Software Engineering Principles Notes 2016-2017

Version: 0.40

Glenn L. Jenkins

Print Date: November 3, 2016

©2016 Dr. Glenn L. Jenkins

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 [44]. 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

[35]
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

[18]

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

[38]
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

[44]
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 [44].

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) [44].

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 [49]:

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 [44]. Computers are through software "logically malleable" and can undertake any task that we can design, train or evolve them to do [34][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 [44]. 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 [44] 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 [38]. 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 [38]. 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 [41].

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 [50].

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 [50].

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' [35]. 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 [44]. 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 [35] 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 [43] 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 [27]
- **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 [27].
  - Paper proposes an iterative development method rather than waterfall (published 1968 at the IBM T.J. Watson Research Centre) [27].
- **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) [38].
  - Some software written with IID methods including USA Space Shuttle flight controls (built in 17 four week iterations) [27].
  - Harlen Mills (a prominent thinker in Software Engineering,) wrote about the failure of waterfall and the need for IID [27].
  - USA Department of Defence adopts the waterfall (DoD-2167) [27].

- **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 [27].

  - 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. [27].

- **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** [44, 3]
  - 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** [42]
  - 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 [16] (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 [44] 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 [44]. In such systems the organisation who create the software control the specification [44] 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 [44]. 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 [44]. Sommerville [44] 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 44 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 [13] suggest that development methods exist on a continuum from adaptive to predictive, or agile to plan driven. Both approaches have their homeground [12], 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 [44]. Sommerville defines these as [44]:

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"[38],) are some combination of the above some organised in different ways and undertaken at different levels of detail [44]. 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 [38], 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 [38]:

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 [38] 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 [38]. See 1.3.

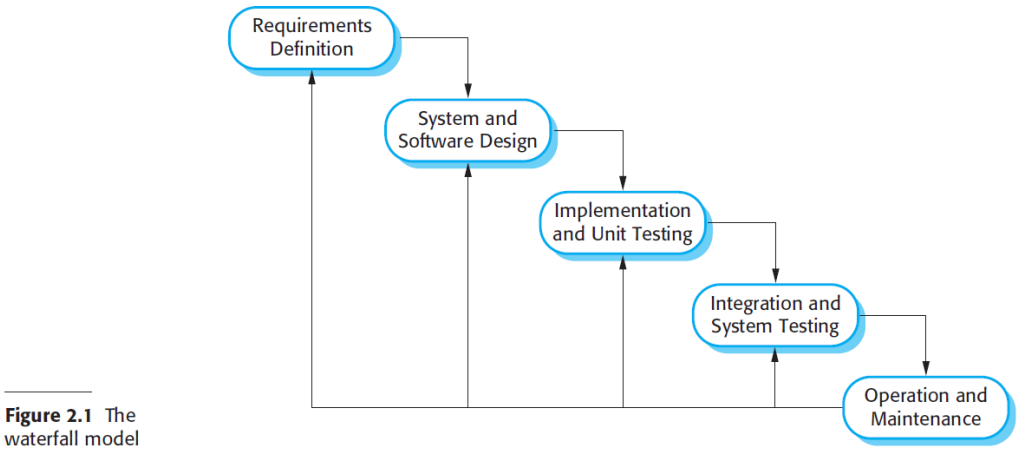

Figure 1.2: from Sommerville [45]: The Waterfall Model

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 [38]. 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. [49] 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 [19]. 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.

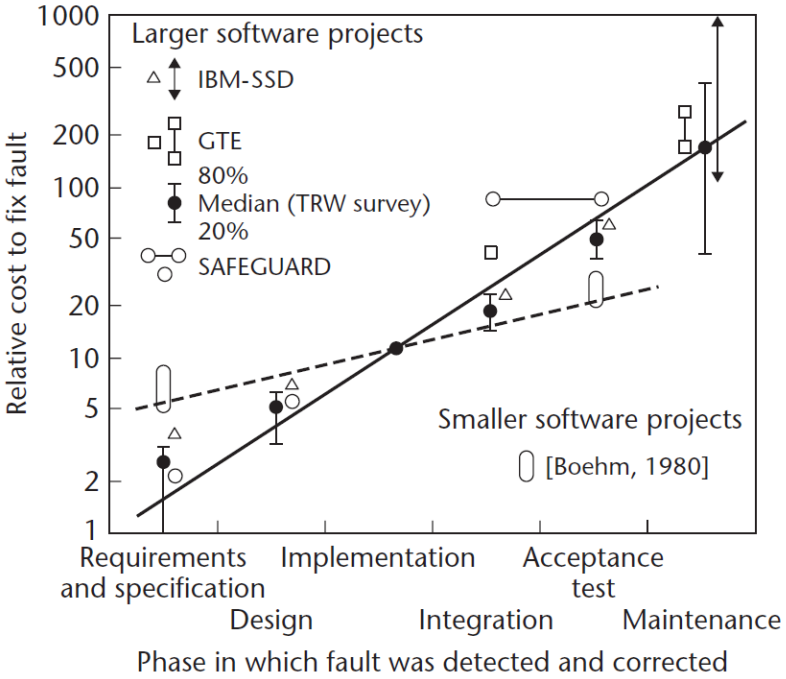

Figure 1.3: from [38].

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

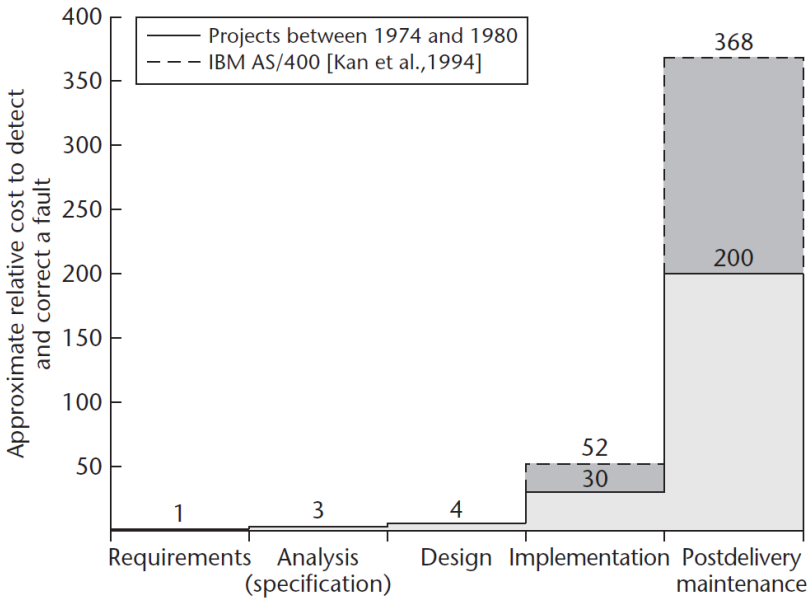

Figure 1.4: from [38].

ever increasing complexity.

- **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 general 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 [49] 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 [19] make this distinction in terms of programming in the small and programming in the large. They and others [38] 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 [38] 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 [44] 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 [45] 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) [38] 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) [38]. 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 nth 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)[38].

- **Product and Artefact** Product is a term often given to a non-trivial piece of software [38]. The term artefact is used to describe any component of this software product e.g. a specification document, a code

module or a manual) [38].

- **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" [38]. 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" [38]. 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.,) [38]. 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 [38]. 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 [38].

- **Software Process** A software process is a sequence of activities which leads to the production of software and is composed of activities [44]. 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 [45].

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 [44]. Sommerville defines these activities as [45]:

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 [45]:

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) [45]. 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 [45]. This is a critical stage, misunderstanding what the customer wants or omitting some crucial detail will result in the development of the wrong system [45] (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) [45].
There are four main activities (see 2.1) [45]:

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

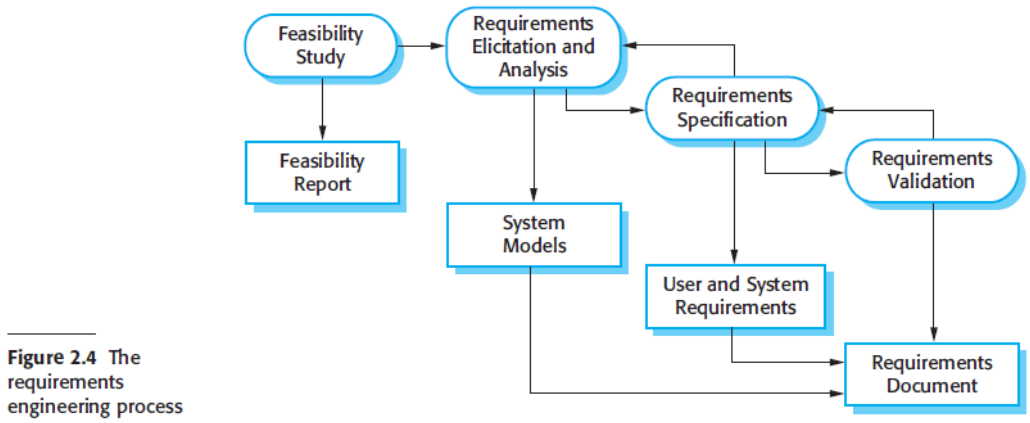

Figure 2.1: from [45].

specification (missing details,) or validation (e.g. conflicting requirements,) will require stakeholders are re-consulted [45].
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" [4]. 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 [4]. 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 [45]. 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 [45].

The design process is always an iterative one [45] 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 45 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.

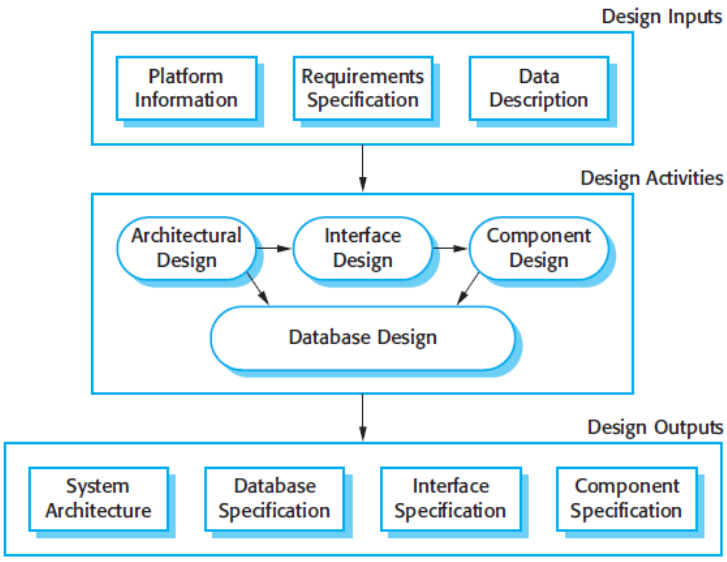

Figure 2.5 A general model of the design process

Figure 2.2: from [45].

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 [45].

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 45 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 [20].

Sommerville 45 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 [45]. Some programmers will create prototypes of challenging components to test ideas/technologies these may form part of the solution or be discarded. Sommerville 45 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?"
[14].

Dynamic testing (where a system or sub-system,) is tested using simulated data is the principle validation technique [45]. 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 45 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 45 are to my mind a little coarse, so I've modified them in line with ISTQB based definitions from here [24]:

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 [49]. 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 [45]. 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 [45].
Van Vliet 49 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 45 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' [45] 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 51, uses the term generic model in much the same way.
General Software Process Models [45]:

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,) [17]).

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 [45]. 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 [35]. 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 [35], a point we'll consider again when looking at iterative methods in general.

2.5.2 Coping with Change

[45] 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 [45]. 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 [45].

Change Avoidance

Here the Software Process includes activities which aim to anticipate changes before significant rework is required, for example prototyping [45]. 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 [45]. 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 12 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 [51]. 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 [51].

"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 49 describing the analogy of Armour 9.

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 [51]. 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 [51]. Feedback from the user can then be used to guide the selection of the next high value increment to be undertaken [51].

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 [38].

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 [45]. 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 [46]. 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 [46].

Sommerville 46 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 [46]. 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 [46].

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 27, 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 [38] draws on a number of research papers which show that more than half of projects displayed one or more symptoms of the software crisis.

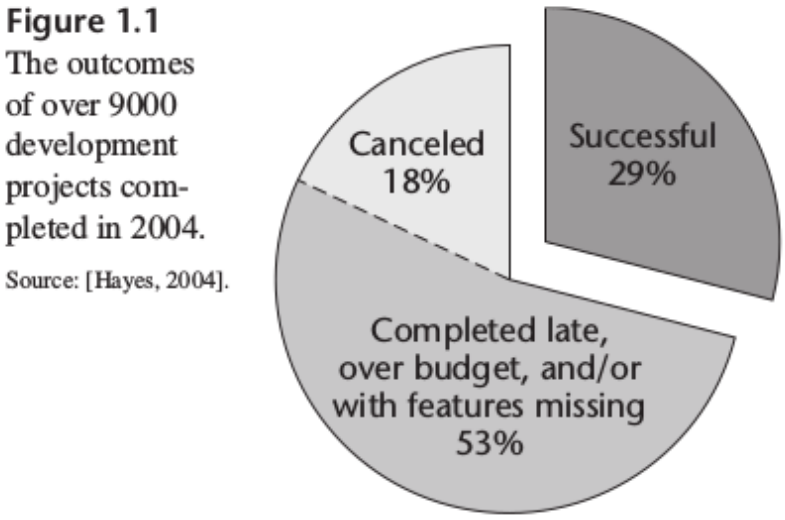

Figure 2.3: from [38].

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 [46].

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 45 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 [27]. 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 [27]. Larman 27 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 [46], 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* [39]. Key participants included Beck (creator of Extreme programming,) Cockburn (Crystal) and Schwaber (Scrum) [39]. Rather than outline a specific software process they produced a set of underlying principles which were common to their individual approaches [39]. 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 46).
Individual methods vary quite widely, while all follow the above common principle each has different processes to achieve this [46]. 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 [38] 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 [40]. This may seem far too simpler distinction and the reality most projects employ a mix of both [12].

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 51 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 [38]. Examples (again from [38]:

- **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 12 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 [12] 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 [12]. 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 [12], 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.

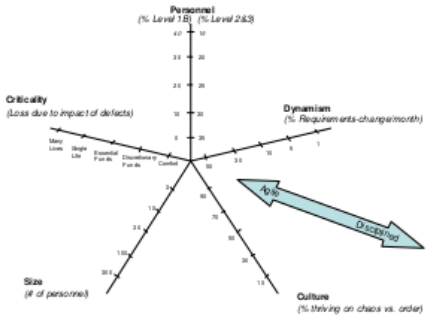

Figure 1. Dimensions affecting method selection

Figure 2.4: A collection of figures/tables from [12], 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 [51]. 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 [51].

"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 49 describing the analogy of Armour 9.

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 [51]. 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 [51]. Feedback from the user can then be used to guide the selection of the next high value increment to be undertaken [51].

3.3 Examples - Plan Driven Approaches

3.3.1 Waterfall

This is a summary of Sommerville 45 (section 2.1.1) and [39] (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 [45]. This is a plan driven process with each phase being planned before execution [45] and no phase being complete until the associated documentation is complete and approved (by the Software Quality Assurance (SQA) group) [38]. Testing is inherent in each phase (and in theory,) continuous [39], 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!)

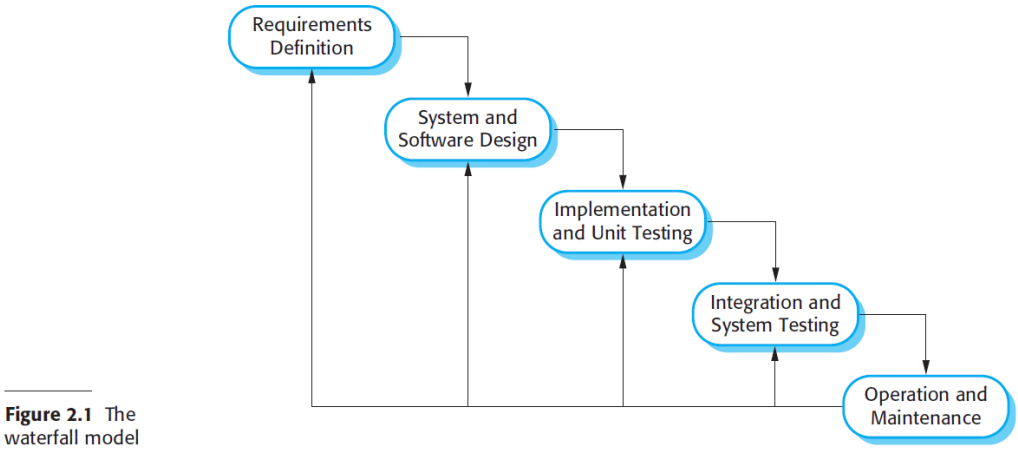

Figure 3.1: from Sommerville [45]: The Waterfall Model

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."

[45, p.30]
Conceptually each phase is checked and signed off (see 3.3.1,) but in practice there is some feeding overlap between stages [45]. 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 [45]. 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 [45].

Advantages

1. Progress is visible (from the production of documents) and can be compared to the plan [45].

2. If hardware etc. is being developed alongside using a traditional engineering approach waterfall allows for a common management model [45].

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 [45].

2. Repeating stages is costly [45].

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

Larman 27 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 [26]. 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 [45].

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 [26]. Though Formal Methods, like so many other things in Software Engineering, can be traced to the Software Crisis [10]. There is a suggestion that while methods for managing software project and the development of sound processes continued to develop, mathematical modelling was shunned [10]. Banach 10 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 [48]. Specifications should be complete, unambiguous, consistent and correct which is difficult to achieve with written

documents [48]. 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 [48]. Storey 48 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 37 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 [48]. 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 [48]. 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) [48].

Haxthausen 26 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 [53]. This is echoed by Hall 25 who suggests that the use of Formal Methods for error reduction in the early stages offers the most tangible benefits to their use. Leveson 31 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 45 (section 2.3.3), I suggest you read this and possibly the original paper by Bohem 15.

Bohem 15 proposed the spiral model in 1988 , this risk driven which each loop of the spiral representing a phase of the software process [45], 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 [45]. 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 [45].

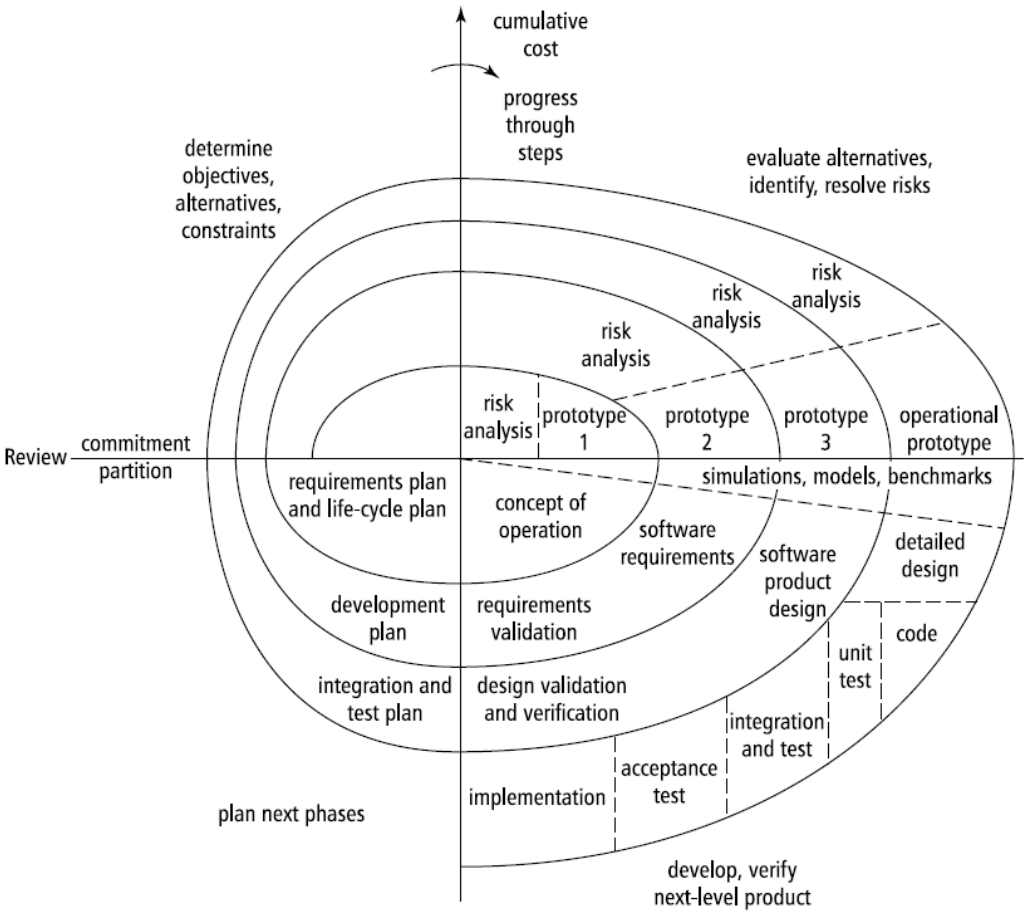

Figure 3.2: from Sommerville [45] The Spiral Model

The sectors of the spiral are outlined by Sommerville 45:

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 [39]. 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 39 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 51 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 [51]:

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).

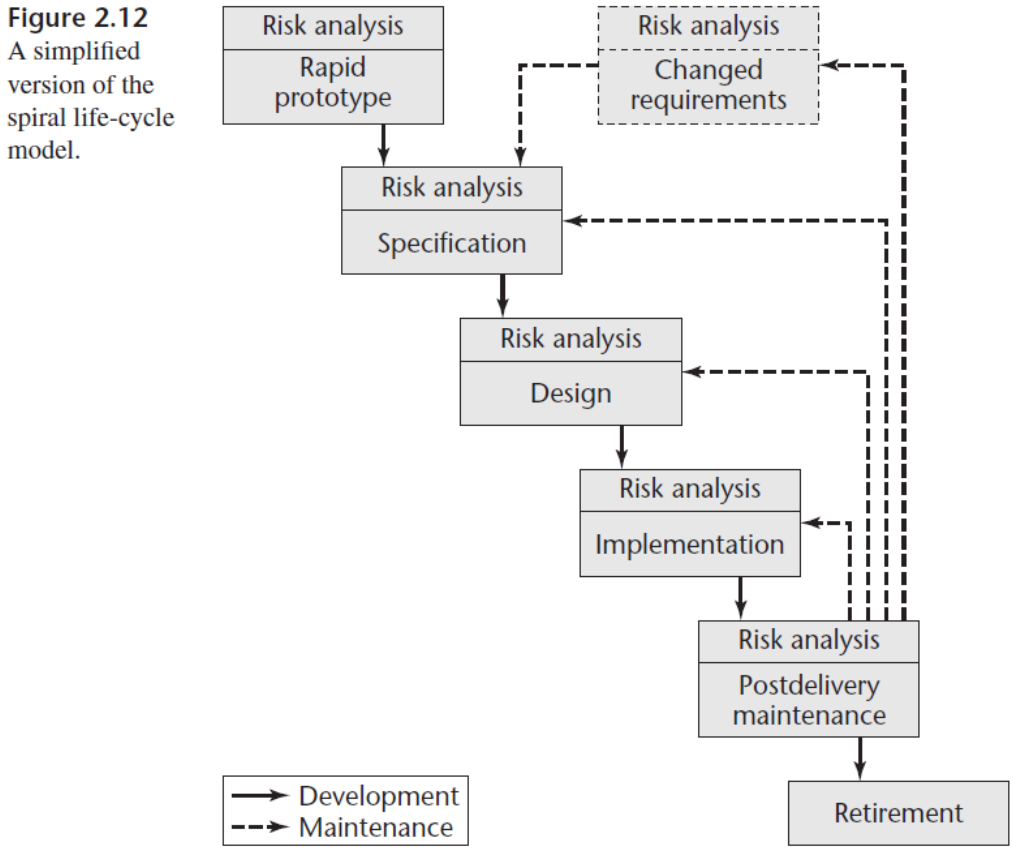

Figure 3.3: from Schach [39]: Spiral is modified waterfall

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) [39].

2. Intended for large projects and has been used in a large number of projects and produced and shown to increase productivity [39].

3. Maintained becomes another cycle there is no distinction between this and development, this prevents it becoming a second class activity [39].

Disadvantages

1. Not all risks can be mitigated, e.g. team performance, hardware promises etc. [40].
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 [39].
3. Risk analysis is costly and requires skilled personnel [39].

Larman 27 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 [39]. 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 [39].
- Maintainance is difficult and error prone [39].
- The above are complicated by the lack of design documentation [39].
-

While this approach can be safely used with small projects (100-200 lines), it is unsatisfactory for projects of any size [39]. 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 [39].

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 [46], 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* [39]. Key participants included Beck (creator of Extreme programming,) Cockburn (Crystal) and Schwaber (Scrum) [39]. Rather than outline a specific software process they produced a set of underlying principles which were common to their individual approaches [39].

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 46).
Individual methods vary quite widely, while all share the above values each has different practices/methods to achieve this [46]. These are summarised in 3.4 from [46].

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 [46].

Agile methods are recommended [38] 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 [40].

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 [32]. 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 32 argues that agile culture is expressed though a set of practices and the erosion of these practices is has diluted the culture. Folwer 23 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 22 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 32 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 [51]:

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 [51]. Rapid Application Development and DSDM emphasize customer collaboration and the role of people in the process [51]. 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 [51]. Prototypes are often used to explore key features of the problem space or the technologies available [51].

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 [51]. in *throw away prototyping* the roughly constructed code is disposed of and a system is designed and coded [51]. 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* [51]. 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 [51].

There are some positives however, the users and developers take a lot of ownership of the system and there is a lot of communication [51]. Agile methods solve this problem with re-factoring, essentially going back and improving each development cycle to ensure the design remains robust [51].

3.4.2 RAD

Rapid Application Development (RAD) like many other iterative methods emphasises customer involvement, prototyping, reuse, automated tools and

small development teams [51] 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 [51]. Another key feature of RAD is the Joint Requirements Planning workshop, joint in that this involves both users and developers [51]. 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 [51]:

- **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 [51]. RAD then calls for a small focused SWAT (Skilled With Advanced Tools,) team to work on developing the system, for details see: [51, 52]

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 [51]. 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 [51]:

- **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 51 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 [51], 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,) [51]. 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 [51]. McDowell et al. 33 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" [11],) checks the code produced identifying misspellings, logical and syntactical errors and ensuring that the code is in keeping with the design [33]. After a designated period the programmers exchange roles, possibly moving to work with others within the programming team [33].

A full list of XP practices is shown in 3.5.

XP teams are described [51] 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. [51]. The desirable features are described

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 [51]: List of XP Practices

in stories on index cards and a number of these will be realised per release [51]. 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 [51]. 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 [51].

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 [51]. 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 [51].

XP is based on five key principles [51]:

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 12 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 51 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 [27]. 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 [45]. 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 refactoring can fit within a UP project [27].

Larman 27 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 [45]. The phases of UP are more closely equated to business and the development of the right product [45].

A Slightly Confusing Diagram

Sommerville 45 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.

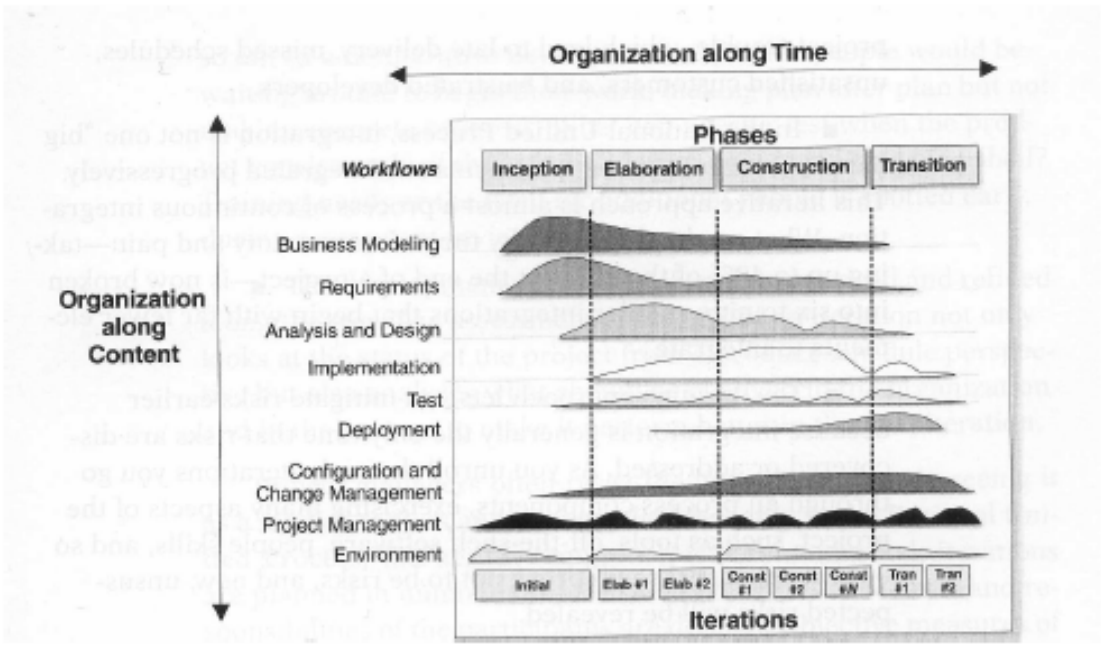

Figure 3.9 Process structure of RUP (*Source: P. Kruchten, The Rational Unified Process, An Introduction*, 2003, Addison-Wesley)

Figure 3.6: from Vilet [51]: RUP diagram

Critical UP Practices / Concepts

Phases in UP

Up has four phases, each containing one or more iterations [27]:

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.

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 [27]. 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) [27].

The UP Disciplines

The unified process defines work activities (such as drawing a particular diagram,) writing a some code within Disiplines (originally called Workflows) [27]. 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.) [27].
This module (shamelessly,) follows Larman 27 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 [27], 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.) [27] 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 45 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.
- 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

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 [45]: RUP workflows

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 6 proposed agile modelling, this is described by Larman 27 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 [27].

Larman 27 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 [27]. Ambler 7 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 27 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 [21], 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 27 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" [8], 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 [27]. 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 [45].

To remind you here four phases of UP, each one may contain one or more iterations [27]:

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 6 this is described by Larman 27 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 [27]. 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 [27]. Ambler 7 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 [28]. 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 28 suggest that around 10% of the use case (**??**) 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 [28]):

- 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 28 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 [28].

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 [28]. 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 [28]. As is the vision which acts as an executive summary of the whole project [28].

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 [28]

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 27). Ours will at this stage will be smaller. Interestingly Larman 27 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 27).

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 [27]: 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" [47]. 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 [47]. Sommerville 47 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 47 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 [47]: 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 47:

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 [47]

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 [47]. 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. [47].

In principle functional requirements should be both [47]:

- **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 [47]. 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 [47].

Non-Functional Requirements

Non-Functional requirements are those which are not concerned with the specific services provided by the system [47]. They relate to what are known as the emergent properties of the system [47], 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 [47]. 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 [47]. 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 [47]:

- *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 47[p.88]).

A warning echoed by [47] 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 [47]. 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 [47]. The general process model is given by [47], 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 [47]. 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 [27].

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

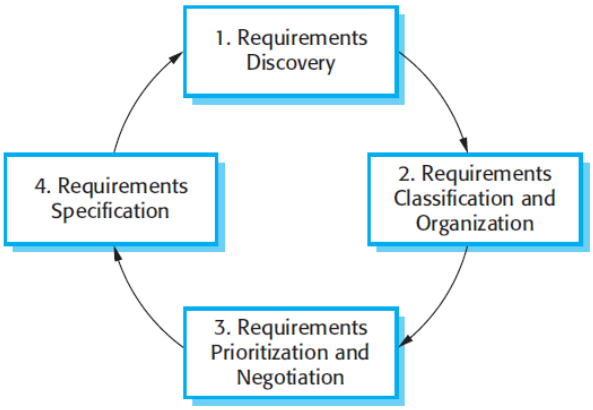

Figure 4.13 The requirements elicitation and analysis process

Figure 4.3: from Sommerville 47: 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 [47]. 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 [47]. 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 [47].

Traditional Techniques I'm going to briefly introduce the techniques here, for a more detailed description see Sommerville 47.

- *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 [47]. There are two distinct types [47]:

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 [47]. 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 [47].

- *Scenarios* - People find working though real-life situations much easier to relate to than abstract descriptions of processes [47]. 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 [47]. 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 [47].

- *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 [47]. We'll be discussing them at lenght next week see **??**.

- *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 [47].

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 [47]. 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 [47].

Sommerville 47 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 47 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 [47]. 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 47 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 47: Ways in which requirements may be documented

the customer really wants (or validation) is an essential one [47]. Several different kinds of check can be undertaken on the requirements document [47]:

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 [47]:

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.

[47] 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 [47]. 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 47 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 [47]. Where the requirements document forms the basis of a formal contract Sommerville 47 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 [47].

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 [47]. 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 29 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 [**?**]. UP welcomes any requirements elicitation method which can add value and that increases user participation, even the simple XP "story card" approach [29]. Larman 29 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+ [29]:

- **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 [29]:

- **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 [29]):

- **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 29 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 coversion. 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 Pattenrs*, chapter Inception is Not the Requirements Phase, pages 47–53. Prentice Hall PTR.

Larman (2014c). *Applying UML and Pattenrs*, 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: . 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 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 [28]. 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 28 suggest that around 10% of the use case (**??**) 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 [28]. 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 [28]. As is the vision which acts as an executive summary of the whole project [28].

Sample Development Case In the development case is shown in 5.1 (from Larman 27) 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 [27]: 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 6) and described by Larman 27 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 [27]. 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 [30]. Larman 30 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 30 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 [30]. The original translation from Swedish is cases of use, poor translation appears to be common in UML [30].

5.3.1 Key Concepts

There are a number of key concepts, these are introduced by Larman 30 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 [30]. This can also include the system under discussion (SuD) itself when it calls upon the services of other systems or subsystems [30]. There are three kinds of external actors [30]:

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 [30]. 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 [30].

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 [1]:
"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' [1]. This seems to be an old term which has been replaced

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 [30]. 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 30 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 [30].

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 [30]. Another key strength is the ability to scale up in terms of formality and sophistication [30], see 5.3.5.

5.3.5 Use-Case Notation

Larman 30 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 30:

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 5.3.5).

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 [30]. Larman 30 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.

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 (5.3.5) has some limited guidance, Larman 30 provides more detail.

Examples

Unlike Larman 30 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 (5.3.5).

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 the OMG (2.5) specification document this 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 [1] 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).

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 (5.4.1) 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 [30] 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 25, to be exact [**?**] 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

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 [**?**]. 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 [**?**]. Italics or underlined text are used in the books examples, in many of the use case tools they are implemented as navigable hyper-links.

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) [**?**]. This splitting of large use case is also recommended by **?** **?** and in more planned approaches used to decompose system behaviour so this can be mapped to individual sub-systems [**?**]

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 30 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 [**?**]. The rational for the extends relation is that under

a given condition at some point an additional behaviour *may* be included [?]. ? ? 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.

5.4.3 Use Case are Requirements in Context

5.4.4 Guidelines for Creating Use Case

5.4.5 Use-Case in UP

5.5 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, 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, based on the examples provided begin work on a glossary and vision document.

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?

5.5.1 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.

5.6 Summary

Larman 30 is very clear on the importance of the textural use case components over the diagrams. Other books discussion Object Orientated Analysis (OOA) are much lighter on the documentation and focus on the models e.g. [**?**] and [**?**]. My previous favourite book on OOA [**?**] 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....
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.

5.7 Preparation for Next Week

Alistair Cockburn author of the popular book on use-case modelling xxxx has written an interesting article on the development of use case diagrams.
YOU will need to do the preparatory work ahead of next weeks class!

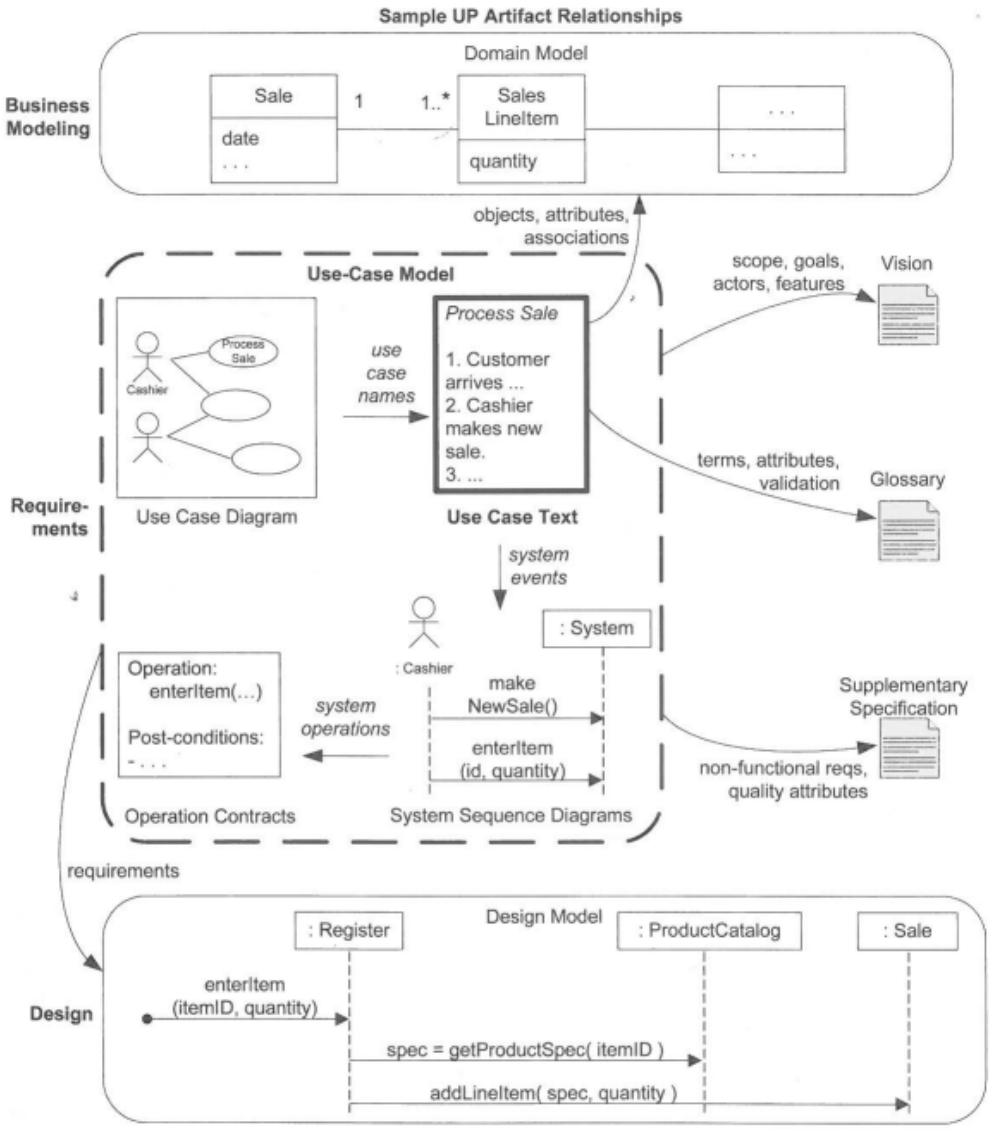

Figure 6.1 Sample UP artifact influence.

Figure 5.2: From Larman [27]: 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 from from Larman 30

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 [30]: 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 [30]: 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 [30]: A FullyDressed Use-Case

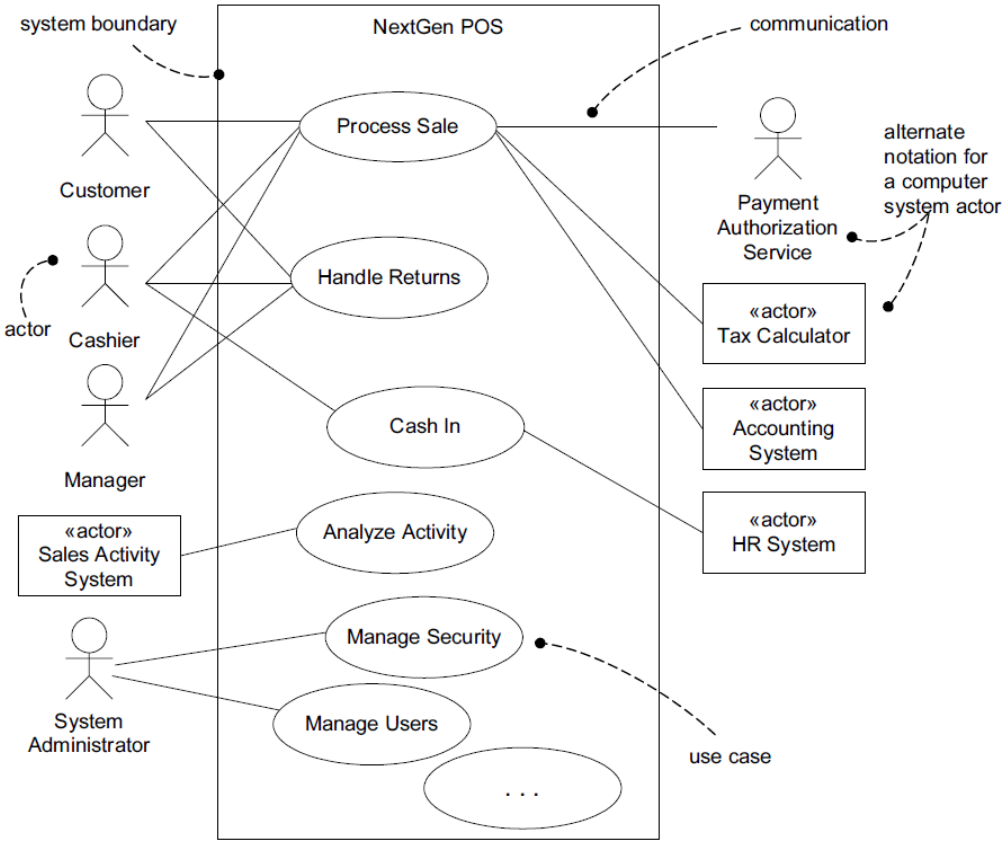

Figure 5.7: from Larman [30]: A partial Use-Case diagram - showing the basic syntax used in high level (context / business model) type use case.

Bibliography

[1] (2015). Unified modelling language: Version 2.5. Available from: http://www.omg.org/spec/UML/2.5/.

[2] 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.

[3] 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)*.

[4] 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.

[6] Ambler, S. (2002a). *Agile Modelling: Evffective Practices for eXtreme programming and the Unified Process*. John Wiley & Sons, Ltd. ISBN: 0-471-20282-7.

[7] 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.

[8] 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.

[9] 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.

[10] Banach, R. (2007). Guest editorial: Formal methods. *Journal of Universal Computer Science*, 13(5):593–606.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Bohem, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5):61–72.

[16] 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`.

[17] 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.

[18] 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*.

[19] 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`.

[20] Folwer, M. Design dead [online]. (2004) [cited 2016.08.23]. Available from: `http://www.martinfowler.com/articles/designDead.html`.

[21] Folwer, M. The new methodology [online]. (2005) [cited 2016.08.23]. Available from: `http://martinfowler.com/articles/newMethodology.html`.

[22] Folwer, M. Semantic diffusion [online]. (2006) [cited 2016.08.23]. Available from: `http://martinfowler.com/bliki/SemanticDiffusion.html`.

[23] Folwer, M. The flacid scrum [online]. (2009) [cited 2016.08.23]. Available from: `http://martinfowler.com/bliki/FlaccidScrum.html`.

[24] 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.

[25] Hall, A. (2007). Guest editorial: Formal methods. *Journal of Universal Computer Science*, 13(5):669–678.

[26] 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.

[27] Larman (2014a). *Applying UML and Pattenrs*, chapter Iterative, Evolutionary, and Agile, pages 17–41. Prentice Hall PTR.

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

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

[30] Larman (2014d). *Applying UML and Pattenrs*, chapter Use Cases, pages 61–101. Prentice Hall PTR.

[31] Leveson, N. G. (1995). *Safeware: System Safety and Computers*, chapter Computers and Risk, pages 21–38. Addison Wesley.

[32] 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`.

[33] 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`.

[34] Moore, J. H. (2008). *Information Technology and Moral Philiospy*, chapter Why we Need Better Eithics for Emerging Technologies, page 26. Cambridge University Press.

[35] Naur, P. and Randell, B. (1968). Software engineering: Report on a conference sponsored by the nato science commitiee. Technical report, NATO, Garmisch, Germany.

[36] Royce, W. W. (1970). Managing the development of large software systems. In *proceedings of IEEE WESCON*, volume 26, pages 328–338. Los Angeles.

[37] 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.

[38] Schach, S. R. (2007a). *Object Orientated Software Engineering*, chapter The Scope of Object-Orientated Software Engineering, pages 3–31. McGraw-Hill, New York, USA.

[39] Schach, S. R. (2007b). *Object Orientated Software Engineering*, chapter Software Life-Cycle Models, pages 32–67. McGraw-Hill, New York, USA.

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

[41] Silberschatz, A., Galvin, P. B., and Gagne, G. (2008). *Operating System Concepts*, chapter Threads, pages 127–152. Wiley Publishing, 8th edition.

[42] 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.

[43] Sommerville, I. Software engineering history [online]. (2010). Available from: `https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/History/`.

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

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

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

[47] Sommerville, I. (2011d). *Software Engineering*, chapter Requirements Engineering, pages 82–117. Addison Wesley, 9th edition.

[48] Storey, N. (1996). *Safety-Critical Computer Systems*, chapter Formal Methods, pages 271–309. Addison-Wesley, Harlow, England.

[49] Van Vliet, H. (2007a). *Software engineering: principles and practice*, chapter Introduction, pages 5–32. Wiley New York.

[50] Van Vliet, H. (2007b). *Software engineering: principles and practice*. Wiley New York.

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

[52] Van Vliet, H. (2007d). *Software engineering: principles and practice*, chapter People Managmenet and Team Organisation, pages 89–107. Wiley New York.

[53] Westland, J. C. (2002). The cost of errors in software development: Evidence from industry. *The Journal of Systems and Software*, 62:1–9.