempt to case outputs over 10 and less than 0.
4.	Use 2. for each output condition.	For example if a routine returns the most relevant articles based on a search string, but never more than four. Provide test cases which result in zero, one and four abstracts. In addition try to get five, for example generating a test case where more than one article generates a top similarity score.
5.	If the input or output of a program is an ordered set (a sequential file, linear list, table etc.) focus on the first and last elements of the table.	This is self explanatory.

Table 18.2: Boundary-value Analysis - Guidance and Examples.

Myers 84(page 61.) has another extended example, but I think we've done enough.

Cause-effect Graphing

The weakness of both of the above approaches is they do not test combinations of input circumstances [84], for example boundary analysis for images sizes may test the maximum width and hight, but if the combination of these leads the program to run out of memory this would not be triggered.

Cause and effect graphing looks to map a scenario as a logical network (using a simple notation) [84].

The process (as outlined by [84]:

- Divide the specification up into workable pieces - e.g. scan item
- Identify the cause and effects in the specification:
  - **Cause** - An input condition (variable or state,) or an equivalence class of input conditions.
  - **Effect** - An output condition (result, state change, transformation etc.)
- The semantic content of the specification is analysed to turn the system into a Boolean graph, annotated with constraints describing the combination of causes and effects allowed by syntactic or environmental constraints.
- Tracing state conditions in the graph leads to a limited-entry (sparse) decision table with each column representing a test case.

The following short example is adapted from that given in Myers 84.

A vending machine takes two inputs from a keypad, the first is a letter representing the column of the product to be vended ("A" or "B"), the second is a number representing the row (these are clearly labelled on above the products in the vending machine). If the first character is incorrect message E1 is output if the digit is incorrect message E2 is output.

The causes for this example are:

1. First (1) input is an "A"
2. First (1) input is an "B"
3. Second (2) input is an digit

The effects are:

1. Vend product (71)
2. Message E1 issued (70)
3. Message E2 issued (72)

The graph for this is shown below (see 18.1), note that intermediate node 11 has been created.

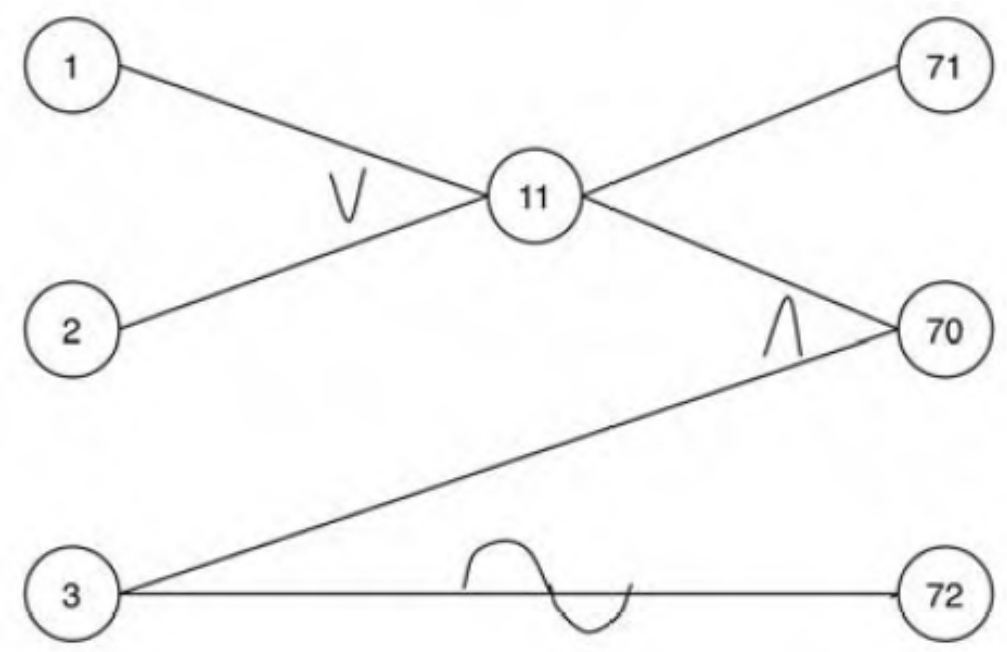


Figure 18.1: A simple cause-effect graph adapted from Myers 84

There is also a slight omission, we've not considered that the input cannot by both "A" and "B" at the same time, we can enforce this using a constraint. Constraints allow us to convey impossible combination of logical events caused by syntax or environment [84] see **??**.

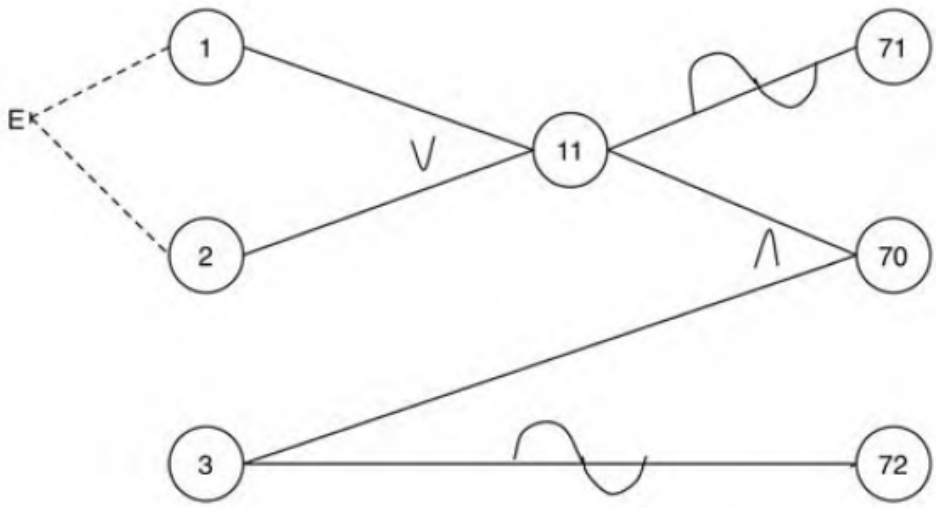


Figure 18.2: Updated to include constraints (again adapted from Myers 84.)

The decision table is as follows (I've created this based on the above example) **??**.

Test Case	1	2	3	4	5	6
1	1	0	1	0	0	0
2	0	1	0	1	0	0
3	1	1	0	0	1	0
70	1	1	0	0	0	0
71	0	0	0	0	1	1
72	0	0	1	1	0	0

Table 18.3: Decision Table - Example.

Myers 84(page 71.) has another extended example (and covers more of the syntax), but I think we've done enough.

Error Guessing

Myers 84 suggest that some individuals are naturally adept at program testing, employing both intuition and experience they determine the most probable types of errors. He suggests that approaches include looking for common 'troublesome' inputs like blanks in fields, zero, negatives for counts etc., for lists empty lists, lists of only one item, lists with duplicates etc. This amounts

to looking for special cases which may have been overlooked by the developer (or assumptions they may have made) [84].

18.5.2 White Box Testing Methodologies

White box testing is concerned with the the proportion of the code which is exercised by the tests, though we accept that the ultimate goal of executing every path is not feasible for non trivial programs [84].

Logic-Coverage Testing

Myers 84 Provides a detailed analysis with example of the various metrics to define successful logical-coverage testing. I've tried to summarise these below and reworked some of his examples in C.

Statement coverage - Ensure every statement in the program is executed at least once. Executing every statement comes no where near executing every path. Given a program with multiple if statements we'd only cover the condition where both are true (see 18.1).

Listing 18.1: Example

```
bool noddy(bool a, bool b)
{
        bool result = false;

    if(a == true && b == true)
    {
                result = true;
        }
}
```

As a further example in 18.2 we'd have complete statement coverage with:

1. A = 2, B = 0, X = 3

Decision or Branch coverage - Ensure each decision has a true/false at least once, this usually satisfies statement coverage (as statements are in branches). There is a suggestion if every branch is covered then every statement is covered, however there are exceptions: Programs with no branches, programs or subroutines with multiple entry points (including where global/object data changes the statements executed) and finally

loops, traversing every branch direction (loop or now) will not necessarily cause all loop iterations to be executed.

Decision testing can be extended so that all branches must be executed and all statements. Can be extended to include all points of entry (e.g. each iteration of a loop) to ensure all conditions (including loop termination) are exercised. This still may not cover all paths and must be extended if the language allows select(case) or other constructs. Consider listing 18.3 decision coverage requires A-true, B-true and A-false, B-false, the logical operator connecting them could be wrong and we'd never known.

In a further example we consider a graph of the example program 18.2 shown in 18.3 If we test paths, *acd* and *abe* using the following test data:

1. A = 3, B = 0, X = 3 (acd)
2. A = 2, B = 1, X = 1 (abe)

The condition X>1 could be incorrect (perhaps it should be X<1) and our test wouldn't find it!

Listing 18.2: Multiple Condition Example

```
void foo(int a, int b, int x)
{
        if(a > 1 && b == 0)
        {
                x = x / a;
        }

        if(a == 2 || x > 1)
        {
                x = x + 1;
        }
}
```

Condition coverage - Ensure that each condition in a decision takes on all possible outcomes at least once and that each point of entry to the program ore sub routine, and ON-units (loops) be invoked at least once. This seems to give the same as condition coverage but consider listing 18.3. Condition coverage requires only A-false, B-true and A-true, B-false, but

the if never fires.

In a further example we consider a graph of the example program 18.2 shown in 18.3 If we test paths, *ace* and *adb* using the following test data:

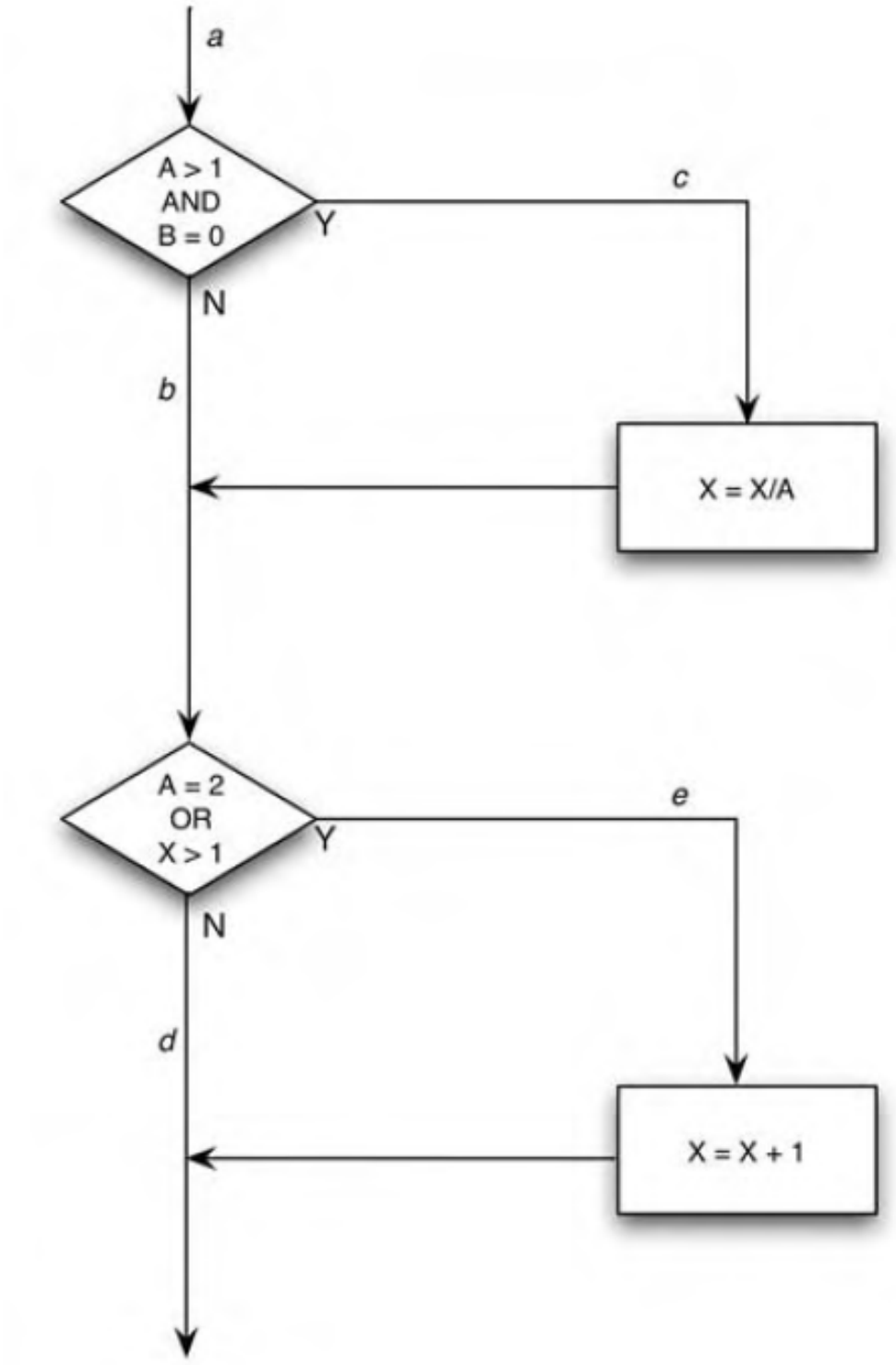


Figure 18.3: A graph for a small program, from Myers 84

1. A = 2, B = 0, X = 4 (ace)
2. A = 1, B = 1, X = 1 (abd)

By chance this gives us the same number of test cases and may ensure that all conditions in a decision are executed.

Decision/condition coverage Essentially combine the above, write enough test cases that each condition in a decision takes on all outcomes at least once, each decision takes on all possible outcomes at least once and each point of entry is invoked at least once (loops, overloads etc.). The weakness of this approach is more difficult to see, this has to do with how the code is generated. If two conditions in a decision are connected by an 'and' and the first is false the second is never executed. Likewise if the first condition of a decision connected by an 'or' is true the second is never executed.

Listing 18.3: Example

```
bool noddy(bool a, bool b)
{
    bool result;

    if(a == true && b == true)
    {
        result = true;
    }
    else
    {
        result = false;
    }
}
```

Multiple-condition coverage - Requires that we write enough tests that all possible combinations of condition outcomes in each decision and all points of entry are executed at least once. A set of test which achieves this should cover all of the above.

Returning to the further example we consider a graph of the example program 18.2 shown in 18.3 There are in face eight combination which must be tested:

1. A > 1, B = 0

2. A > 1, B != 0
3. A <= 1, B = 0
4. A <=1 1, B != 0
5. A = 2, X > 0
6. A = 2 1, X <= 0
7. A != 2, X > 0
8. A != 2, X > 0

Cases 5 through 8 express values at the point of the second if statement, requiring us to ensure that it will reach this value after modification. Despite the large number of combinations these can be covered by only four test cases:

1. A = 2, B = 0, X = 4 (Covers 1, 5)
2. A = 2, B = 1, X = 1 (Covers 2, 6)
3. A = 1, B = 0, X = 2 (Covers 3, 7)
4. A = 1, B = 1, X = 1 (Covers 4, 8)

18.5.3 Testing Strategy

[84] suggests that these should be combined to maximise the chance of finding errors. Further more he suggest:

1. If input conditions are a factor, start with cause-effect graphing.
2. Use boundary-value analysis.
3. Identify valid and invalid equivalence classes and supplement the above as required.
4. Use error-guessing to add additional test cases.
5. Examine the programs logic and the extent to which the criterion of the white box tests covered earlier have been met (software may perform this test,) then add test cases until a satisfactory level is reached.

18.6 Phases of Testing

The above testing strategy is great for programs which are relatively small, such as embedded systems however to tackle anything larger we need a more structured approach [85]. The text I'm referring to here shows its age to some extent in that at the time it was written smaller systems were more common. Myers 85 and others cover this in more detail.

18.6.1 Unit Testing

Unit testing is the process of testing the building blocks of the program (that is sub-programs, subroutines, procedures, functions or classes) rather than initially test the program as a whole [85, 90]. There are three key motivations for this [85]:

- To help manage the combined elements of testing by initially focusing on smaller units.
- To ease the task of debugging (locating errors,) by reducing the size of the space in which the error may exist.
- To allow for parallelism in program testing, as multiple modules can be individually tested.

During unit testing the majority of the test which takes places is white box orientated, primarily as testing larger entities (such as programs,) becomes far less feasible [85]. Subsequent testing (of larger entities,) moves towards finding different types of errors associated with not meeting requirements rather than logical errors [85]. Normally we begin with the white box methods described above 18.5.2 then supplement these by applying black box tests to the modules specification [85].

18.6.2 Integration Testing

When testing modules there are two key considerations, how the module works on its own and how it combines with others [85]. The latter is referred to as Incremental testing by Mayers but most authors seem use the term Integration testing [90]. The essential concept here is that we're bringing together tested modules and make sure they work together, the question is how? Myers 85 suggests that the approaches can be generally classified as non-incremental (or "big-bang") and incremental (which can be further

divided into "top-down" and "bottom-up" [85].

In order to consider the implementation strategies we'll introduce the ubiquitous block diagram (see 18.4) of "modules", note that these may be subroutines or procedures in an assembler program, functions in a higher level language, methods of a class or classes themselves. I've even also modules/units described as the code developed by a single developer.

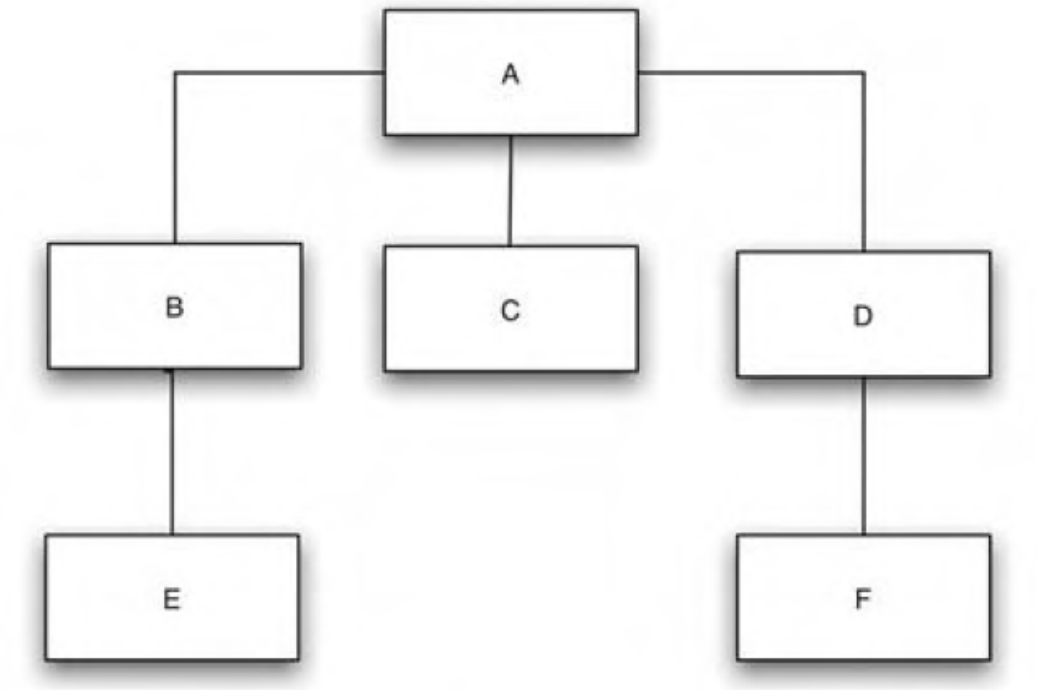


Figure 18.4: A program composed of multiple modules A-F, from Myers 85

Non-Incremental Testing

The non-incremental approach (or "big-ban") approach is a very traditional approach to testing, each module is tested as a stand-alone (either in series or parallel) [85]. These are then combined to form the program [85].

The testing of each module requires a special driver module, which runs the test cases against the module (i.e. making calls and passing arguments,) in a addition one or more stub modules to fill in for other modules [**?**]. For example to test unit B in the example above 18.4 we would need a driver and stub for E. The stub must provide the same services as E, though it

may 'fake' and processing and simply return set values.

Incremental Testing

An alternative approach is incremental testing, rather than test each module in isolation the next module to be tested is first combined with a set of modules which have already been tested [85]. Working in this way has some advantages Myers 85 lists these as follows:

- Less work in terms of drivers and stubs, this will become apparent when we consider the two approaches to incremental testing (top-down and bottom-up).
- Errors caused by mismatching interfaces or incorrect assumptions among modules are detected early (rather than at the end). This aids debugging as we can work on the interfaces between the new module and those we've already tested in isolation.
- More testing is undertaken as the complete (and tested,) parts of the system are executed repeatedly as more parts are added.

There are some disadvantages [85]:

- The time taken to run the incremental test will probably be greater as much more code will be executed as the system becomes more complete.
- The non-incremental testing offers more opportunity for parallel activity, i.e more modules can be simultaneously tested (if you have the people).

Top-Down Testing .

Top down testing (or top down development,) is a method of coding, testing and combining modules [85]. In the above example 18.4 we might order these, A, B, C, D, E, F and therefore only require stubs for B, C and D while testing A, E while testing B and F while testing D.

? ? suggests that this is typically a difficult process. He notes that it is often difficult to test the interface to a program without restarting from the start of series of operations each time (potentially time consuming). Also that there is a temptation to start testing before design of the lower level dependencies has been completed, clearly an issue given what we know about

the iterative nature of design. Finally he notes that in practice due to the difficulties associated with writing good stub classes (i.e. can be used for more than one test case etc.) developers often put off complete testing of high level components until later. The key advantage (and one exploited by TDD,) is testing begins with the layers closest to the business logic (and these layers are tested the most) [85].

Bottom-Up Testing .

The bottom-up strategy begins with the terminal modules, i.e. those with no dependencies [85]. In the above example 18.4 we might order these, E, C, F, B, D, A and therefore only require drivers for testing E, C and F, then B, D and finally A.

Myers 85 notes that the key drawback of this approach is that there is no skeletal programming until the very end (i.e. the key business logic is not exercised until last). The difficulties in fully testing modules due to missing dependencies disappears and fewer stubs are created [85]. This is not completely true, the real world we end up creating stubs for our colleagues modules where they must be integrated with ours.
Myers 85 in his final comparison (see 18.5) suggests that bottom-up has the edge, however his reason is the availability of tools which eliminate the need to write drivers (e.g. test frameworks). As mock frameworks now exist this may no longer hold.

Comparison of Top-down and Bottom-up Testing

Top-down Testing	
Advantages	*Disadvantages*
1. This is advantageous if major flaws occur toward the top of the program. 2. Once the I/O functions are added, representation of test cases is easier. 3. Early skeletal program allows demonstrations and boosts morale.	1. Stub modules must be produced. 2. Stub modules are often more complicated than they first appear to be. 3. Before the I/O functions are added, the representation of test cases in stubs can be difficult. 4. Test conditions may be impossible, or very difficult, to create. 5. Observation of test output is more difficult. 6. It allows one to think that design and testing can be overlapped. 7. It induces one to defer completion of the testing of certain modules.
Bottom-up Testing	
Advantages	*Disadvantages*
1. This is advantageous if major flaws occur toward the bottom of the program. 2. Test conditions are easier to create. 3. Observation of test results is easier.	1. Driver modules must be produced. 2. The program as an entity does not exist until the last module is added.

Figure 18.5: A comparison of top-down and bottem-up intergraiton, from Myers 85

Guidelines for Performing the Tests

Myers 85 suggests the following hints and guidelines:

1. Remember when a tests fails it may be the test case which is incorrect, this must be checked along with the code it tests.

2. The use of automated test tools can reduce some of the coding effort required to write drivers.

3. Flow analysis tools can provide useful metrics, find unreachable code and other simple tests.

4. Look for side effects, e.g. check that the command-query principle has not been violated (**??**).

5. Psychological problems associated with programmers testing their own modules can be overcome by swapping modules for testing.

6. If a subset of modules contain an abnormally high number of errors, its likely they will contain more and need further testing.

18.6.3 Higher-Order Testing

Having completed unit and integration testing we have at least some confidence that the program operates as we expected, however, consider the following quote from Myers 86 "A software error occurs when the program does not do what its end user reasonably expects it to do." This broader definition suggests that even if our program executes the tests perfectly there may still be errors caused by errors in the transmission and translation of information [86]. That is the system may do exactly what is was designed to do, but this may not be what was originally required.

Clearly the first step to trying to solve this problem is to ensure the process of soliciting the users requirements and turning them into software is as robust as it can be [86] (the topic of much of the early part of this module). However we also need to consider ensuring that this is the case, by undertaking a broader set of tests which consider how the program meets the needs of its users [86].

? ? suggest that while this is a more traditional view of testing it can still be applied in iterative and incremental projects (as we'll see later **??**. This model is normally shown as a 'V' showing the relation between phases

of translation from End-user to code and the testing of each translation (see 18.6).

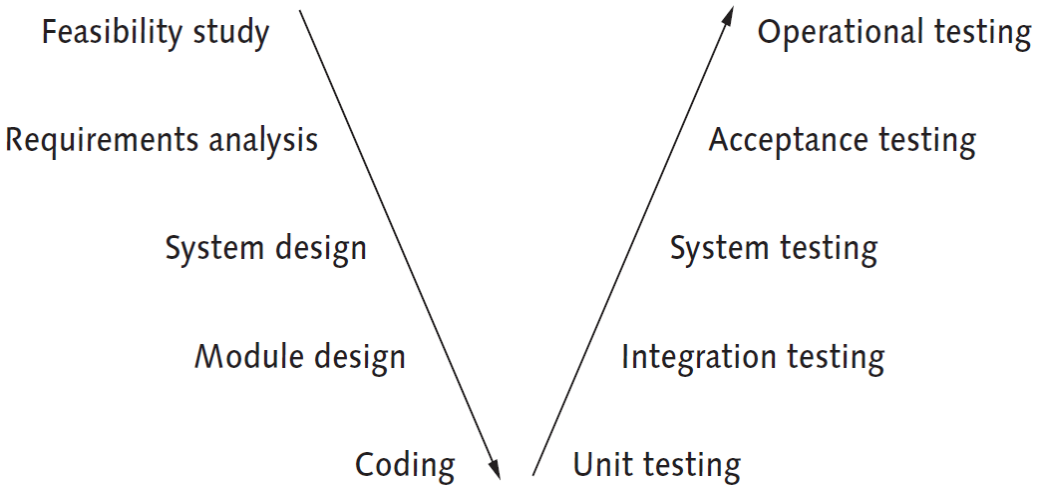


Figure 18.6: Classing test planning model, from ? ?

Function Testing

Function and system testing are according to Myers 86 two distinct stages though often considered as one (I think ? ? does this for example). This stage of testing begins with the specification for the system and test that it meets its functional requirements, for all but the smallest systems this is normally black box testing (see here 18.5.1 for methodologies) [86].

System Testing

System testing is (as suggested above,) often misunderstood, it is not simply testing a program to its functions (the services it provides) work as expected (this is Function Testing) [86]. Instead we return to the original objectives of the problem, the goals of the user and the constraints imposed by external factors and derive tests cases based on these objectives [86]. The inherent difficulty is that we ideally want to test the objectives against the system without knowing in advance the behaviours provided by that system [86], i.e. we're looking to approach the system as a user with a goal and criterion for the satisfaction of that goal in mind. Note here that this may combine functional requirements with non-functional, e.g. the user wants to get x done but it should take no longer than y. Myers 86 suggests that this should be possible using the highest level description of the users needs and the

documentation for using the system.

These tests should be carried out by professional testers possibly with the assistance of end users, ideally these should have no association with the project or the company who have produced the software [86].

Due to the highly specific nature of this process there are no well defined test-case design methodologies and a substantial amount of creativity is required [86]. Myers 86 however suggests that the following categories should at least provide guidance as to the kind of tests required (though these may not be applicable to every program).

Facility Testing .

The most obvious kind of system test is to determine whether all the facilities (rather than functions) are present [86]. This is perhaps akin to making sure that we can achieve all the user goals rather than making sure all the use-case can be performed (which is more functional testing). Myers 86 suggests that this is often possible without a computer, we simply compare the objectives of the system with the documentation to ensure parity.

Volume Testing .

As the name suggests this type of testing involves subjecting the program to large (or difficult to process,) volumes of data, the purpose of which being to show that the system cannot handle the volume of data specified in its objectives [86]. Many of the examples quoted seem to involve databases and the number of records but these are not the only tests, we may look at loading very large files, huge models in a game, large numbers of tasks in an operating system etc.

Stress Testing .

Stress testing subjects the program to heavy loads or stresses, i.e. a peak volume of data or activity encountered over a short period of time [86]. We're not interested in whether the system can cope with a large number of records, but if it can cope with large numbers of records all being added at the same time. A good example from Myers 86 is that of an air traffic control system, if the maximum number of places in its sector is 200 a good stress tests would be to simulate 200 planes. This can be taken further by examining the edition

of the 201st plane and simultaneously adding large numbers of planes to the sector. This is more applicable to programs which operate under varying loads, are interactive, real-time or process control rather than batch control systems [86]. Another tangible example is that of concurrent users for web-based applications and databases [86], multi-player games would also require such testing.

Usability Testing .

Another important category of system test case is that which attempts to identify usability (HCI or human-factor) problems [86]. This is an area which has grown significantly in recent years [86] and continues to evolve rapidly as devices become more common and their interfaces more varied. Usability testing is the subject of a module on the third year of some programs, Myers 86 suggests the following:

- Has each interface been tailored to the training, experience and environment of the end user?
- Are the program outputs meaningful, considered and clear?
- Are the error messages easy to read/understand? E.g. "File not found" rather than "IEK002A OPEN ERROR ON FILE 'SYSIN' ABEND CODE=102" the latter being common in 1970s software systems. The modern day "An unknown error has occurred" is about as useful.
- Does the total set of all user interfaces exhibit conceptual integrity (are the ideas expressed in the same way throughout). Is there an underlying consistency and uniformity of syntax (same names for things and operations) , conventions (buttons in the same order etc.,) semantics, format, style and abbreviations.
- Where accuracy is vital, is the sufficient redundancy in the system? This is not just a question of asking "Are you sure?", e.g. entering passwords and emails twice on web forms, re-entering your secure key on when transferring money in on-line banking.
- Are the options present excessive or are some more likely to be used than others, one modern trend is towards basic and advanced menus and even towards frequency based ordering.
- Does the system provided immediate feedback on all inputs e.g. buttons animating when clicked (this seems obvious,), selections which have

been made should be shown, when a processing or communication are likely to be slow some message, or graphic should be shown to inform the user of current progress.

- Is the program easy to use? Does it present enough information to the user for them to use it with out extensive training, is navigating between screens intuitive (e.g. is it clear how to return to the main menu). Is the user input case sensitive (if so is the user informed) etc.

Security Testing .

Security in applications has undergone a transformation as the number of connected devices has increased [121]. While initial concerns were for information security and the changes ICT brought to the control of protected information (previously guarded by physical access protocols) [121]. Recently the protection of not only information, but individuals, organisations and their assets (devices, machinery, infrastructure etc.,) have required securing leading to the defining of cybersecurity [**?**].

The increasing concerns of society have resulted in many programs to have security objectives hence the requirements of test cases which attempt to circumvent the security of a system (or its dependent systems e.g. operating system, database etc.) [86]. This may include know exploits for third party systems, operating systems or other similar systems or the use of general exploits as published in relevant sources [86].

Performance Testing .

Most programs will have some performance requirement, this may be in terms of speed (response time,) or other metric e.g. throughput rate under a given workload/configuration etc. [86]. Again we note that the purpose of testing is to show the system does not meet its performance objectives [86].

Storage Testing .

Some programs will have storage objectives, this may include main memory, secondary memory or temporary files - test cases must be devised to show that these objectives are not met [86].

Configuration Testing .

Operating systems, DBMS, middle-ware, server-side applications etc., all need to be correctly configured in order to work [86]. This may include the amount of available memory on a system, versions of libraries installed etc., these kind of tests must consider minimal and maximal values and the versions of required libraries. In my experience forward compatibility is not always guaranteed and even software from top manufacturers can show weird configuration bugs (e.g. IBMs Rational Modeller at one stage required the loop back address was at the top of the hosts file).

Compatibility/Configuration/Conversion Testing .

This type of testing is concerned with ensuring compatibility between new and old systems or between the system an existing systems [86]. Modern programs often replace previous versions either when these are retired or replaced as the result of merger or acquisition, in such cases we need to ensure data (e.g. database records,) from one system can be moved to the other without issue. We also need to consider upgrades of various depended components and how these can be tested [86], consider updating the DBMS, windowing API (.Net) or graphics API (DirectX).

Installability Testing .

The installation of software systems is an important step, in some systems e.g. embedded systems or bespoke systems the development team can be reasonably confident of the hardware/system state at the time of install. However in other systems where multiple (potentially conflicting) programs are running parallel ensuring or where a user is expected to undertake the install the procedure (or its automation) must be carefully tested [86].

Reliability Testing .

Improving reliability is a goal of testing for all types of program, however some software for example corporate networks, Internet Services Providers (ISPs) and telecommunications software is expected to work flawlessly for years with minimal downtime [86]. This kind of testing is a specialist subject which some of you will encounter again in the third year.

Recovery Testing .

Programs such as operating systems, DBMS etc. will have recovery objectives which provide guidance as to how long it should take to recover from a software/data errors and hardware failure [86]. This kind of testing may include induced communication or data errors, injected programming errors, disconnecting or causing parity errors in hardware [86]. Again this kind of testing is a specialist subject which some of you will encounter again in the third year.

Serviceability Testing .

Some programs may have objectives related to how easily they can be serviced or repaired [86]. Test cases here will be based on the facilities provided to diagnose issues, time to debug, maintenance procedures and internal documentation [86].

Documentation Testing .

System testing is also concerned with the accuracy of documentation provided to users, this is usually tested by using the documentation to guide the creation of other system tests (e.g. stress tests) [86]. All documentation should be proof read and checked for accuracy and clarity, any examples provided should be re-formulated as test cases as part of this process [86].

Procedure Testing .

Finally many programs are part of larger systems which may or may not be computerised and prescribed procedures which need to be carried out by the operator must also be tested [86]. For example there may be recovery procedures which involve rebooting servers and reinitialising services, these must be documented. Backup systems may involve the creation and storage of physical copies (e.g. tapes in a fire proof safe).

Acceptance Testing

The acceptance test is an unusual one as its normally undertaken by the customer or end user to ensure that the program meets their needs [86]. In the case of software developed to contract the software will be compared to the original contract [86]. As with all testing tests are devised which attempt to show we've not met the needs of the customer or the contract [86].

Installation Testing

The last test process is not related to the development process but considers the installation of the software, there are usually a number of factor which need to be considered for example [86]:

- Users must select options, testing must ensure that all sets of options result in a valid configuration.
- Files and libraries must be installed and located. Many programs rely on static or shared libraries being distributed with the software or linked at runtime.
- Valid hardware configurations must be present.
- Programs may need network connectivity.

Test Completion Criteria

Myers 86 provides a lengthy discussion of the possible criterion for ending testing. He suggests that basic metrics such as "Until we run out of time" and "Until all the tests fail to uncover more errors" are unhelpful as neither provides any incentive to find errors. His categorisation of better stopping criteria are outlined below.

Use of Specific Test-case-design Methodologies .

This is not the best, we use the use of given methods as a basis for completion [86]. For example (from Myers 86):
Module testing ends when - the test cases derived to satisfy the multi-condition-coverage criterion and boundary-value analysis of the module interface specification are no longer successful in finding errors.

Function testing is complete when - the test cases from cause-effect graphing, boundary-value analysis and error guessing are no longer successful in finding errors.

Myers 86 notes some issues:

- It may not be possible to use these methodologies in the current phase.
- Its a subjective measure, while the tester may have applied the methodology they may not have done it correctly (or thoroughly).

Detection of a Given Number of Errors .

As the goal of testing is to find errors a better criteria is to find a pre-defined number of errors based on estimates of the total number of errors in the program, the percentage of the errors which can feasibly be found and the fraction of these errors originating in the various phases of analysis and design will be found in this phase of testing [86]. There are a number of ways of gaining such estimates including experience with similar programs, inserting errors and seeing what percentage are found in a given period, using industry averages etc. [86].

Rate of Error Detection .

The final method relies on judgement and intuition and requires the tracking of the number of errors found per time unit during each testing phase [86]. These can be plotted and the shape of the resulting graph used to determine whether to continue the test or move to the next phase [86]. Ideally if the number of errors is decreasing or even better demonstrates an asymptotic decay we can safely move to the next phase, if its increasing we should probably continue with this phase [86].

Myers 86 concludes that combination of these is probably required. He suggests that in unit testing and integration testing we ensure that we've written representative test cases using by deploying appropriate methodologies. In system testing we test until we've found a predefined number of errors (based on estimates) or the scheduled time for this phase has elapsed, provided analysis of the error/time graph indicates that the test has become unproductive.

18.7 Tutorial

Hopefully if you all did the background reading we'll have lots of time to work though this:

1. Attempt the gtest for the simple example provided.

Note: The focus of this is working with git, version control is big in the industry right now and this will feed into other topics such as continuous development.
Questions to think about (or what kind of exam question could I get on this).

- What are the types of testing?

- What are the phases of testing and what type of testing is employed at each?
- Provide a detailed description of incremental and non-incremental testing with reference to the process of testing and combining modules in each case?

18.8 Summary

Testing is a key process in software development which is unfortunately overlooked. There are a number of psychological issues including the need for a destructive mind set while programming and the goal of finding errors. A successful test is one which finds errors rather than one which does not. Testing proceeds in phases from the testing of units of the program, to combinations of units, to the system as a whole and the kind of test undertaken moves from white-box to black box. The final stages of testing begin to explore the emergent properties (non-functional requirements,) in addition to the functional ones.

18.9 Preparation for Next Week

18.9.1 Essential Reading

Read next weeks notes and the relevant chapters of Myers book: Myers, G. J. (2004a). *The ART of Software Testing*, chapter The Psychology and Economics of Program Testing, pages 5–20. John Wiley & Sons, Ltd.
Myers, G. J. (2004d). *The ART of Software Testing*, chapter Module (Unit) Testing, pages 91–123. John Wiley & Sons, Ltd.
Myers, G. J. (2004e). *The ART of Software Testing*, chapter Higher-Order Testing, pages 129–156. John Wiley & Sons, Ltd.

18.9.2 Background Reading

[90]
There are more contemporary tests on software testing, we'll cover some of this next week: .
Ratzmann, M. and De Yong, C. (2003b). *Software Testing and Internationalization*, chapter Rapid Application Testing, pages 71–147. Lemoine

International, Inc.

YOU will need to do the preparatory work ahead of next weeks class!

Google Test: Introduction and worksheet

19.1 Introduction

Google Test is a C++ test framework written to be platform neutral (i.e. it compiles for Linux, Windows and Mac.) [**?**]. Its features include [**?**]:

- Test are independent and repeatable, this is ensured by creating a new object to run the test on each time. This reduces the chance of one test causing another to fail. Tests can also be run independently this avoids running a large test suite to debug a single test.
- Tests are organised to reflect the code structure, tests can share data and subroutines.
- The test are both portable and reusable as the code is platform nutral it can be compiled for different OSes allowing test harnesses to be run on different platforms when porting.
- Failing tests provide as much information as possible, however they do not stop the testing. The current tests is stopped and the failure reporting, allowing multiple failed tests to be examined in a single run-edit-compile cycle.
- Google Test does most of the work, it collects the tests together and keeps track of them without requiring the developer to assign specific names etc.
- Optimisations such as sharing resources across multiple tests etc. are possible.

Google Test is based on an of testing frameworks known as xUnit [**?**], we'll mention this briefly as its history is interesting.

19.1.1 xUnit

XUnit is a name given to a family of testing frameworks [?] in which x is replaced with something language specific e.g. pyUnit, JUnit, CppUnit etc. According to [?] it was developed from ideas in the Smalltalk communit. Agile pioneer Kent Beck (a big fan of automated testing,) would encourage (and guide) the teams he mentored to create a unit test framework for their project. These ideas were ported to Java in 1997 when Beck and Erich Gammer (another Agile pioneer,) travelled together to the OOPSLA conference in Atlanta (on the plane). This was refined by Fowler and others and was quickly accepted by a growing Extreme Programming and Test Driven Development community. It was followed by a number of ports, varying from line by line copies of the original to complete re-writes.

19.2 Using Google Test

This section covers the use of some of the basic features of Google test in particular the creation of test fixtures for a given class. The original code was written with reference to this guide. This guide is written based on the Google Test Primer this is an excellent source on the workings of the API but not really formatted as a tutorial.

19.2.1 Basic Concepts

Google Test works on the basis that a test program contains multiple test cases [?]. Each Test Case contains one or more tests grouped to reflect the structure of the tested code, where such tests need to share objects or subroutines (e.g. connections to test oracles, stubs, mock objects etc.,) they can be built into a test fixture class [?]. Each test case contains one or more test which verify the codes behaviour, if the test crashes or an assertion fails then the test is said to have failed [?].

The tests themselves are composed of assertions, statements which check if a condition is true [?]. The result is either a success, non-fatal failure or fatal failure (the latter aborts the current method).

19.2.2 Assertions

Assertions are the building bocks of tests, they look like macros and are used to make assertions about a class or function [?]. When an assertion fails

Google Test prints the line number of the assertion in the source file and the failure message (custom messages can also be provided) [?].

ASSERT and EXPECT

ASSERT this is a kind of assertion, when this fails it is considered fatal and the current function is aborted [?]. The documentation [?] later explains that these tests should be used when it makes not sense to continue and that they may cause memory leaks in the harness (giving secondary errors from leak detection software). For example, if we were testing reading a file we might first test if the file opens with an assert. If the file doesn't open any tests on reading from the file are pointless.

EXPECT this is another kind of assertion which does not fail fatally, this allows us to continue with other tests and report more than one failure [?]. For example if we're writing a vector class we might test getting and setting x, y and z in a single test with multiple expects, this would allow us to easily spot copy/paste errors between gets or sets.

Listing 19.1: Using ASSERT and EXPECT

```
/* Somwhere in the code under test two vectors have been
 created and populated (vector<int> x and vector<int> y).
*/

/* In our test */

ASSERT_EQ(x.size(), y.size()) << "Lenght of vectors is no
point testing element equality";

for(int i = 0; i < x.size(); i++)
{
    EXPECT_EQ(x[i], y[i]) << "Elements not the same at
    index " << i;
}
```

Note in the example (see 19.1) that the use of ASSERT in the above skips testing of individual element equality tests if the vectors are of different sizes. Also that as we're using EXPECT when testing elements we can report multiple elements which differ.

Types of Assertion

I've tried to restrict this list to the most useful, Googles documentation covers the full list in some detail.

Basic Assertions

These are the simplest kind of assertion, the test whether a condition (e.g. a boolean flat etc.).

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_TRUE(condition);	EXPERT_TRUE(condition);	condition is true
ASSERT_FALSE(condition);	EXPERT_FALSE(condition);	condition is FALSE

Binary Comparisons

These assertions compare two values and provide standard logical comparisons.

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_EQ(val1, val2);	EXPERT_EQ(val1, val2);	val1 == val2
ASSERT_NE(val1, val2);	EXPERT_NE(val1, val2);	val1 != val2
ASSERT_LT(val1, val2);	EXPERT_LT(val1, val2);	val1 < val2
ASSERT_LE(val1, val2);	EXPERT_LE(val1, val2);	val1 <= val2
ASSERT_GT(val1, val2);	EXPERT_GT(val1, val2);	val1 > val2
ASSERT_GE(val1, val2);	EXPERT_GE(val1, val2);	val1 >= val2

String Comparisons

Reasonably advanced string comparison assertions are provided, these work with C strings (char*) for std::string use EXPECT_EQ and EXPECT_NE.

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_STREQ(val1, val2);	EXPERT_STREQ(val1, val2);	same content
ASSERT_STRNE(val1, val2);	EXPERT_STRNE(val1, val2);	different conent
ASSERT_STRCASEEQ(val1, val2);	EXPERT_STRCASEEQ(val1, val2);	same content ignore case
ASSERT_STRCASENE(val1, val2);	EXPERT_STRCASENE(val1, val2);	different content ignore case

Included in the advanced documentation (rather than the primer,) are some floating point assertions. These by default for accuracy to 4 ULPs (Units of Last Place), but we have the option to specify the absolute error.

Fatal Assertion	Nonfatal Assertion	Verifies
ASSERT_FLOAT_EQ(val1, val2);	EXPERT_FLOAT_EQ(val1, val2);	same to 4 ULPs
ASSERT_DOUBLE_EQ(val1, val2);	EXPERT_DOUBLE_EQ(val1, val2);	same to 4 ULPs
ASSERT_NEAR(val1, val2, abs_error);	EXPERT_NEAR(val1, val2, abs_error);	same withing abs_error

19.2.3 Simple Tests

I'm going to skip this, time is short and test fixtures are far more useful (in my opinion) so we'll start with a simple test fixture. The example code provided are based on a class I wrote for the Group Maker application I showed you at the start of the year (and used to pick the groups).

19.2.4 Test Fixtures - multiples tests with the same data

Googles documentation [**?**] suggests that Test Fixtures should be used whenever more than one test operates on similar data. In the example we have a single class (though we could setup a group of classes for testing - integration testing,) and we'll initialise and test that class multiple times. In this case the class is the similar data, however we may create helper methods within the class to assist object creation/setup etc. which we can

also share.

Google Test relies on a large number of macro definitions and so we have to be careful to include all the relevant code. Some of the errors we get will refer to code generated by the macros, this is generated during compilation and without a detailed knowledge of them (and C++) they are difficult to understand.

The steps to create a test fixture according to the Google documentation are [**?**]:

1. Derive a class from ::testing::Test, starting its body with public or protected as we'll want to access fixtures from other classes (see A in 19.3).

2. Declare and objects you'd like to use in the class (see B in 19.3).

3. Include SetUp() and TearDown() the first sets up the objects required for testing while the second destroys them (see C in 19.3).

4. Include an empty constructor and destructor, these only need to be filled particlar circumstances e.g. needed to define constants which can't be changed (see the https://github.com/google/googletest/blob/master/googletest/docs/FAQ.md#should-i-use-the-constructordestructor-of-the-test-fixture-or-the-set-uptear-down-function) (see D in 19.3).

5. If required defined any subroutines to help, for example we might have a function which generates random numbers in a range for testing or provides an independent check of object behaviour. (An example, included for the sake if it is showin in E in 19.3)

Listing 19.2: Decleartion of A Test Fixture

```
#ifndef NAMELISTTESTHARNESS_H
#define NAMELISTTESTHARNESS_H

#include <limits.h>
#include <gtest/gtest.h>

#include <memory>

using std::shared_ptr;
```

```
class NameList;

class NameListTestHarness : public ::testing::Test
{
protected:
    virtual void SetUp();
    virtual void TearDown();

    shared_ptr<NameList> nameList;

public:
    NameListTestHarness();
    virtual ~NameListTestHarness();

    int fakeHelperMethod();
};

#endif
```

The test cases do not need to be added to the definition (.h file) they can simply be defined in the text fixture using the TEST_F() macro, I've provided one from the tutorial in **??**) where appropriate the labels from above have been included.

Listing 19.3: Decleartion of A Test Fixture

```
#include <limits.h>
#include "gtest/gtest.h"

#include "NameListTestHarness.h"
#include "NameList.h"

/* Reference: http://www.yolinux.com/TUTORIALS/Cpp-GoogleTest.html */

NameListTestHarness::NameListTestHarness()
{
    this->nameList = nullptr;
}

NameListTestHarness::~NameListTestHarness()
{
    this->nameList.reset();
}
```

```
void NameListTestHarness::SetUp()
{
    this->nameList = shared_ptr<NameList>(new NameList());
}

void NameListTestHarness::TearDown()
{
    this->nameList.reset();

    //Test the point has been deallocated.

    this->nameList = nullptr;
}

int NameListTestHarness::fakeHelperMethod()
{
    //This is a fake, just to show how its done.
    return 0;
}

TEST_F(NameListTestHarness,constructorTest)
{
    ASSERT_EQ(this->nameList->getSize(),0);

    ASSERT_EQ(this->nameList->begin(), this->nameList->end());
}
```

The above example **??** shows a single test fixture, the code provided for the tutorial includes a lot more.

The Google Test Process

For each of the tests defined with TEST_F() Google Test undertakes the following [**?**]:

1. Create a new text fixture object
2. Initialise this using the SetUp() method
3. Run the test (remember it only does one at a time)
4. Clean up by calling tearDown()
5. Delete the fixture, note that google NEVER re-uses a test object

19.3 Setup for Google Test

The setup process depends on the availability of pre-built libraries for your system, where these exist they simply need to be made available when compiling (i.e. provide the compiler with the location of the headers and library files, and instruct it to link them).

Fedora 24

Installation is reasonably simple, we install **gtest** the runtime library and **gtest-devel** the development libraries (headers and libraries to compile against). Note: You will need root access to do this, there should be no need in the lab as the libraries are already installed.

Listing 19.4: Install Commands

```
$ sudo dnf install gtest
$ sudo dnf install gtest-devel
```

When linking against these libraries using g++ we only really need to pass the compiler the $-lgtest$ and $-lgtest_main$ arguments to link the libraries as they are installed to the default directories for headers/libraries used by the compiler.

19.3.1 Windows 10 / Visual Studio 2015

I used an online guide this was written for Visual Studio 2010 but appears to work.

I will write my own guide for this - at present my VM is still installing Visual Studio 2015!

19.4 Exercises

To gives you some practice using git/Bitbucket I've provided the code for today's exercises as a number of public repositories on bitbucket. I suggest you fork these to your bitbucket account, then clone them to your desktop before undertaking the exercise and pushing them back.
NOTE: At the time of writing the Visual Studio 2015 repo is empty, my Windows 10 VM is still installing Visual Studio 2015!

1. Pull the relevant gtest project from Google, I've tried to make this as flexible as possible in term of project type and OS.

- Linux Makefile version
- Qt Creator version
- Linux Microsoft Code
- Visual Studio 2015

2. Get the harness built and working, if something fails try to work out why.

3. There is a flaw in the harness (I noticed recently,) there is no test to ensure a valid file-name.

4. Add this test, then look for other potential issues.

5. Using this as a template test something from your group project, another assignment from last year.

Bibliography

[RUP] *The Rational Unified Process Product. The browser-based online documenation for the RUP, sold by IBM, and previously by Rational Corp.*

[2] (1998). Data protection act 1998.

[3] Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. (2002). Agile software development methods: Review and analysis. Available from: `http://www.pss-europe.com/P478.pdf`.

[4] Aghaei, S., Nematbakhsh, M. A., and Farsani, H. K. (2012). Evolution of the worlds wide web: From web 1.0 to web 4.0. *International Journal of Web and Semantic Technology (IJWesT)*.

[5] Alexander, I. (2004). *Scenarios, Stories Use Cases*, chapter Part I: Overview, pages 3–24. John Wiley & Sons, Ltd.

[Ambler] Ambler, S. History of the unified process [online, cited 28-9-2016]. Available from: `http://www.enterpriseunifiedprocess.com/essays/history.html`.

[7] Ambler, S. (2002a). *Agile Modelling: Evffective Practices for eXtreme programming and the Unified Process.* John Wiley & Sons, Ltd. ISBN: 0-471-20282-7.

[8] Ambler, S. (2002b). *Agile Modelling: Evffective Practices for eXtreme programming and the Unified Process*, chapter Agile Modelling and the Unified Process, pages 223–232. John Wiley & Sons, Ltd. ISBN: 0-471-20282-7.

[9] Ambler, S. (2002c). *Agile Modelling: Evffective Practices for eXtreme programming and the Unified Process*, chapter Nuturing an Agile Culture, pages 89–100. John Wiley & Sons, Ltd. ISBN: 0-471-20282-7.

[10] Armour, P. (2001). The business of software: Zeppelins and jet planes: A metaphor for modern software projects. *Commun. ACM*, 44(10):13–15. Available from: `http://doi.acm.org/10.1145/383845.383849`.

[11] Armstrong, M. (2009a). *Armstrong's Handbook of Managment and Leadership*, chapter The Process of Management and Leadership, pages 3–14. Kogan Page, London, UK, 2nd edition.

[12] Armstrong, M. (2009b). *Armstrong's Handbook of Managment and Leadership*, chapter The Role of the Leader, pages 31–40. Kogan Page, London, UK, 2nd edition.

[13] Banach, R. (2007). Guest editorial: Formal methods. *Journal of Universal Computer Science*, 13(5):593–606.

[14] Begel, A. and Nagappan, N. (2008). Pair programming: what's in it for me? In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 120–128. ACM.

[15] Boehm, B. and Turner, R. (2003). Observations on balancing discipline and agility. In *Agile Development Conference, 2003. ADC 2003. Proceedings of the*, pages 32–39. IEEE.

[16] Boehm, B. and Turner, R. (2004). *Balancing Agility and Discipline: A Guide for The Perplexed.*, chapter Appendix A, pages 165–194. Addison-Wesley. ISBN 0-321-18612-5.

[17] Bohem, B. (1989). Software risk management. In Ghezzi, C. and McDermid, J., editors, *2nd European Software engineering Conference*, Lecture Notes in Computer Science, pages 1–19.

[18] Bohem, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5):61–72.

[19] Brooks, J. and P., F. (1987). No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19. Available from: `http://dx.doi.org/10.1109/MC.1987.1663532`.

[20] Chacon, S. and Straub, B. (2014a). *Pro Git*, chapter Getting Started, pages 27–42. Apress, Berkely, CA, USA, 2nd edition.

[21] Chacon, S. and Straub, B. (2014b). *Pro Git*, chapter Git Basics, pages 43–81. Apress, Berkely, CA, USA, 2nd edition.

[22] Chacon, S. and Straub, B. (2014c). *Pro Git*, chapter Git Basics, pages 81–125. Apress, Berkely, CA, USA, 2nd edition.

[23] Clemente, P. J. and Hernández, J. (2003). Aspect component based software engineering. In *The Second AOSD Workshop on Aspects Components and Patterns for Infrastructure Software ACP4IS*, pages 39–42.

[24] Committee, I. S. C. et al. (1990). Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). los alamitos. *CA: IEEE Computer Society*.

[25] DeRemer, F. and Kron, H. (1975). Programming-in-the large versus programming-in-the-small. *SIGPLAN Not.*, 10(6):114–121. Available from: `http://doi.acm.org/10.1145/390016.808431`.

[26] Douglass, B. P. (2004a). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Object Orientation with UML 2 - Structural Aspects, pages 77–139. Object technology. Addison Wesley, 3rd edition.

[27] Douglass, B. P. (2004b). *Real Time UML: Advances in the UML for Real-Time Systems*. Object technology. Addison Wesley, 3rd edition.

[28] Douglass, B. P. (2004c). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Analysis of Real-time Systems, pages 227–331. Object technology. Addison Wesley, 3rd edition.

[29] Douglass, B. P. (2004d). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Anlysis of Real-time Systems, pages 331–399. Object technology. Addison Wesley, 3rd edition.

[30] Douglass, B. P. (2004e). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Object Orientation with UML 2 - Behavioural Elements, pages 139–215. Object technology. Addison Wesley, 3rd edition.

[31] Douglass, B. P. (2004f). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Architectural Design, pages 473–526. Object technology. Addison Wesley, 3rd edition.

[32] Douglass, B. P. (2004g). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Mechanistic Design, pages 527–588. Object technology. Addison Wesley, 3rd edition.

[33] Ekvall, G. and Arvonen, J. (1991). Change-centered leadership: An extension of the two-dimensional model. *Scandinavian Journal of Management*, 7(1):17–26.

[34] Folwer, M. Design dead [online]. (2004) [cited 2016.08.23]. Available from: `http://www.martinfowler.com/articles/designDead.html`.

[35] Folwer, M. The new methodology [online]. (2005) [cited 2016.08.23]. Available from: `http://martinfowler.com/articles/newMethodology.html`.

[36] Folwer, M. Semantic diffusion [online]. (2006) [cited 2016.08.23]. Available from: `http://martinfowler.com/bliki/SemanticDiffusion.html`.

[37] Folwer, M. The flacid scrum [online]. (2009) [cited 2016.08.23]. Available from: `http://martinfowler.com/bliki/FlaccidScrum.html`.

[38] Fowler, M. (2003a). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[39] Fowler, M. (2003b). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Use Cases, pages 79–85. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[40] Fowler, M. (2003c). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Introduction, pages 14–26. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[41] Fowler, M. (2003d). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Class Diagrams, pages 79–85. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[42] Fowler, M. (2003e). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Sequence Diagrams, pages 47–56. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[43] Fowler, M. (2003f). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Class Diagrams: Advanced Concepts, pages 56–72. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[44] Freeman, S. and Price, N. (2010). *Growing Object-Orientated Sfotware, Guided by Tests*, chapter What is the point of test driven development?, pages 1–29. Addison Wesley.

[45] Graham, D., van Veenendaal, E., Evans, I., and Black, R. (2006). *Foundations of Software Testing*, chapter Testing throughout the software lifcycle, pages 35–56. Cengage Learning, EMEA.

[46] Haberberg, A. and Rieple, A. (2008). *Strategic management: theory and application*, chapter What is Strategic Management, pages 46–100. Oxford University Press.

[47] Hall, A. (2007). Guest editorial: Formal methods. *Journal of Universal Computer Science*, 13(5):669–678.

[48] Haxthausen, A. (2010). An introduction to formal methods for the development of safety-critical applications. Technical Report DK-2800 Lyngby, DTU Informatics, Technical University of Denmark.

[49] Larman (2014a). *Applying UML and Patterns*, chapter Iterative, Evolutionary, and Agile, pages 17–41. Prentice Hall PTR.

[50] Larman (2014b). *Applying UML and Patterns*, chapter Inception is Not the Requirements Phase, pages 47–53. Prentice Hall PTR.

[51] Larman (2014c). *Applying UML and Patterns*, chapter Evolutionary Requirements, pages 47–61. Prentice Hall PTR.

[52] Larman (2014d). *Applying UML and Patterns*, chapter Use Cases, pages 61–101. Prentice Hall PTR.

[53] Larman (2014e). *Applying UML and Patterns*, chapter Relating Use Cases, pages 493–501. Prentice Hall PTR.

[54] Larman (2014f). *Applying UML and Patterns*, chapter Other Requriements, pages 101–123. Prentice Hall PTR.

[55] Larman (2014g). *Applying UML and Patterns*, chapter Iteration 1 - Basics, pages 123–131. Prentice Hall PTR.

[56] Larman (2014h). *Applying UML and Patterns*, chapter Domain Models, pages 131–172. Prentice Hall PTR.

[57] Larman (2014i). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 173–181. Prentice Hall PTR.

[58] Larman (2014j). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

[59] Larman (2014k). *Applying UML and Patterns*, chapter Requirements To Design - Iteratively, pages 131–172. Prentice Hall PTR.

[60] Larman (2014l). *Applying UML and Patterns*, chapter Logical Architecture and UML Package Diagrams, pages 197–212. Prentice Hall PTR.

[61] Larman (2014m). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

[62] Larman (2014n). *Applying UML and Patterns*. Prentice Hall PTR.

[63] Larman (2014o). *Applying UML and Patterns*, chapter On To Object Design, pages 213–220. Prentice Hall PTR.

[64] Larman (2014p). *Applying UML and Patterns*, chapter Object Design Examples with GRASP, pages 312–362. Prentice Hall PTR.

[65] Larman (2014q). *Applying UML and Patterns*, chapter GRASP: Designing Objects with Responsiblities, pages 312–362. Prentice Hall PTR.

[66] Larman (2014r). *Applying UML and Patterns*, chapter Case Studies, pages 41–47. Prentice Hall PTR.

[67] Larman (2014s). *Applying UML and Patterns*, chapter GRASP: More Objects with Responsiblities, pages 401–343. Prentice Hall PTR.

[68] Larman (2014t). *Applying UML and Patterns*, chapter UML Activity Diagrams and Modeling, pages 477–485. Prentice Hall PTR.

[69] Larman (2014u). *Applying UML and Patterns*, chapter Mapping Designs to Code, pages 369–385. Prentice Hall PTR.

[70] Larman (2014v). *Applying UML and Patterns*, chapter UML Tools and UML as Blueprint, pages 395–401. Prentice Hall PTR.

[71] Larman (2014w). *Applying UML and Patterns*, chapter Test Driven-Development and Refactoring, pages 385–395. Prentice Hall PTR.

[72] Larman (2014x). *Applying UML and Patterns*, chapter UML State Machine Diagrams and Modelling, pages 485–493. Prentice Hall PTR.

[73] Leveson, N. G. (1995). *Safeware: System Safety and Computers*, chapter Computers and Risk, pages 21–38. Addison Wesley.

[74] Martin, B. The true corruption of agile [online]. (2014). Available from: https://8thlight.com/blog/uncle-bob/2014/03/28/The-Corruption-of-Agile.html.

[75] McDowell, C., Werner, L., Bullock, H., and Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, pages 38–42, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/563340.563353.

[76] Moore, J. H. (2008). *Information Technology and Moral Philiospy*, chapter Why we Need Better Eithics for Emerging Technologies, page 26. Cambridge University Press.

[77] Mullins, L. J. (2008a). *Essentials of Organsiational Behaviour*, chapter The Nature of Leadership, pages 220–299. Prentice Hall, Harlow, England.

[78] Mullins, L. J. (2008b). *Essentials of Organsiational Behaviour*, chapter Workgroups and Teams, pages 217–295. Prentice Hall, Harlow, England.

[79] Mullins, L. J. (2008c). *Essentials of Organsiational Behaviour*, chapter Organisational Strategy, Ethics and Responsiblities, pages 350–460. Prentice Hall, Harlow, England.

[80] Mullins, L. J. (2008d). *Essentials of Organsiational Behaviour*, chapter Organisational Control and Power, pages 421–459. Prentice Hall, Harlow, England.

[81] Mullins, L. J. (2008e). *Essentials of Organsiational Behaviour*, chapter Organisational Development, Culture and Change, pages 462–494. Prentice Hall, Harlow, England.

[82] Myers, G. J. (2004a). *The ART of Software Testing*, chapter The Psychology and Economics of Program Testing, pages 5–20. John Wiley & Sons, Ltd.

[83] Myers, G. J. (2004b). *The ART of Software Testing*. John Wiley & Sons, Ltd.

[84] Myers, G. J. (2004c). *The ART of Software Testing*, chapter Test-Case Design, pages 43–91. John Wiley & Sons, Ltd.

[85] Myers, G. J. (2004d). *The ART of Software Testing*, chapter Module (Unit) Testing, pages 91–123. John Wiley & Sons, Ltd.

[86] Myers, G. J. (2004e). *The ART of Software Testing*, chapter Higher-Order Testing, pages 129–156. John Wiley & Sons, Ltd.

[87] Naur, P. and Randell, B. (1968). Software engineering: Report on a conference sponsored by the nato science commitiee. Technical report, NATO, Garmisch, Germany.

[88] OMG (2015). Unified modelling language: Version 2.5. Available from: `http://www.omg.org/spec/UML/2.5/`.

[89] Ratzmann, M. and De Yong, C. (2003a). *Software Testing and Internationalization*, chapter Rapid Application Testing, pages 13–24. Lemoine International, Inc.

[90] Ratzmann, M. and De Yong, C. (2003b). *Software Testing and Internationalization*, chapter Rapid Application Testing, pages 71–147. Lemoine International, Inc.

[91] Robbins, S. P. (2001a). *Organisational Behaviour*, chapter Leadership and Trust, pages 312–349. Prentice Hall, New Jersey, USA, 9th edition.

[92] Robbins, S. P. (2001b). *Organisational Behaviour*, chapter Motivation: From Concept to Applications, pages 118–215. Prentice Hall, New Jersey, USA, 9th edition.

[93] Rollinson, D. (2008a). *Organisational Behaviour and Analysis: An Intergrated Approach*, chapter Leaders and Leadership: The Foundations, pages 349–393. Prentice Hall, Harlow, England, 4th edition.

[94] Rollinson, D. (2008b). *Organisational Behaviour and Analysis: An Intergrated Approach*, chapter Contingency Theories and Recent Approaches, pages 368–392. Prentice Hall, Harlow, England, 4th edition.

[95] Rollinson, D. (2008c). *Organisational Behaviour and Analysis: An Intergrated Approach*, chapter Organisational Structure, pages 501–530. Prentice Hall, Harlow, England, 4th edition.

[96] Rollinson, D. (2008d). *Organisational Behaviour and Analysis: An Intergrated Approach*, chapter Power, Politics and Conflict, pages 501–530. Prentice Hall, Harlow, England, 4th edition.

[97] Rollinson, D. (2008e). *Organisational Behaviour and Analysis: An Intergrated Approach*, chapter Work Motivation and Job Design, pages 194–226. Prentice Hall, Harlow, England, 4th edition.

[98] Rollinson, D. (2008f). *Organisational Behaviour and Analysis: An Intergrated Approach*, chapter Work Motivation: Process Theories, pages 227–247. Prentice Hall, Harlow, England, 4th edition.

[99] Rollinson, D. (2008g). *Organisational Behaviour and Analysis: An Intergrated Approach*, chapter Organisational Culture and Climate, pages 591–628. Prentice Hall, Harlow, England, 4th edition.

[100] Royce, W. W. (1970). Managing the development of large software systems. In *proceedings of IEEE WESCON*, volume 26, pages 328–338. Los Angeles.

[101] Rushby, J. (1993). Formal methods and the certification of critical systems. Technical Report CSL-93-7, SRI International, Menlo Park, CA. Also issued as Formal Methods and Digital Systems Validation, NASA CR 4551.

[102] Schach, S. R. (2007a). *Object Orientated Software Engineering*, chapter The Scope of Object-Orientated Software Engineering, pages 3–31. McGraw-Hill, New York, USA.

[103] Schach, S. R. (2007b). *Object Orientated Software Engineering*, chapter Software Life-Cycle Models, pages 32–67. McGraw-Hill, New York, USA.

[104] Schach, S. R. (2007c). *Object Orientated Software Engineering*, chapter The Software Process, pages 68–100. McGraw-Hill, New York, USA.

[105] Schach, S. R. (2007d). *Object Orientated Software Engineering*, chapter The Software Process, pages 287–331. McGraw-Hill, New York, USA.

[106] Schach, S. R. (2007e). *Object Orientated Software Engineering*. McGraw-Hill, New York, USA.

[107] Silberschatz, A., Galvin, P. B., and Gagne, G. (2008). *Operating System Concepts*, chapter Threads, pages 127–152. Wiley Publishing, 8th edition.

[108] Sohaib, O., Hussain, W., and Badini, M. K. (2011). User experience (ux) and the web accessibility standards. *International Journal of Computer Science Issues*, 8(3):584–587.

[109] Sommerville, I. Software engineering history [online]. (2010). Available from: `https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/History/`.

[110] Sommerville, I. (2011a). *Software Engineering*, chapter Introduction, pages 3–27. Addison Wesley, 9th edition.

[111] Sommerville, I. (2011b). *Software Engineering*, chapter Software Processes, pages 27–55. Addison Wesley, 9th edition.

[112] Sommerville, I. (2011c). *Software Engineering*, chapter Agile Software Development, pages 56–81. Addison Wesley, 9th edition.

[113] Sommerville, I. (2011d). *Software Engineering*, chapter Requirements Engineering, pages 82–117. Addison Wesley, 9th edition.

[114] Sommerville, I. (2011e). *Software Engineering*. Addison Wesley, 9th edition.

[115] Stewart, J. (2006). Transformational leadership: An evolving concept examined through the works of burns, bass, avolio, and leithwood. *Canadian Journal of Educational Administration and Policy*, 54:1–29.

[116] Storey, N. (1996). *Safety-Critical Computer Systems*, chapter Formal Methods, pages 271–309. Addison-Wesley, Harlow, England.

[117] Van Vliet, H. (2007a). *Software engineering: principles and practice*, chapter Introduction, pages 5–32. Wiley New York.

[118] Van Vliet, H. (2007b). *Software engineering: principles and practice*. Wiley New York.

[119] Van Vliet, H. (2007c). *Software engineering: principles and practice*, chapter The Software Life Cycle Revisited, pages 45–77. Wiley New York.

[120] Van Vliet, H. (2007d). *Software engineering: principles and practice*, chapter People Managmenet and Team Organisation, pages 89–107. Wiley New York.

[121] Von Solms, R. and Van Niekerk, J. (2013). From information security to cyber security. *Computers & Security*, 38:97–102.

[122] Westland, J. C. (2002). The cost of errors in software development: Evidence from industry. *The Journal of Systems and Software*, 62:1–9.

[123] Zaleznik, A. (1977). Managers and leaders: Are they different@. *Harvard Business Review*.