Introduction to Programming: GRASP Patterns
2016-2017

Version: 0.23

Glenn L. Jenkins

Print Date: March 7, 2017

©2017 Dr. Glenn L. Jenkins

Contents

0.1 Foreword 4

1 The Development of the Object Model - Part 1 5

1.1 Introduction 5
1.2 Domain Models 6
1.2.1 Domain Models and Decomposition 7
1.2.2 Conceptual Classes 7
1.2.3 Examples 7
1.2.4 Lowering the Representational Gap 8
1.3 Domain Diagram Creation 8
1.3.1 How to find Conceptual Classes 9
1.3.2 Noun Phrase Identification 11
1.3.3 Class Diagrams at Different Levels 14
1.4 Domain Diagram Syntax 15
1.4.1 Basic Class Diagrams 15
1.5 Domain Modelling Guidelines 21
1.5.1 Think Like a Map Maker 21
1.5.2 Concept or Attribute 21
1.6 Tutorial 22
1.6.1 Case Study 1 22
1.7 Summary 23
1.8 Preparation for Next Week 23
1.8.1 Essential Reading 24
1.8.2 Background Reading 24

2 The Development of the Object Model - Part 2 25

2.1 Introduction 25
2.2 Orientation - from last week 25
2.3 Associations 25
2.3.1 Identifying Associations 29
2.4 Attributes 31
2.4.1 What attributes should be shown? 32

2.4.2 Identifying Attributes 35
2.5 Is the domain diagram correct? 35
2.5.1 The Role of Domain Models in UP 36
2.6 Tutorial 36
2.6.1 Case Study 1 36
2.7 Summary 37
2.8 Preparation for Next Week 38
2.8.1 Essential Reading 38
2.8.2 Background Reading 38

3 Responsibility Driven Design and GRASP - Part 1 39
3.1 Introduction 39
3.2 Responsibilities and Responsibility Driven Design 39
3.2.1 Aside: Class Responsibility Communication Cards . . . 40
3.2.2 RRD 40
3.2.3 GRASP - Design Principles (mini-patterns) 42
3.2.4 Example - Solving A Simple Design Problem using a GRASP Pattern 43
3.3 Class Diagrams 49
3.3.1 Associations Between Classes 50
3.3.2 Specialised Associations 56
3.4 Sequence Diagrams 57
3.4.1 Participants and Messages 57
3.4.2 Frame Operators 64
3.5 Tutorial 68
3.5.1 Case Study 1 69
3.5.2 Case Study 2 73
3.6 Summary 74
3.7 Preparation for Next Week 74
3.7.1 Essential Reading 74
3.7.2 Background Reading 75

4 Responsibility Driven Design and GRASP - Part 2 76
4.1 Introduction 76
4.2 Recap: Responsibilities and Responsibility Driven Design (RRD) 76
4.3 Responsibilities and Responsibility Driven Design (RRD) . . . 77
4.3.1 Recap: What is RRD? 77
4.3.2 Solving A Simple Design Problems using a GRASP Patterns - Continued 78
4.4 More Class Diagrams 98
4.4.1 Dependency 98

4.5 More Sequence Diagrams 99
4.5.1 Collections and Return Variables 99
4.5.2 Messages to Self 100
4.6 Tutorial 101
4.6.1 Case Study 1 101
4.6.2 Case Study 2 106
4.6.3 Domain Object Diagram for Case Study 2 107
4.6.4 System Sequence Diagram for Case Study 2 107
4.7 Summary 108
4.8 Preparation for Next Week 109
4.8.1 Essential Reading 109
4.8.2 Background Reading 109
References 110

0.1 Foreword

These are fairly advanced topics, I wanted to show you how we get a set of objects (such as those I've given you in the coursework,) and how we work out how they will perform operations together. I'm almost certain the content of these notes is 'too much' but I'm hoping that the key concepts:

- Identify objects you need
- Apply patterns to work out how they're going to before the required system behaviour
- Evaluate the design with useful metrics and apply good practice

Will be enough to get you going with developing Object Orientated software as you go into year two.

The Development of the Object Model - Part 1

1.1 Introduction

During the initial phases of software developed we dedicate a substantial amount of time to identifying the needs of the user, i.e. the requirements of the system. In the Unified Process (or UP) this forms part of what is called the Requirements discipline. This generates artefacts (documents, models etc.) called Use-case, Vision, Supplementary Specification , Glossary, Business Rules (Domain Rules). I've selected the UP as an example as its an object oriented software developed methodology and includes two key actives relevant to our current study.

Business Modelling

Firstly the development of the Domain Model (an artefact from the Business Modelling discipline), in lay terms this is an attempt to work out what are the key objects and their conceptual relationships in the problem space. These will form the basis of our software objects later. Larman 7 states that the object domain model is the most important object orientated model in analysis (noting that use-case, while useful, are not object based). The model illustrates useful concepts within the domain, i.e. things of interest which will need to be represented in software [7]. It also forms the basis of our design models here we attempt to lower the conceptual gap between our problem space and the solution by using similar names e.g. we might represent a Sale with a class called Sale.

In terms of learning this the diagrams (more UML,) see at first to be the most difficult part however the syntax is simple, consistent and very well documented. The most difficult part is using the diagrams correctly to elicit a representative domain model.

Design

The second key activity is the development of the Design Model. The design model describes the logical design , including Software class diagrams, object interaction diagrams package diagrams etc.[8].

1.2 Domain Models

The Domain Model is used to show noteworthy concepts, bounded by the use-case (scenario etc.,) of the current iteration [7]. UP is an iterative method, we target one operation of the system at a time. Essentially we're looking to identify things (though not always physical things,) we're interested in modelling. For example in a hotel we might include in our model, Room, Guest and Reservation only two of which are physical objects in our problem space.

Larman 7 notes that there are a number of synonyms including conceptual models, domain object models and analysis object models. A closely related terms is that of domain layer (also sometimes called the domain model,) which describes the classes below the user interface which have responsibilities (i.e. contain the logic for the application) [7].

The Domain Model allows us to visualise and relate concepts in the domain while abstracting key details [7]. For this we use the UML class diagram notation, however we're careful to use only a subset of it omitting the method signatures from each class. The Domain Model provides a conceptual perspective and shows [7]:

- domain objects or conceptual classes
- associations between conceptual classes, e.g. Sale relates to a number of items.
- attributes of conceptual classes e.g. each Sale has a Date and Time.

The Domain Model is not a design diagram, it should have nothing which relates to implementation (e.g. data types, collections,) or to domain layer classes with responsibilities (i.e. methods) [7].

1.2.1 Domain Models and Decomposition

As with many software development (and more generally engineering) methods object orientation promotes divide an conquer [7]. Unlike traditional structured analysis which worked based on decomposition into processes or functions (think back to semester one Introduction to Programming) [7]. Object orientated development relies on the identification and modelling of entities, things (objects,) in the domain which though their interaction (working together,) perform the required functionality of the system [7]. In this respect the identification of suitable objects from the domain is a key part of the object orientated approach and its importance cannot be overlooked.

1.2.2 Conceptual Classes

The Domain Model illustrates conceptual classes or vocabulary, these are essentially ideas, things or objects [7]. More formally they can be considered in terms of:

- **Symbol** - A chosen symbol for a concept, e.g. the English word *Sale*.
- **Intention** - The intention of the symbol, e.g. to represent a purchase transaction has a data and time.
- **Extension** - All the examples of sales, e.g. the set of all sales.

1.2.3 Examples

The example from Larman 7 is show in 1.1. See 1.4 for an explanation of the syntax.

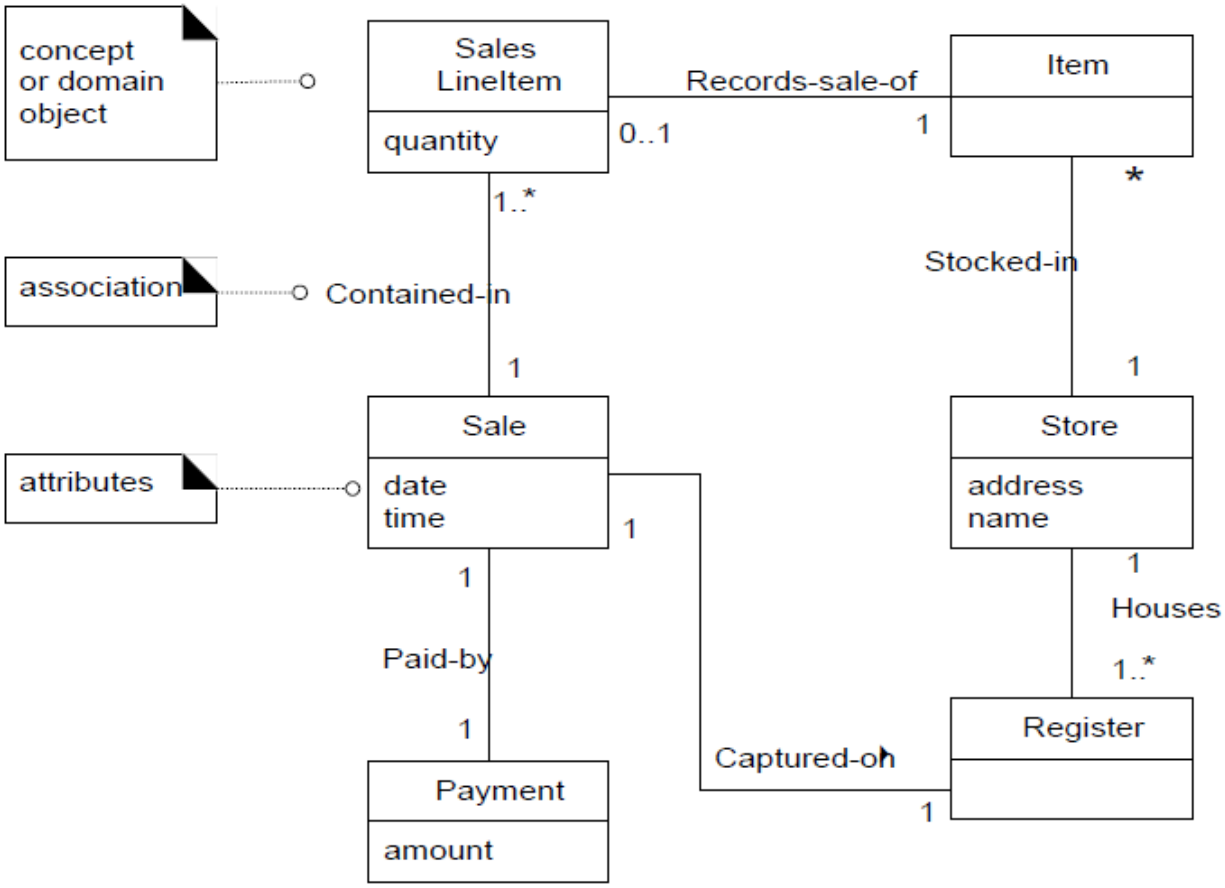


Figure 1.1: An example domain diagram from Larman 7

1.2.4 Lowering the Representational Gap

The representational gap is that between the software model of the system and our mental model of the problem space [7]. One of the advantages of object orientation over other techniques is our ability to create objects in the domain layer (below the interface layer in the software,) to represent those in the problem space [7]. Reducing the Representational Gap makes the software easier to understand and modify [7].

1.3 Domain Diagram Creation

Larman 7 provides some guidance for the creation of the Domain Model. He condenses this to three steps *Bounded by* (i.e. assuming the limitations of the current use-case/scenario etc.):

1. Find the conceptual classes (see 1.3.1).
2. Draw them as classes in a UML class diagram (see 1.4.
3. Add associations (see 2.3) and attributes (see 2.4).

1.3.1 How to find Conceptual Classes

Identifying conceptual classes is often straight forward,identifying them all and not ending up with some attributes as classes etc. is more complicated. Larman 7 suggests the following methods:

1. Reuse or Modify Existing Models.
2. Use a Category List.
3. Identify Noun Phrases.

Use A Category List

Category lists are as the name suggests common categories of candidate class [7], perhaps the easiest to consider is "Phyiscal Objects". The categories provided by Larman 7 are focused towards information systems (see 1.2).

Conceptual Class Category	Examples
physical or tangible objects	*Register* *Airplane*
specifications, designs, or descriptions of things	*ProductSpecification* *FlightDescription*
places	*Store* *Airport*
transactions	*Sale, Payment* *Reservation*
transaction line items	*SalesLineItem*
roles of people	*Cashier* *Pilot*
containers of other things	*Store, Bin* *Airplane*
things in a container	*Item* *Passenger*
other computer or electro-mechanical systems external to the system	*CreditPaymentAuthorizationSystem* *AirTrafficControl*
abstract noun concepts	*Hunger* *Acrophobia*
organizations	*SalesDepartment* *ObjectAirline*
events	*Sale, Payment, Meeting* *Flight, Crash, Landing*
processes (often *not* represented as a concept, but may be)	*SellingAProduct* *BookingASeat*
rules and policies	*RefundPolicy* *CancellationPolicy*
catalogs	*ProductCatalog* *PartsCatalog*

records of finance, work, contracts, legal matters	*Receipt, Ledger, EmploymentContract MaintenanceLog*
financial instruments and services	*LineOfCredit Stock*
manuals, documents, reference papers, books	*DailyPriceChangeList RepairManual*

Figure 1.2: Based on Larman 7: Categories to aid identification of conceptual classes.

1.3.2 Noun Phrase Identification

Linguistic analysis of domain documentation or descriptions of domain operations can be used to identify candidate classes [7]. Blindly mapping nouns to classes is however impossible as there will be some duplication, alternative names for objects in different roles etc., however this is an excellent source of inspiration [7]. The fully dressed use-case(s) provide a lot of descriptive text related to the domain and provide an excellent point at which to start this technique, Larman 7 provides a marked up example see 1.3.

Main Success Scenario (or Basic Flow):

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

Cashier repeats steps 2-3 until indicates done.

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

Extensions (or Alternative Flows):

7a. Paying by cash:

1. Cashier enters the cash **amount tendered.**
2. System presents the **balance due,** and releases the **cash drawer.**
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

Figure 1.3: Based on Larman 7: Example of noun phrase identification for objects from a fully dressed use-case.

Other Techniques

Douglass 1 provides the following useful techniques for object identification see 1.4

Table 6-1: *Object Discovery Strategies*

Strategy	Description
Underline the noun	Used to gain a first-cut object list, the analyst underlines each noun or noun phrase in the problem statement and evaluates it as a potential object.
Identify causal objects	Identify the sources of actions, events, and messages; includes the coordinators of actions.
Identify services (passive contributors)	Identify the targets of actions, events, and messages as well as entities that passively provide services when requested.
Identify messages and information flow	Messages must have an object that sends them and an object that receives them as well as, possibly, another object that processes the information contained in the messages.
Identify real-world items	Real-world items are entities that exist in the real world, but that are not necessarily electronic devices. Examples include respiratory gases, air pressures, forces, anatomical organs, and chemicals.
Identify physical devices	Physical devices include the sensors and actuators provided by the system as well as the electronic devices they monitor or control. In the internal architecture, they are processors or ancillary electronic widgets.
Identify key concepts	Key concepts may be modeled as objects. Bank accounts exist only conceptually, but are important objects in a banking domain. Frequency bins for an on-line autocorrelator may also be objects.
Identify transactions	Transactions are finite instances of associations between objects that persist for some significant period of time. Examples include bus messages and queued data.

Table 6-1: *(cont.)*

Strategy	Description
Identify persistent information	Information that must persist for significant periods of time may be objects or attributes. This persistence may extend beyond the power cycling of the device.
Identify visual elements	User interface elements that display data are objects within the user interface domain such as windows, buttons, scroll bars, menus, histograms, waveforms, icons, bitmaps, and fonts.
Identify control elements	Control elements are objects that provide the interface for the user (or some external device) to control system behavior.
Apply scenarios	Walk through scenarios using the identified objects. Missing objects will become apparent when required actions cannot be achieved with existing objects.

Figure 1.4: From Douglass 1: A selection of object discovery strategies.

1.3.3 Class Diagrams at Different Levels

Both Larman 7 and Fowler 3 describe UML being used at different levels, this relates back to our assertion at the start of the section that UML is a language used for a number of purposes. Larman 7 identifies:

- **Essential or Conceptual Perspective** – diagrams are interpreted as representing real world entities and their relations in the problem domain (this is where our Domain Diagram fits). See 1.6.
- **Specification Perspective** – diagrams are interpreted as representing software elements, including classes interfaces, packages etc. but with no reference to the features of the implementation language. We note that the notation used is the same as that for the conceptual perspective, though clearly extended a little.
- **Implementation Perspective** – diagrams are interpreted as representing software elements in a particular language i.e. including the specifics of the language chosen.

1.4 Domain Diagram Syntax

This section begins to introduce the UML (Unified Modelling Language).

It is vitally important to note that the UML is just a language and it is important to distinguish between what Larman 7 describes as raw UML with its applications within methodology defined diagrams. The Domain Diagram uses the class diagram notation, as will the Design Model we produce later in this phase. UML is a language, we use the same words and punctuation to write a report as we do to write a story or a joke. The bad news I suppose is you're going to have to learn UML while learning to use it.

Note: There is a major difficulty here, either I introduce the techniques to identify conceptual classes (classes in the domain model) and then how to represent them (which I have done,) or as I'm going to do with associations show you how they are represented then how they are found.

1.4.1 Basic Class Diagrams

Fowler 4 states that the class diagram describes the kind of objects in a system and the static relationships between them. They show the properties and operations (the UML term is **features**,) of the class along with any constraints on its creation [4]. Fowler 4 expresses all of these in a single example 1.5, I'll add a little detail but to be honest there are so many excellent and detailed books on the UML finding information should not be a problem.

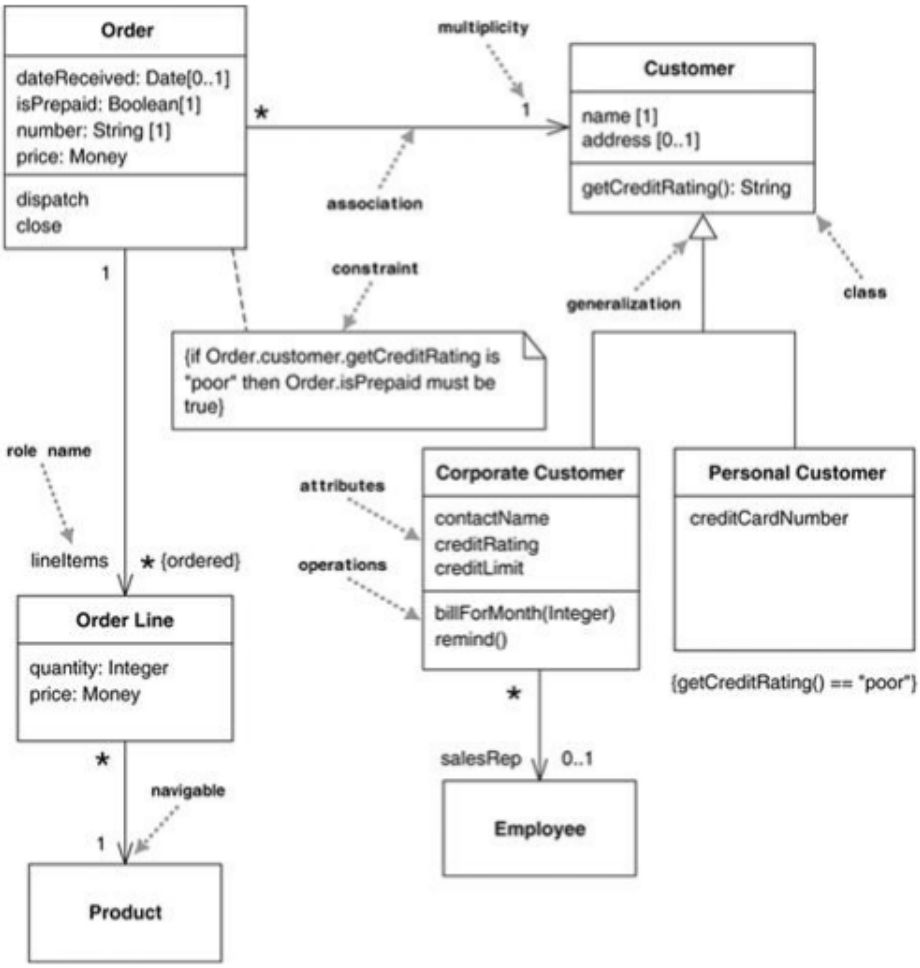


Figure 1.5: Based on Fowler 4: Class diagram example. Sorry about the quality I'm working from home and only have access to the eBook version.

Representing Classes

The class is represented by a box subdivided into three sections, the top section contains the class name, the middle section its attributes or more broadly the properties of the class while the lower section is reserved for the behaviours or operations of the class. The syntax of the diagrams changes depended how they are being used (see 1.3.3). An example of conceptual classes (as discussed in the sections above is shown in 1.6.

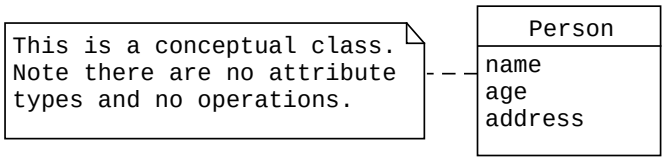


Figure 1.6: A Conceptual Class, of the kind that would be used to represent 'ideas' in the problem domain. This is the first of my own figures, not the best example to be honest. I need to come up with something better.

The UML notation provides the facility to include much more detail in a class diagram, the design and implementation diagrams use more of the syntax see 1.7 and 1.8. Fowler 4 provides a detailed discussion of these, there are also many other sources. The design class shown in 1.7 shows as much syntax as we'll need for now.

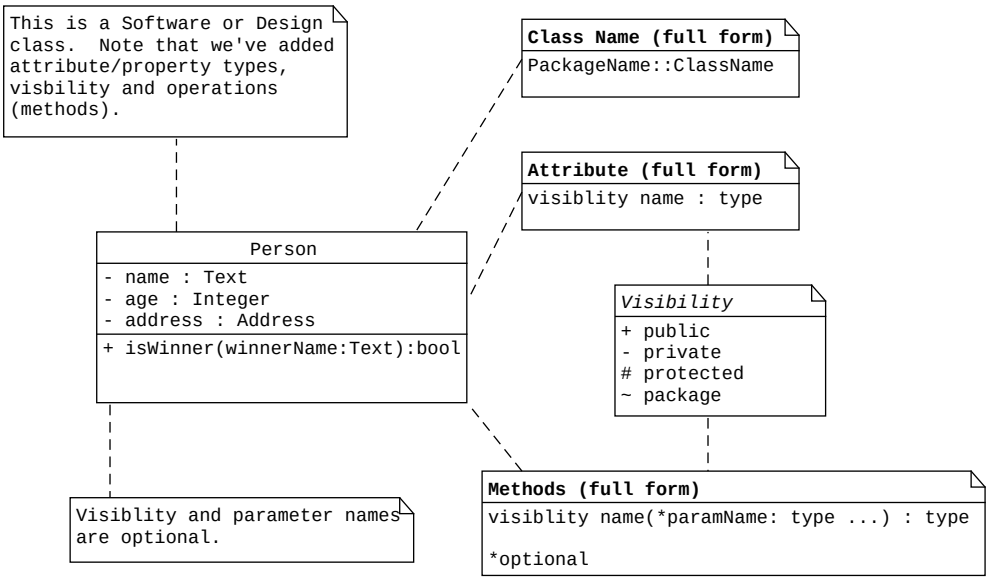


Figure 1.7: A Design Class, this is probably the more common form of class diagram and uses more of the defined 'language'. Again the example is not the most representative.

The Implementation class shows a much more information, including the get and set methods, constructors etc. which were omitted in the Design class. These are implicit (the variables are private and will need to be accessed,)

but are only shown if there is something interesting about the get/set. This very verbose format would only be used rare circumstances in my opinion. I've used this format when specifying classes for students to implement and this format may be generated from code by a tool.

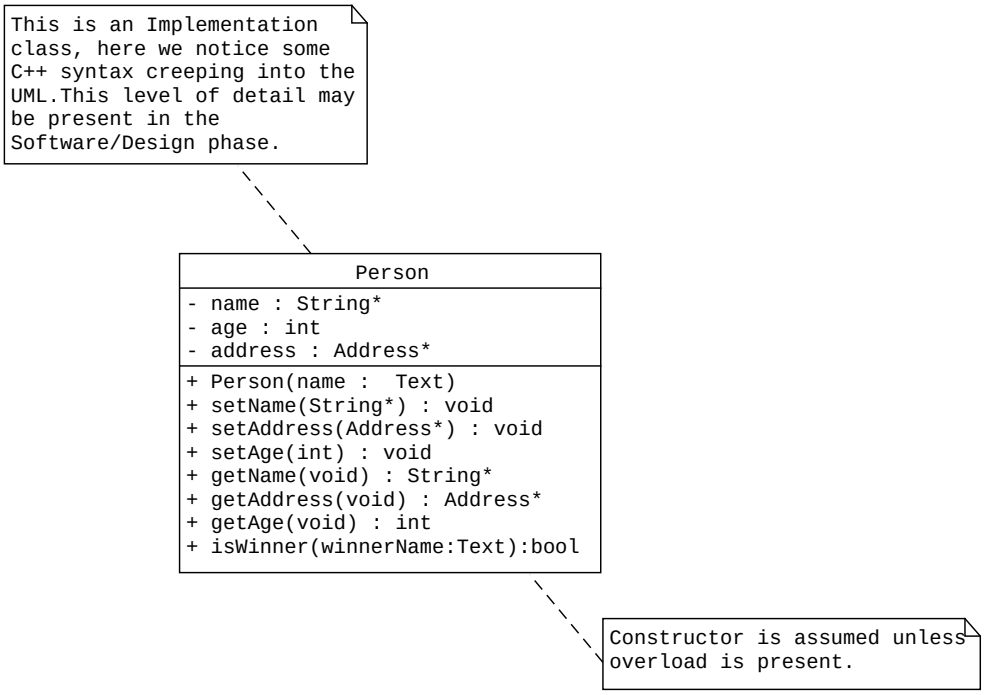


Figure 1.8: An Implementation Class, this is the kind of diagram you'd see developed for implementation by a junior/student or as an output from too which reverse engineered code into a class diagram.

Some of you may by now have spotted the issue with our example, which I have allowed to cascade though the diagrams.

Representing Associations

Associations are relationships between instances of classes indicating that some meaningful and interested connection exists between them [7]. We'll consider associations and attributes in more detail next week, however as object diagrams without them are often incorrect we need to know a bit about them up front.
According to Larman 7 associations should be shown when one object needs

to show that knowledge of a relationship needs to be preserved for some length of time (milliseconds or years). At this stage (object modelling,) these will not always translate into relationships between classes in software [7]. Larman 7 suggests the following guidelines for the domain model, show only associations for which knowledge of the relationship is preserved for some duration and those derived from the Common Associations List.
We're not going to talk about this list in any detail until next week.
Representing associations is reasonably simple, we use a line with an association name [7] (see 1.9). At the end of each line we have a number or range representing multiplicity (number of) for instances of classes [7]. The association is inherently bidirectional and can be logically traversed in either direction, the association may have a reading direction arrow which suggests how the association name should be read - this has no meaning in the model just helps the reader [7].

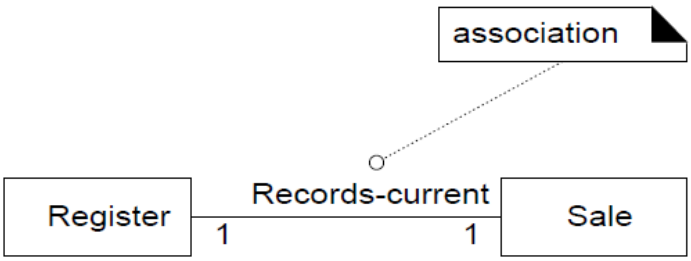


Figure 1.9: From Larman 7, a simple association with unit multiplicity.

In later sections we'll consider these relations in terms of design and include restrictions on navigation, multiplicity, names for association end points (roles) within the classes. However these are all for design models rather than domain models!

We can however correct our earlier diagrams our conceptual class 1.6 had a concept as an attribute (see 1.5.2). I've fixed this in 1.10. This had not been spotted at the point we moved to the design class and is present here also, this probably wouldn't happen in the real world. There are instances however where you suddenly realise that meta-data has crept into a class, i.e something you though was an attribute has turned out to be a variable. For Example, you have a sensor which gets a single temperature reading but you later realise that it has to be calibrated and add a calibration constant. This should now be modelled as a separate class.

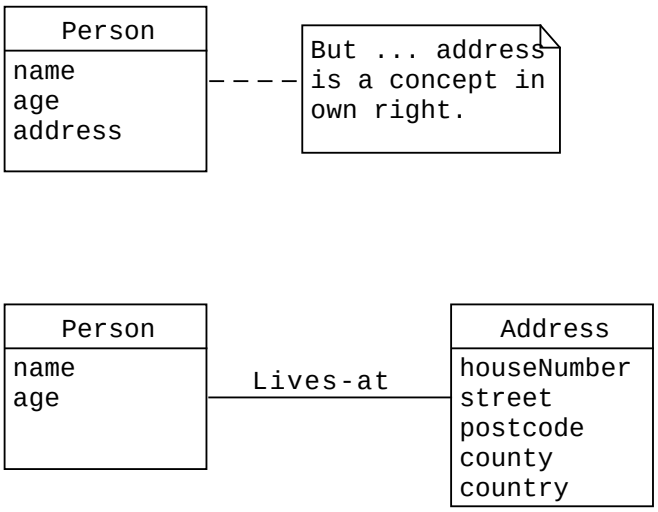


Figure 1.10: A Conceptual Class, of the kind that would be used to represent 'ideas' in the problem domain. This is a fixed version of 1.6 which separates out the x concept.

Classes and Class Diagrams in UP

UP defines several models which may use the class diagram, so far we've looked at the domain model [7]. Each model imposes its own terminology on the boxes (classes) of the model, for example in the Domain Diagram these represent domain concepts or conceptual classes [7]. To simplify things Larman 7 defines up front the terminology he's using (which is both UP and UML friendly,) and which we'll use in these notes.

Name	UP Model	Perspective	Description
Conceptual Class	Domain Model	Conceptual	A entity, concept or thing.
Software Class	NONE	Specification	A class representing the design or implementation of a given software component irrespective of language or method.
Design Class	Design Model	Specification or Implementation	A class in a design model, in UP this can be either a design model or an implementation model.
Implementation Class	NONE	Implementation	A class written in a given language such as Java or C++.
Class	NONE	NONE	as in the UML something which could represent any of the above depending on context.

1.5 Domain Modelling Guidelines

Larman 7 provides a number of useful pointers and additional guidance for creating domain models, a summary of which is provided here.

1.5.1 Think Like a Map Maker

When cartographers produce maps they use the local place names rather than creating their own, in a similar way when developing domain models we should use the terminology of the domain [7]. This applies to both objects and their attributes and possibly to the names of certain relationships [7]. This can be taken further, cartographers do not add cars, or people to their maps in the same way objects are discarded (or ignored) if they have no relevance in the problem domain [7].

1.5.2 Concept or Attribute

Larman 7 suggests that one of the most common mistakes when modelling is to misidentify a concept (object) as an attribute of another. For example,

Sale may have an attribute called *store* to record the location from which an item was purchased. Larmans golden rule here is **If we do not think of attribute X as text or a number in the real world, then its probably a conceptual class**. In the example above *store* should be a conceptual class in its own right which has an association with sale. We'll come on to associations in a bit.

1.6 Tutorial

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

1. Read the case study 1.6.1, then in teams of two produce:
    - An object model.

1.6.1 Case Study 1

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

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

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

Scenario – Sunny Day 2

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

Scenario – Formatting error scenario

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

1.7 Summary

In summary the process of developing an object orientated system requires that we model objects within the problem space. Using the same names for classes as conceptual entities helps us lower the representational barrier aiding our understanding of the software.

There are a large number of suggested techniques, however experience is the key to identifying which of these are strong candidates for objects in our domain.

Next week we'll continue to develop the object model looking at the properties of the classes (attributes and associations).

1.8 Preparation for Next Week

As your first years I'm only going to ask you to **try** and read though next weeks notes before the class.

1.8.1 Essential Reading

Larman 7 Provides an overview of what we've done so far and introduces the next topic.

Larman (2014a). *Applying UML and Patterns*, chapter Domain Models, pages 131–172. Prentice Hall PTR.

NOTE: Those of you who have the older version of Larman (before 3rd edition,) will find the order of the chapters has changed. The large 9th chapter is split over three smaller chapters in this book and proceeded by an unrelated chapter on System Sequence Diagrams (which we'll come back to later). So if you have that version, just read Chapter 10.

We're hopefully going to finish object models, next week and then move on to System Sequence Diagrams.

Larman (2014c). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 173–181. Prentice Hall PTR.

1.8.2 Background Reading

The following covers the development of object models really well:

Douglass, B. P. (2004a). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Anlysis of Real-time Systems, pages 331–399. Object technology. Addison Wesley, 3rd edition.

The same book has some really good sections on use case modelling, I'm slightly nervous about recommending it as I know how easy it is to get a copy.

The Development of the Object Model - Part 2

2.1 Introduction

In the last class we started looking at the creation of a Domain Model. This is is used to show noteworthy concepts, bounded by the use-case (scenario etc.,) of the current iteration [7]. It is used to identify things (though not always physical things,) we're interested in modelling. For example in a hotel we might include in our model, Room, Guest and Reservation only two of which are physical objects in our problem space.

2.2 Orientation - from last week

Last week we started looking at object domain diagrams, but only got as far as identifying the objects 1.3. This week we're moving on to consider the various methods to identify their associations and attributes.

2.3 Associations

It is useful to show the associations required for this iteration, to satisfy the information requirements of the current scenarios (use case) and which aid understanding of the domain [7].

An association is defined as the "semantic relationship between two or more classifiers that involve connections amongst their instances" [7] in this instance the class is our classifier so we're talking about relations between individual objects. Hopefully we covered this last year in the programming module and you've all implemented associations, you may have already as-

sociated one class with another in modules running alongside this one this year. The example shown in 2.1 provides some of the syntax, this is not all the Raw UML associated with Association (I'll cover this when we start looking at design diagrams).

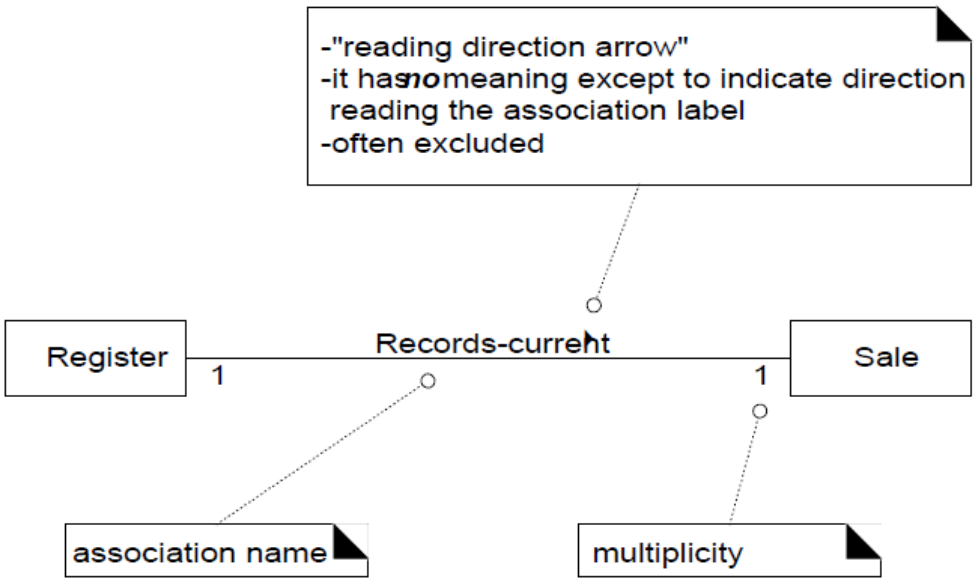


Figure 2.1: An example assocation from Larman 7

What Associations should be included?

Larman 7 states that only those relations which are preserved for some duration. This may be anything from milliseconds to years depending on the context.

Deciding exactly which ones we show and which are visual noise, is difficult and Larman 7 suggests:

- Associations for which knowledge of the relationship needs to be preserved for some duration (i.e. "need-to-know" associations).
- Associations derived from the Common Association List.

Syntax: Associations between Conceptual Classes

While we need some of the class diagram syntax for association we don't need all of it, I'm going to leave some of it until we actually need it (we'll focus on

the raw UML we need for object domain diagrams for now). The diagram 2.2 shows a connection between two objects, this has a directed association name (see 2.3) and multiplicity at each end (see 2.3).

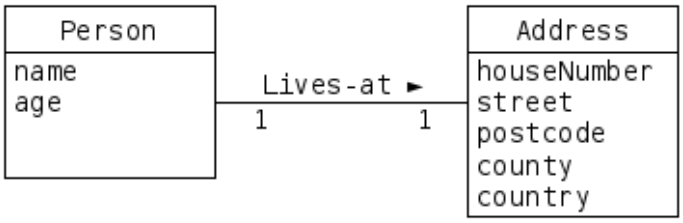


Figure 2.2: A home made example based on those in Larman 7, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image. I've had to screen shot this one to get it to work.

Naming Associations

Simple names such as "has" and "uses" are a poor choice as they do not enhance our understanding of the domain [7]. In the example 2.2 "has" could have been chosen at as the association name but would have added nothing to our understanding, however "Lives-at" tells us its their home address. Larman 7 suggests using a *ClassName-**VerbPhrase**-ClassName* format which is both informative and can be read. While his examples use a hyphenated format, camel case is also acceptable.

Roles & Multiplicity

The end of each association has a role and can have a name, multiplicity expression and navigability [7]. We're going to look at most of these later in the course when we look at design diagrams, however multiplicity is useful now.

Multiplicity :

Multiplicity simply defines how many of each class are involved in an association at one time [7]. That is how many instances can be connected, we can improve / extend our earlier example (person-address) by considering that more than one person can live at the same address. It may be that a person lives at a number of addresses over the course of their lives, but for

this system they have only one current address. The examples shown in 2.3 are reasonably representative.

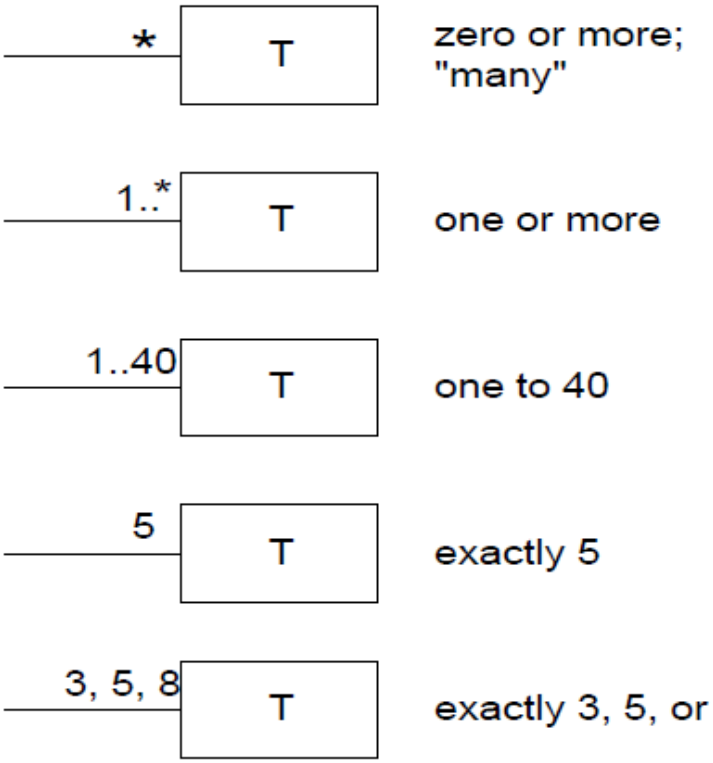


Figure 2.3: A home made example based on those in Larman 7, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image. I've had to screen shot this one to get it to work.

Larman 7 notes that there is some disconnect between the use of associations in domain diagrams and in design diagrams here. While in the real world we may not record the absence of something we may wish to do so in the software case to ensure we know that it should be there and its absence is an error.

The multiplicity of a relationship is always based on the needs of the scenario, [7] cites Rumbaugh's example where a Person is associated with a Company. For tax purposes this will probably be a list of companies (i.e. all those who the person currently works for) for a Union probably only one.

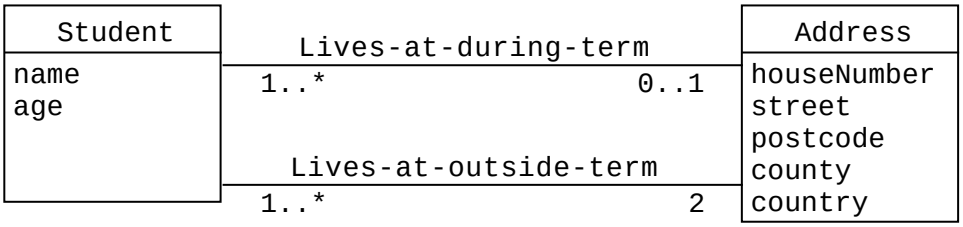


Figure 2.4: A home made example based on those in Larman 7, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image, I've not screen-shot this and its lost its arrows.

A final note is that we can have multiple many to many relations [7], this is perhaps best illustrated though an example see 2.4. Here each student has two addresses a term time address (which may or may not be present,) and a home address. Both of these are address objects, note that one or more students can live at the address in each case.

2.3.1 Identifying Associations

Larman 7 recommends a category list to identify potential associations this is shown in 2.5.

Category	Examples
A is a transaction related to another transaction B	*CashPayment—Sale* *Cancellation—Reservation*
A is a line item of a transaction B	*SalesLineItem—Sale*
A is a product or service for a transaction (or line item) B	*Item—SalesLineItem (or Sale)* *Flight—Reservation*
A is a role related to a transaction B	*Customer—Payment* *Passenger—Ticket*
A is a physical or logical part of B	*Drawer—Register* *Square—Board* *Seat—Airplane*

Category	Examples
A is physically or logically contained in/on B	*Register—Store, Item—Shelf* *Square—Board* *Passenger—Airplane*
A is a description for B	*ProductDescription—Item* *FlightDescription—Flight*
A is known/logged/recorded/reported/ captured in B	*Sale—Register* *Piece—Square* *Reservation—FlightManifest*
A is a member of B	*Cashier—Store* *Player—MonopolyGame* *Pilot—Airline*
A is an organizational subunit of B	*Department—Store* *Maintenance—Airline*
A uses or manages or owns B	*Cashier—Register* *Player—Piece* *Pilot—Airplane*
A is next to B	*SalesLineItem—SalesLineItem* *Square—Square* *City—City*

Figure 2.5: A table from Larman 7, this shows the common associations list.

For those who need further explanation/examples please see:

Larman (2014a). *Applying UML and Patterns*, chapter Domain Models, pages 131–172. Prentice Hall PTR.

2.4 Attributes

It is useful to identify attributes of conceptual classes where these are required to satisfy the information requirements of the current scenario [7]. Attributes are defined as "a logical data value of an object" [7].

2.4.1 What attributes should be shown?

Larman 7 suggests that we should only show those attributes which are suggested by the requirements artefacts (use-cases scenarios etc.) i.e. where we need to remember information.

Looking back at our previous examples our selected examples were out of context and the need for the selected variables gets lost. However, if we consider a Sale being stored in a till system we'll need to store the data and time of that sale. If we're storing a student as part of a Student Information System (at UWTSD) we'd need to store the P number along with other details such as full name, title etc.

Attribute Notation

Attributes are shown the in second compartment (from the top) (see diagram 2.6) and may have a type and other information (see 2.7. The full notation is given in 2.6 we note that we can include:

- Derived attributes which are preceded by a /(perhaps the best example being age, which can be calculated based on the data of birth)
- Visibility - which uses the same syntax as methods (i.e. $+, -, \#, \&$).
- Property String - this describes the properties of the data e.g. {Read-Only}.
- Optional values - indicates weather a given value is required 0 or not 1.

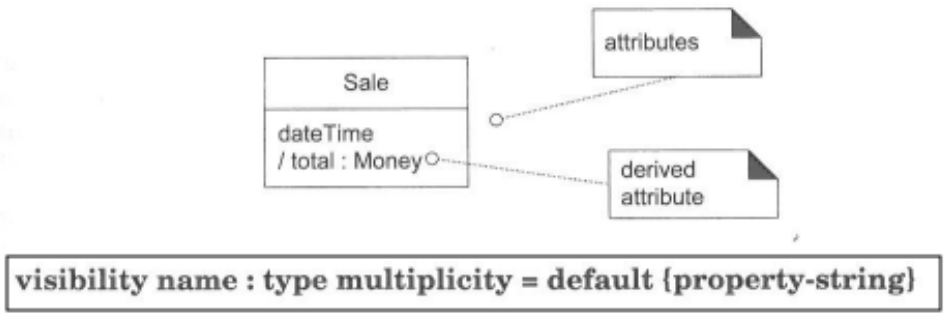


Figure 2.6: An example from Larman 7, this shows some basic syntax (as might be used in an domain object model. The full syntax (raw UML) is given for attributes in the box.

Derived Attributes - Are interesting we use them to communicate that while this is a noteworthy attribute, it is derivable from other data and therefore does not need to be stored [7]. The UML convention is to use a / symbol before the attribute name [7]. Its also useful to note that there are a number of useful attributes which can be derived from the multiplicity of a relation - e.g. we could work out the number of students in a group from the multiplicity without storing its separately [7].

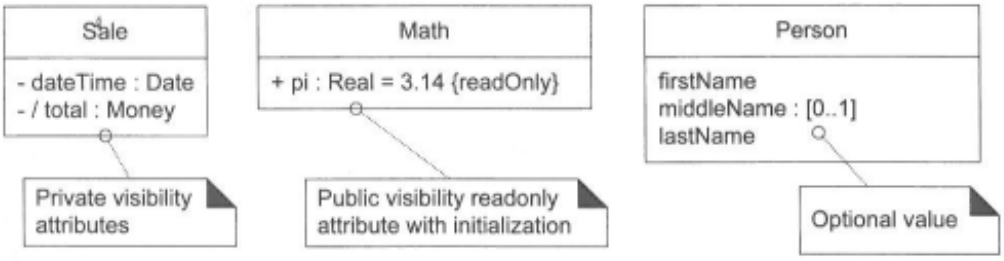


Figure 2.7: A second example from Larman 7, this shows how other information may be captured. For recording purposes Larman suggest including this in the glossary see 2.4.1

Where to Record Attribute Requirements?

Adding the additional information above, e.g. derived attributes, property strings and optional values embeds requirements or domain rules in the domain model [7]. Larman 7 suggests that while this can be useful when creating domain diagrams with domain experts (users etc.) the domain module is not always updated or used as a reference and these should therefore be transferred to the glossary.

What are Suitable Attribute Types?

Attributes should be "primitive types" that is something which is not a concept within our system (e.g. Number, Text, Boolean etc.) [7]. We avoid at this stage considering primitive types from the design/implementation perspective, i.e. int, float etc. Some of these will require an object at design time to represent them (e.g. Text/String, DateTime etc.). While Larman 7 provides quite an exhaustive list, I suggest we restrict ourselves to:

- **Basic Data Types:** Boolean, Number, Character and String (or Text).
- **Common Complex Data Types:** Date, Time and Colour

- **Geometric Primitives:** Point, Rectangle.
- **Enumerations:** e.g. Size=small,regular,large.

Larman 7 goes on to discuss data types in some detail, he concludes that the term "primitive" implies a set of values for which a unique identity is not meaningful. For example, the Integer 5 cannot be distinguished from another Integer 5, nor a string containing "dog" from another string containing "dog". However there may be more than one instances of person, with the same name. In this instance would not compare two entities by instance i.e. x == 5, we would used x.equals(y) that is compare the values of the entity.

Guideline: Defining new Data Type Classes

There are a number of instances where we encounter this issue of non "primitive" data types, often this suggests a conceptual entity which we should consider. Some of these will be very small i.e. have no meta-data but have associated rules governing their use or correctness [7]. For example consider StudentID, we may at first suggest a number (i.e ignore the P in the case of UWTSD Swansea students,) or possibly text however there may be sub-parts within the I.d. e.g. the first year of enrolment may be the first two digits, the department the second two etc.

Larman 7 provides some some common cases for making an attribute a new data type with examples:

- Attribute composed of multiple sections
  - Phone Number
  - Address
  - Full Name
- Attribute has associated operations, such as parsing, validation etc.
  - Student Number
  - Car Registration
  - ISBN
- Attribute is has other attributes (meta-data)
  - Promotional Price - has a start and end date
- Attribute is a quantity with a unit, (see 2.4.1).

  - Price - has a unit of currency e.g. Pounds, Euros, Groats etc.
  - Weight - has a unit of measure e.g. grams, kilograms etc.

- Attribute is an abstraction of one or more types with some of these quantities.
  - Larmans sale item object has an identifier which is a generalisation of specific types Universal Product Code (UPC) and European Article Number (EAN).

Whether we include these as separate objects in our domain diagram or not depends on what we are looking to emphasise in our digram [7]. At this stage (i.e. while your a student trying to show you can model in objects,) I would suggest expressing all data-type objects as separate and making no assumptions.

Guideline: Avoid Relating Objects using Attributes

Attributes should not be used to relate two conceptual classes in the domain model [7]. This is often a case of database driven thinking where **foreign key** type attributes are included [7]. For example we may include a CourseID in a student object, while we may do this at a later stage to enable us to write a student to a database as a record. This is a design decision which should be avoided at this stage [7].

Guideline: Modelling Quantities and Units

Larman 7 suggests that the majority of numeric quantities should not be represented as plain numbers, he suggests that giving a price of 20 or a weight of 100 does not provide enough information. It is common in this case to represent such values with an associated Unit, describing the units used [7]. This may be used to format output and for validation purposes.

2.4.2 Identifying Attributes

Larman 7 provides no guidance for identifying attributes at this stage.

2.5 Is the domain diagram correct?

In short no. But we're not looking for correct, our aim is to create a useful diagram one which we can use as a tool for communication and understanding

[7]. A useful model captures key domain abstractions and the information required to understand the domain, i.e. its concepts, terminology and relationships [7].

2.5.1 The Role of Domain Models in UP

The Domain Model is exploratory and may evolve over a number of iterations during inception, however by the end of which it may well be completed [7]. As we are exploring the system one use-case or scenario at a time we'll expand the domain model for each as required, however by the end of elaboration there may be no 'new' concepts to uncover. Larman 7 stresses here that we should avoid trying expand the model to cover everything, i.e. stick to the scenario/use-case at hand, avoiding 'waterfall' (big model up front) thinking.

2.6 Tutorial

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

1. Read the case study 2.6.1, then in teams of two produce:
    - Refine the object model from last week, adding attributes and associations as required.

2.6.1 Case Study 1

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

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the

conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

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

Scenario – Sunny Day 2

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

Scenario – Formatting error scenario

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

2.7 Summary

In summary the process of developing an object orientated system requires that we model objects within the problem space. Using the same names for classes as conceptual entities helps us lower the representational barrier aiding our understanding of the software. The domain model is an incomplete Business Object Model which focuses on a small part of the business/system concerned with the software project at hand [7].

This week we've looked at the inclusion of relations and attributes in our domain model to capture essential details. This provides us with a first holistic structural view of our system which will be refined in the design model towards something we can implement in software.

Next week we'll look at mapping system behaviour to the objects in the domain model, this will allow us to create a design class diagram.

2.8 Preparation for Next Week

No need to do the background reading, as its only your first year. Please have a read though next weeks notes before the class.

2.8.1 Essential Reading

Larman 7 Provides an overview of what we've done so far and introduces the next topic.

2.8.2 Background Reading

The following covers the development of object models really well:

Douglass, B. P. (2004a). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Anlysis of Real-time Systems, pages 331–399. Object technology. Addison Wesley, 3rd edition.
The same book has some really good sections on use case modelling, I'm slightly nervous about recommending it as I know how easy it is to get a copy.
YOU will need to do the preparatory work ahead of next weeks class!

Responsibility Driven Design and GRASP - Part 1

3.1 Introduction

In the previous notes (2) we'd started object design and were looking to begin mapping the behaviour of the system to these. When we study this in more detail we'll have the analysis artefacts (use case, SSD, etc to guide us,) however, for the simple problems we'll solve between now and then I'm hoping a simple set of patterns will be enough.

In an attempt to systematically design object orientation systems adopted a responsibility driven design approach (RDD). This suggested the use of GRASP patterns in order to assign low level responsibilities and we'd looked at one of these patterns. I'll recap the key concepts before adding more patterns.

3.2 Responsibilities and Responsibility Driven Design

Designing software objects by thinking in terms of responsibilities, roles and collaboration is both popular and well established in Software Engineering [10]. This is part of a larger approach called responsibility-driven design (or RDD,) but may also be familiar from other techniques such as CRC cards 3.2.1. Larman is keen to point out the generality of this approach and we can think of the work we've done with structures and functions in these terms.

3.2.1 Aside: Class Responsibility Communication Cards

Class Responsibility Communication (CRC) cards are a popular text orientated technique created by Kent Beck and Ward Cunningham (founders of the ideas of XP and design patterns) [11]. Larman 11 suggests the primary reason for this may be cognitive styles, however other factors experience, comfort etc. may contribute. Using CRC cards for design is a reasonably social activity (see next paragraph,) again this doesn't suit everyone.

CRC cards are essentially index card (which you may have never seen - see ,) with written on it the name of the class it represents and its responsibilities (what it is responsible for knowing and doing,) and who collaborates with it in each case [11]. The example below is a hand drawn template from Larmans book **??**.

The process of using the cards involves taking the conceptual classes as a starting point sitting around a table working though "what if' scenarios or use case with the cards [11]. This is almost a role playing exercise with individuals playing the part of various classes, oversights and omissions can be easily spotted and this provides a good starting point for responsibility based design. We however are starting lower down with the GRASP patterns, however the CRC technique is useful at a higher level.

3.2.2 RRD

In RRD objects are attributed responsibilities, in terms of what they are responsible for knowing and what they are responsible for doing [10]. The UML defines responsibilities as "A contract or an obligation of an element in its relationship to other elements" [16], this is an abstraction [10].

There are two types of responsibility knowing and doing [10]:

- **Doing** - Responsibilities may include:
  - Doing something itself, e.g. a calculation, creating and object etc.
  - Causing something to happen in another object, or changing its state.
  - Controlling and coordinating activities in other objects (e.g. **??**)
- **Knowing** - Responsibilities may include:
  - Holding data itself in private attributes

- Knowing about related objects
- Knowing about data it can derive or calculate

One key observation (which sort of links us back to CRC cards,) is that objects can collaborate (work with other objects,) while fulfilling the responsibilities in either knowing or doing [10]. Another is that while methods will ultimately full fill the responsibilities of a class, responsibilities are not methods [10], I've tried to adapt some examples from Larmans book to give you an idea.

Class	**Responsibility**	**Collaboration Classes**	**Collaboration Method**
Sale	Responsible for creating *SalesLineItem*s (doing)	None	None
Sale	Responsible for knowing the sale total (knowing)	*SalesLineItem*s	*Sale* is associated with *SalesLineItem*s which are part of the sale. It calculates the total by asking each in turn for their totals (price * quantity). This uses *getSubTotal()* in *SalesLineItem*.

Table 3.1: Examples of objects with responsibilities, appropriate collaborators and collaboration methods (adapted from Larman 10

Guideline: Look for Responsibilities in the Domain Model

Any attributes which are shown in the domain model are things which the equivalent class in the design model is responsible for knowing [10]. For example, if the *Sale* class from Larmans example has an attribute called *time* in the domain model it follows that class it inspires with the same name in the design model (remember our aim to lower the representational gap,) will be responsible for knowing the time of the sale.

Guideline: More than one Responsible class

Small (simple) responsibility may only require a single method (for example "create a sale",) where as more complex responsibilities ("store the sale data

in a relational database") may require the interaction of many more classes, possibly including a subsystem with a responsibility to "provide access to a relational database" [10]. RDD then includes the idea of collaboration, responsibilities are fulfilled either by a method or with other objects methods [10] (see 3.1).
Larman 10 notes that RDD is a general metaphor for thinking about OO software design, objects are like people with different roles working together to achieve some goal. Each is responsible for part of the tasks and and like objects they operate as a community of collaborating responsible individuals.

3.2.3 GRASP - Design Principles (mini-patterns)

GRASP stands for General Responsibility Assignment Software Patterns (technically GRAS Patterns,) with the name being specifically chosen to emphasise the importance of grasping these principles for the development of software [10]. GRASP is a learning aid which names and describes some basic principles to assign responsibilities to objects, proving a simple first step towards RDD [10]. It also provides a foundation for applying more complex patterns such as the Gang of Four (GoF) patterns which are 'larger' and deal with coordinating higher level behaviour. Some of these (or perhaps all,) will not really be new to you, but you won't have seen them named and their pros and cons discussed. Naming them allows them to be explained and applied [10] leading to a more systematic approach to the delegation of responsibilities (and therefore object behaviour).

When do we assign Responsibilities?

Responsibility are assigned either during design when we draw interaction diagrams and decide where the methods which fulfil these responsibilities go or while coding [10]. The GRASP principles can be applied to guide choices about assigning responsibilities and can therefore be applied during design (while drawing interaction diagrams,) or during coding [10].

Patterns

Experienced software developers build up and deploy a "repertoire of both general principles and idiomatic solutions" (that is a generalised bits of solutions,) which if codified in a appropriate format, i.e. named, with a description of the problem and solution give a pattern [10]. Named patterns provide a way chunking and incorporating that concept in to our understanding and memory [10]. That is if we see something described as a pattern (and

named,) we can think of it as a pattern and remember it, perhaps associated with other similar pattern. This also facilitates communication we can talk about using the Abstract Factory Pattern, to solve a problem and other developers instantly know what we're talking about [10].

According to Larman 10 Patterns are usually expressed in terms of a general problem and solution which can be applied in many contexts, there may also be some consideration of the trade-offs to be considered. The pattern will then suggest which responsibilities should be associated with which objects (given their roles in the pattern).

Pattern Name:	**Information Expert**
Problem:	What is the basic principle by which to assign responsibilities to objects.
Solution:	Assign a responsibility to the class that has the information needed to fulfil it.

Table 3.2: An example pattern from Larman 10 using his format.

Clearly the pattern shown in 3.2 is both very general and perhaps obvious, however naming it helps use learn, discuss and deploy [10].

Before looking at some patterns being used in a real scenario we'll note some final points from Larman 12, patterns should not be new ideas (they should be established ones which are being named). That being said I've found a lot of new variations on old patterns, which I suppose is just people taking the time to recognise that there are many versions of something. The GRASP patterns do not state new ideas they name widely used principles, to an expert they should be fundamental and familiar [10]. To you (second year students,) they'll hopefully be thing which you were starting to get to grips with in your object originated coding, or even guidance for problems which are on the horizon in your coding.

3.2.4 Example - Solving A Simple Design Problem using a GRASP Pattern

Rather than start with a full scenario (or use case,) which is where we'd normally start in terms of mechanistic design, I'm going to start with some 'parts' of a larger solution. Seeing these simple patterns deployed in a number of examples will allow us to work up to a full scenario, essentially this will be

the detail behind the call to system from our SSD (see **??**). I've stolen these examples from Larman 10, we'll work though something similar for other examples next year.

Question: Who creates the Square object?

Larman 10 provides a very detailed discussion of this example and the case study it comes from, this is a much abbreviated version and I recommend you read the full one. Larman really explains his thought process well.

A common (and basic,) problem in OO design is who creates a particular object, this is a 'doing' responsibility which must be assigned to some object [10]. The example we'll use is from a scenario/use-case for the first elaboration iteration of a Monopoly game, where the game is initially set-up, a single piece is added to the board and moved around the board by dice roles (without player interaction). **? ?** covers this in some detail but we're only looking at some elements of it anyway.

The question is:

"Who creates the Squares which represent the Monopoly board?"

First lets look back at the object domain model to see what classes we have in addition to square (see 3.1).

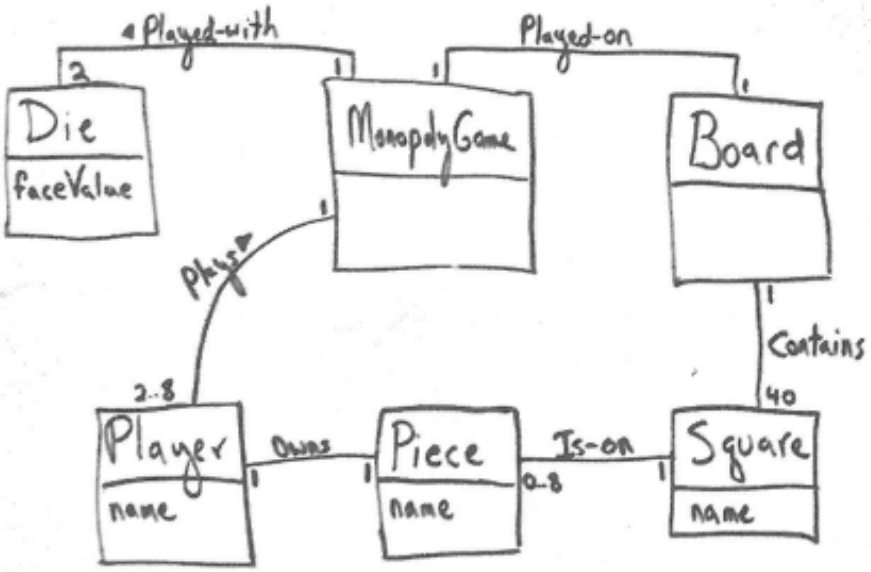


Figure 3.1: The domain model for the first elaboration iteration from Larman 10. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

There are several options according to Larman 10, I've put his discussion of each into a tabular form to aid clarity (see 3.3).

Class Name:	Patterns or Principles	Evaluation
Dog	Lower Representational Gap (LRG)	An arbitrary class could create squares, however this doesn't fit well with our mental model of the problem.
Player	LRG	Conceptually the player is associated with a piece which is associated with a square, again this doesn't fit.
Board	Creator	This is the obvious choice, (and Larman 10 suggests the choice of Software professionals he's encountered world wide), its also in keeping with the Creator pattern.

Table 3.3: Evaulation of the candidates for creating square based on Larman 10.

Note that this is essentially responsibility assignment, we've assigned the responsibility for creating square to Board.

Lets look at the creator pattern (3.4) and then look at the process by which it is used.

Pattern Name:	**Creator**
Problem:	Who Creates A?
Solution: (this an be viewed as advice)	Assign class B the responsibility to create an instance of class A if one or more of these) is true: 1. B "contains" or compositely aggregates A. 2. B records A. 3. B closely uses A. 4. B has the initializing data for A.

Table 3.4: The Creator Pattern Larman 10 using the less detailed format.

Aside: See the sections on Aggregation 3.3.2 a concepts we've hopefully already covered.

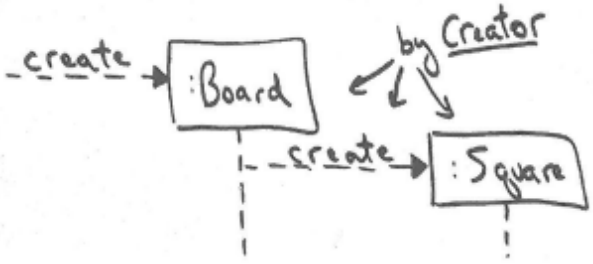
Software Objects and the Design Class Diagram - DCD .
Before applying the pattern we have to understand that we're now working with software objects 1.3.3 and not domain objects [10]. These will later be shown in a DCD (Design Class Diagram,) but where do they come from? We'll work though the process of applying the creator pattern and hopefully it will become clear.

This is the process described in Larman 10, in a simpler format with more descriptive text.

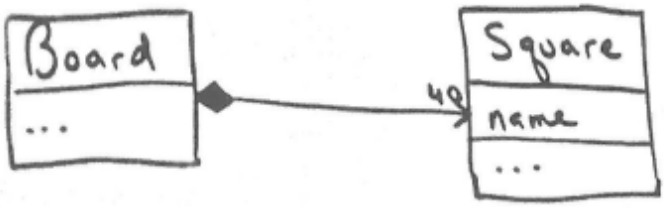
Step	In this example.
1. Identify and describe the problem:	*Who creates square?*
2. Decide on a pattern to solve the problem:	*We've chosen **Creator**.*
3. Look for existing software objects to satisfy the roles of A and B in the pattern: • **iteration *n*:** Look for existing software objects in our DCD. • **iteration 1:** We are just starting the design, using LRG (lower representational gap) identify appropriate domain classes (from the domain model,) and promote them to software classes. Add these to the DCD.	*Board and Square are made software objects, and Board (B) is chosen to be the creator of Square (A) in our example.*

We now have to consider the relation between these two objects. Larman 10 suggests that as board will create squares, they will be part of the board and when board is destroyed it will manage their destruction they should be related by composition. A quick discussion of composition can be found here 3.3.2.

Agile modelling suggests that both the dynamic (behavioural - sequence,) and static (structural - DCD) should be worked on in parallel [10]. Larmans images (see 3.2) are cleaned up whiteboard images, these early stages really need whole team involvement.



Applying the Creator pattern in a dynamic model.



In a DCD of the Design Model, *Board* has a composite aggregation association with *Squares*. We are applying Creator in a static model.

Figure 3.2: The creator pattern being appiled and the start of the DCD from Larman 10. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

3.3 Class Diagrams

In a previous section we looked at the use of raw UML to describe classes at different levels 1.3.3, we're now looking at design level class diagrams and the classes in our diagrams are moving from conceptual to design. We'll recap a little of this, before going on to consider more UML syntax pertinent to this stage.

The UML notation provides the facility to include more detail in a class diagram than shown in the conceptual classes of the domain diagram. Larman 13 provides a single diagram which shows most of the syntax, shown in **??**.

Constructors

It is normal for the constructor to be omitted, if shown it may be prefixed with a stereotype «constructor »[13].

Operations to Access Attributes

Access operations such as *getX* and *setX* are normally omitted (or filtered - in sophisticated tools) as they clutter the diagrams without adding anything interesting [13]. An exception here is any interesting attribute changing methods such e.g. *credit/debit/getBallence* for an Account object. This is perhaps essential when modelling on a whiteboard as these waste space/time an ellipsis ... can be added to show there other attributes/methods are known about but not shown [14].

3.3.1 Associations Between Classes

Much of the syntax has been covered before (when considering conceptual class models,) but we need to take this a little further to consider more sophisticated associations.

Representing Associations

Associations are relationships between instances of classes indicating that some meaningful and interested connection exists between them [7]. We'll consider associations and attributes in more detail next week, however as object diagrams without them are often incorrect we need to know a bit about them up front.
According to Larman 7 associations should be shown when one object needs to show that knowledge of a relationship needs to be preserved for some length of time (milliseconds or years). As is hopefully clear from the section above (3.2.4) many of these relations emerge from the application of basic principles and patterns and can be added to the diagram as these are deployed.
Representing associations is reasonably simple, we use a line with an association name [7] (see 3.3). At the end of each line we have a number or range representing multiplicity (number of) for instances of classes [7]. The association is inherently bidirectional and can be logically traversed in either direction, the association may have a reading direction arrow which suggests how the association name should be read - this has no meaning in the model just helps the reader [7].

Now we begin to consider these relations in terms of design and include restrictions on navigation, multiplicity, names for association end points (roles) within the classes.

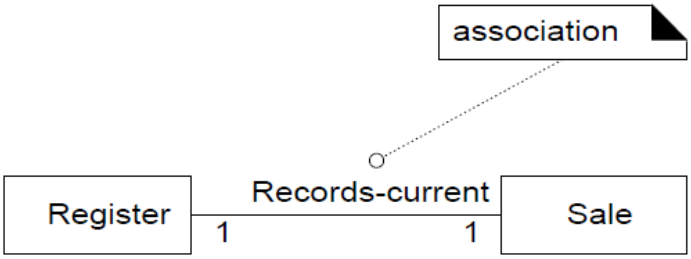


Figure 3.3: From Larman 7, a simple association with unit multiplicity.

Syntax: Associations between Classes

The diagram 3.4 shows a connection between two objects in both the domain diagram form and design class diagram. The top one shows directed association name (see 3.4) and multiplicity at each end (see 3.5). It is more common to see the role names at this stage rather than the relation name, this representation is show in the lower half of 3.4

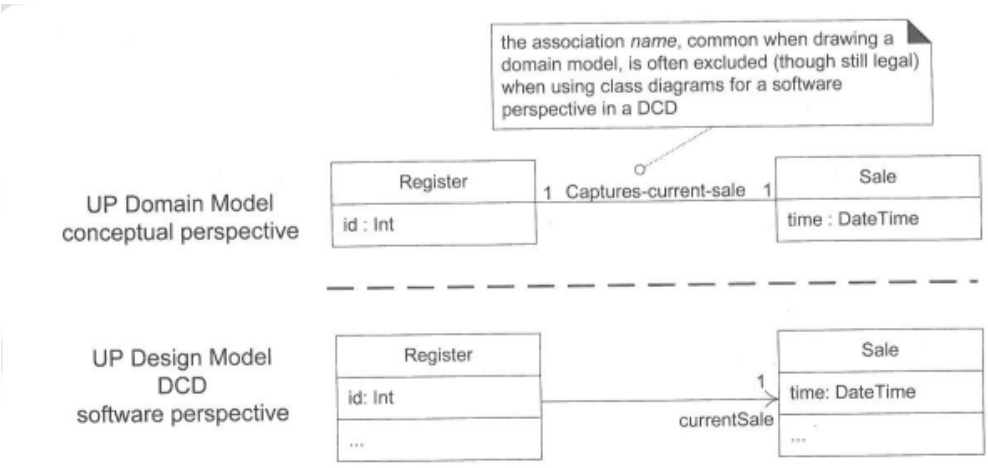


Figure 3.4: An example from Larman 7, this shows the difference in associations between domain object diagram and design class diagram.

Naming Associations

Simple names such as "has" and "uses" are a poor choice as they do not enhance our understanding of the domain [7]. In the example 2.2 "has" could have been chosen at as the association name but would have added nothing to our understanding, however "Lives-at" tells us its their home address. Larman

7 suggests using a *ClassName*-***VerbPhrase***-*ClassName* format which is both informative and can be read. While his examples use a hyphenated format, camel case is also acceptable.

Roles & Multiplicity

The end of each association has a role and can have a name, multiplicity expression and navigability [7].

Multiplicity :

Multiplicity simply defines how many of each class are involved in an association at one time [7]. That is how many instances can be connected, we can improve / extend our earlier example (person-address) by considering that more than one person can live at the same address. It may be that a person lives at a number of addresses over the course of their lives, but for this system they have only one current address. The examples shown in 3.5 are reasonably representative.

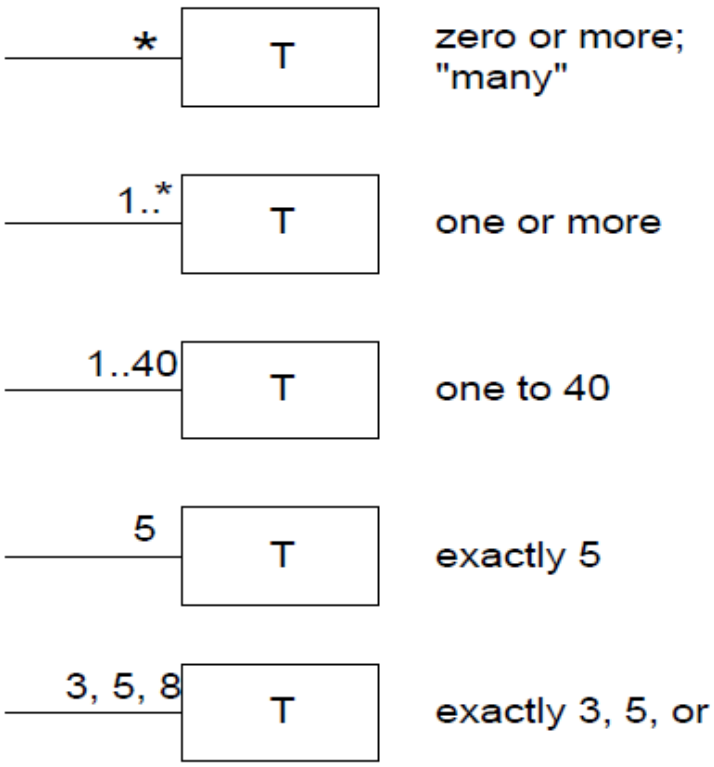


Figure 3.5: A home made example based on those in Larman 7, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image. I've had to screen shot this one to get it to work.

The multiplicity of a relationship is always based on the needs of the scenario, [7] cites Rumbaugh's example where a Person is associated with a Company. For tax purposes this will probably be a list of companies (i.e. all those who the person currently works for) for a Union probably only one.

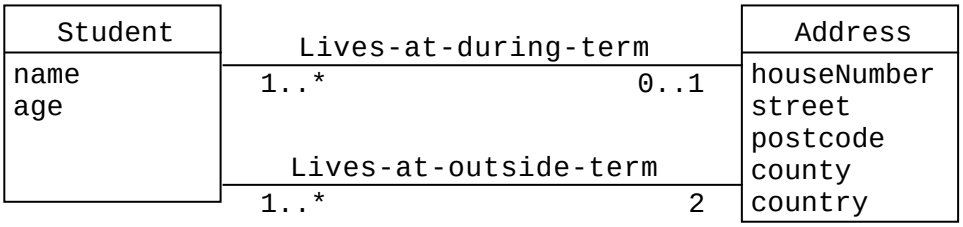


Figure 3.6: A home made example based on those in Larman 7, the tool seems to display the reading direction arrow with no problem but struggles to export this to an image, I've not screen-shot this and its lost its arrows.

Roles :

We can have multiple many to many relations [7], this is perhaps best illustrated though an example see 3.6. Here each student has two addresses a term time address (which may or may not be present,) and a home address. Both of these are address objects, note that one or more students can live at the address in each case.

Association Syntax - Attribute Text and Association Lines

The role name is sometimes used on the association line itself, of may be omitted an included in the list of attributes or the line omitted and the attribute shown with a class type [13]. See the diagram 3.7

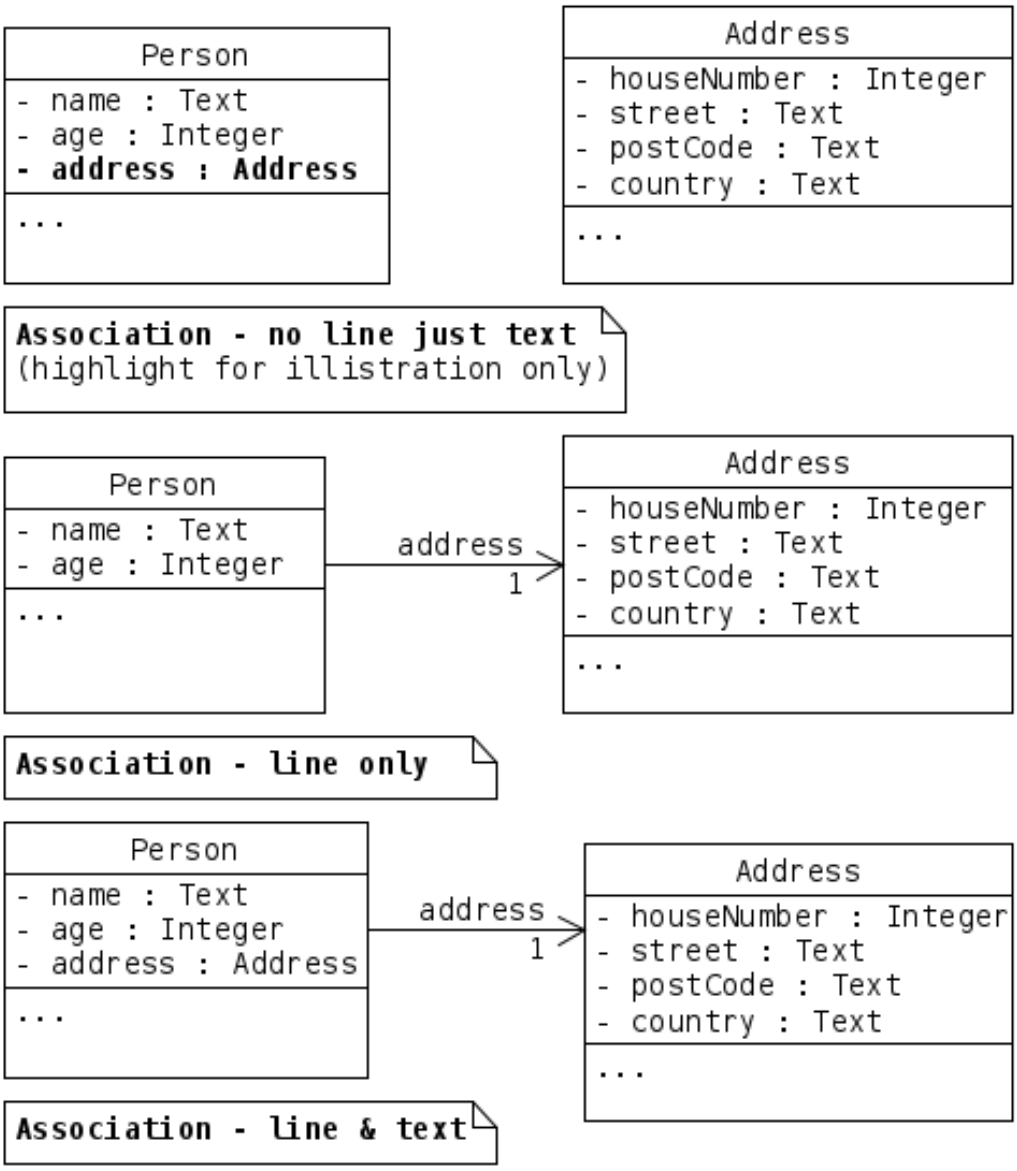


Figure 3.7: Ways in which we can show role names and associations, based on Larman 13.

Larman 14 provides the following guidance:

Type	Guidance	Example
Association Line	The visual emphasis is important, i.e. this needs to be seen. The relationship may be an important one in terms of domain logic.	*Register to Sale, Person to Address etc.*
Attribute Text	The relation would add clutter to the diagram, the relation is an obvious one. The relationship is not important in the context of the domain perhaps to a utility or support class.	*Sale to DateTime, Person to Text etc.*
Both	This is usually what tools and lecturers trying to make a point to their students do.	*Last years Introduction to Programming coursework*

We can go further and include details of multiplicity and property strings (special keywords such as ordered which tell us about the collection of objects) [13], see **??**. I've included this as its something we might want to use in the coursework, I'd not be to worried at this stage however if you omitted the property string.

3.3.2 Specialised Associations

Association is quite a general relationship between objects, we can think of this as two objects which are connected but can exist independently. Where one object is part of or owned and controlled by another we can use UML to show that this is the case.

Composition

Composition (also known as composite aggregation) is a strong kind of whole part aggregation and the most useful [13]. The implications of the relation are [14]:

1. That an instance of the part (e.g. Square) belongs to only one composite instance (e.g. Board) at a time.

2. That the part must always belong to an instance (though it may move from instance to instance).

3. That the composite part (whole) is either directly responsible for creating, managing and destroying the parts or collaborates with others to do so.

Where a class orchestrates the behaviour of a number of internal parts it becomes a structured class for which there is an alternate syntax (we'll come to these later **??**.

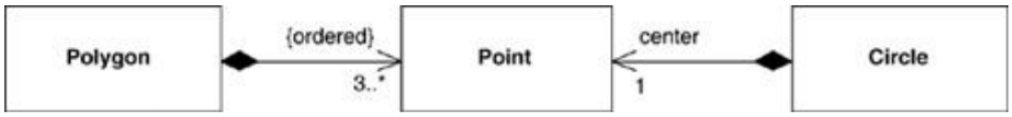


Figure 3.8: Associations (directed with multiplicity) from Larman 13.

One point noted by Fowler 5 is than the association is usually has a multiplicity of one at the part end. The syntax is filled diamond arrow head which points to the whole of the whole-part relation, this is shown in the figure 3.8.

Aggregation

Aggregation is a weaker form of Composition, it loosely suggests a whole part relation but its unclear how this differs from a more traditional associations [14]. I've always interpreted it as a part which cannot be used alone but may exist to be used by other objects. I've always taught this to my students (using the example of a car and a tow-ball,) however both Larman 14 and Fowler 5 quote Rumbaugh (one of the original creators) who suggested it exists because people though it should be there. Both authors go on to say that its probably best to stick with association and composition (or composite aggregation as Larman calls it). Its represented as a unfilled diamond but otherwise identical to composition.

3.4 Sequence Diagrams

You're never going to be tested on this, but to show you how classes talk we need some way of showing how it works. UML exists to show the architecture and behaviour of object orientated systems so its the natural choice.

3.4.1 Participants and Messages

Sequence diagrams are one of two kinds of interaction diagram provided by UML (the other being communication diagrams,) [14], which we're going to

leave to a future iteration. Sequence diagrams are more commonly used than interaction diagrams as they emphasise sequence over structure, making it easy to find the next message (with time going down the page) [2]. A communication diagram by contrast is simply an object diagram (like a class diagram but showing individual objects,) which shows the messages passed during an interaction [2].

Sequence diagrams are the more notationally rich [14] and hence our first choice, 3.9 is based on Larman 14 but with some additional annotations of my own.

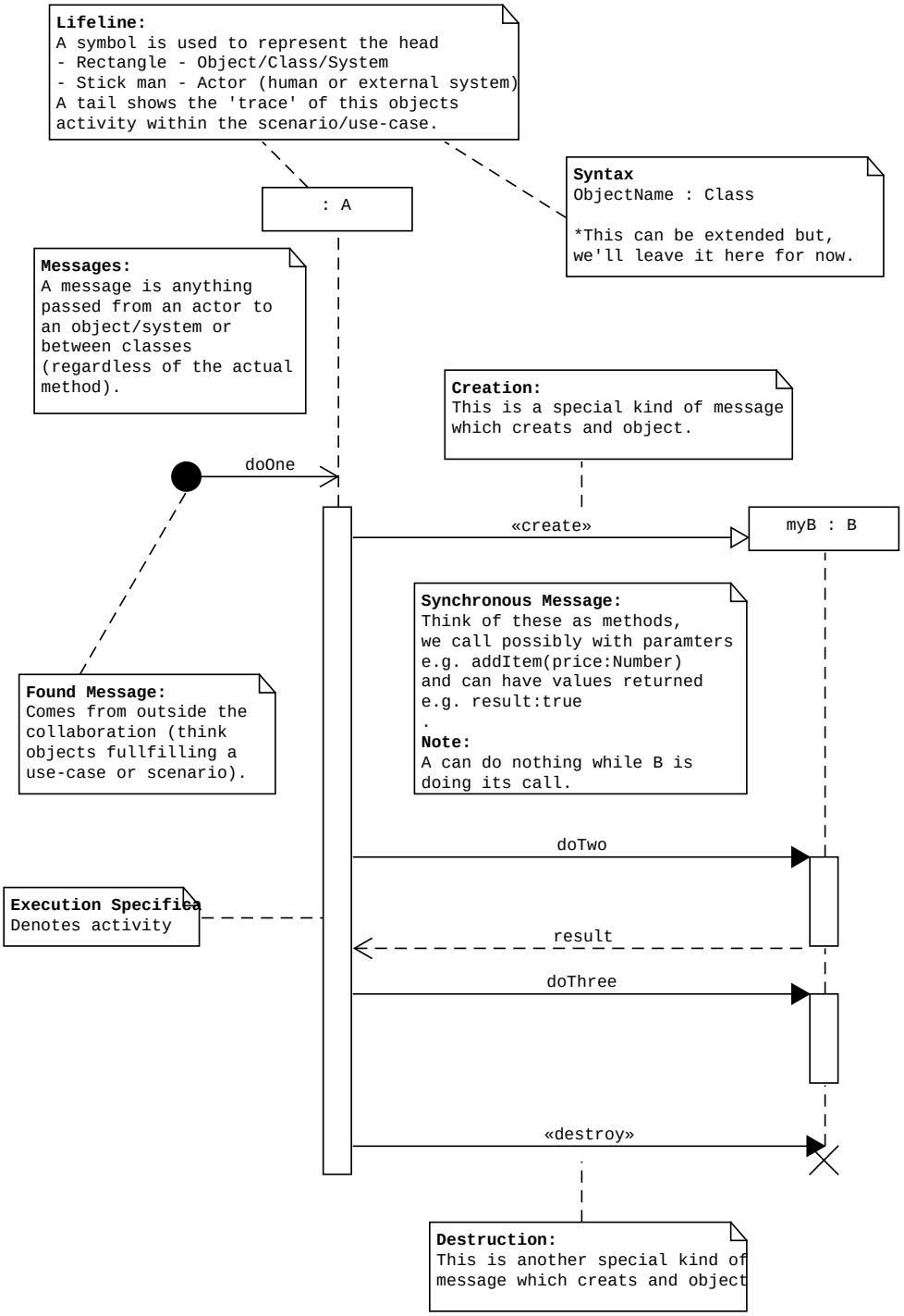


Figure 3.9: Based on Larman 9, augmented with my own annotations inspired by Fowler 6 and Douglass 2

The key elements (in 3.9) shown are:

- **Participant** - This is shown in the rectangle at the top of the lifeline, it has usually has the format **objectName:ClassName** however this can be extended to it can be used to describe multiple objects etc. objectName[n]:ClassName. The UML specification [16] provides more detail (though in a less than readable format). Common participants are show in 3.10.

- **Synchronous Message** - The sequence diagram syntax allows us to show both asynchronous and synchronous messages. The diagram above shows the latter, the best way to think of these is as one object calling a method on another, perhaps with arguments and returns (see 3.4.1 for more examples and discussion.

- **Create and Destroy Messages** These are special kinds of messages which have a stereotype associated with them i.e. «create» and «destroy». These can be though of as calling a notional constructor and destructor for the object. These are not normally shown in System Sequence diagrams.

- **Found & Lost Messages** - These are messages which leave the boundary of the current sequence diagram. Sequence diagrams are normally used to describe an interaction between classes (this could be a scenario, single use case, part of a use-case etc.) where messages enter or leave context they are represented as lost and found messages.

- **Execution Specification** - Processing within a method is often shown as a block on the lifeline, these are often used to reinforce the difference between synchronous and asynchronous messages. These are more commonly seen in diagrams created with UML tools and known as the Activation Bar in UML 1.0 [14].

NOTE: The syntax of sequence diagrams has changed over the various versions of UML, there are also a number of tools, books, authors who appear to impose their own limitations or add their own flourishes. Please make every effort to use the correct syntax, if your in any doubt check the notes. If you're writing your diagrams up in a tool and it is unable to generate the correct syntax, put a note to explain the issue. As with the other diagrams I'm not looking for perfect syntax (the logic is more important,) however there is a point at which it becomes just lines and boxes and is no longer UML.

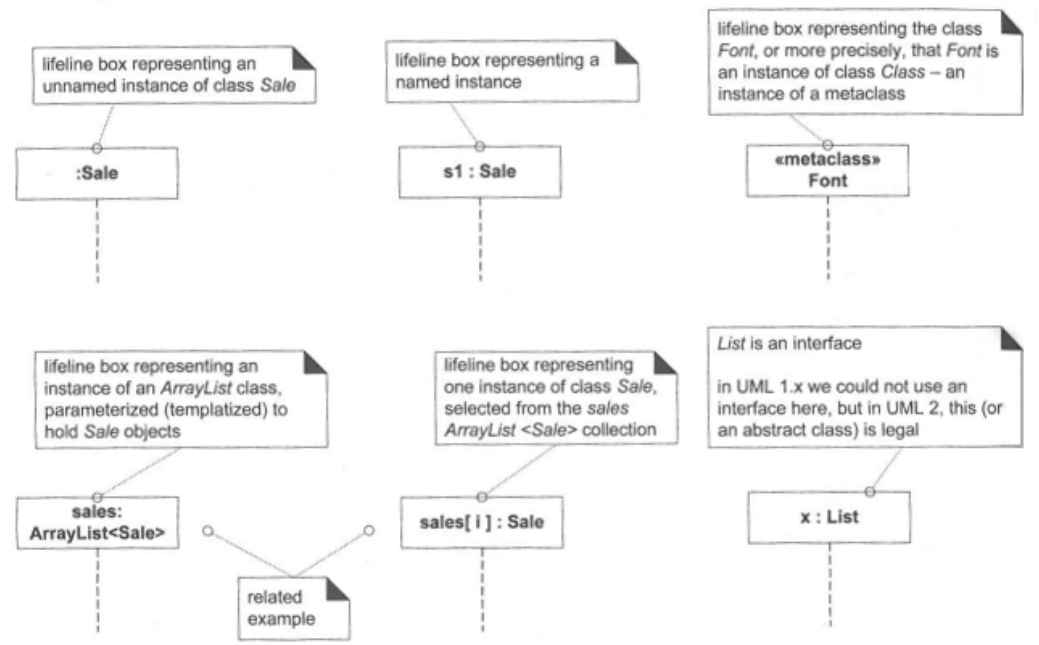


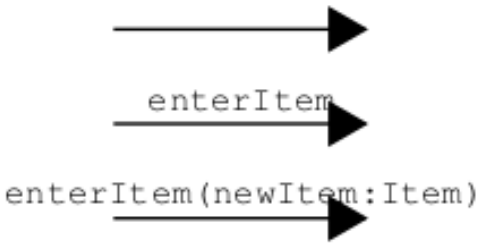
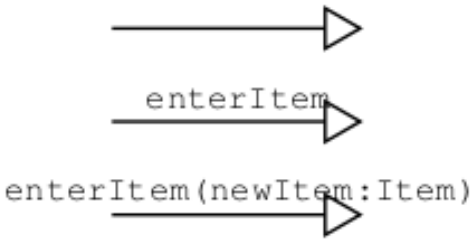
Figure 15.5 Lifeline boxes to show participants in interactions.

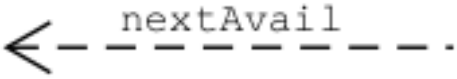
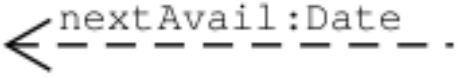
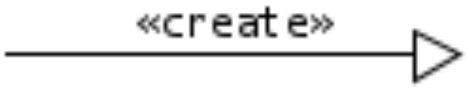
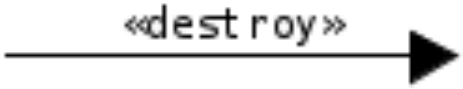
Figure 3.10: From Larman 14.

Larman 14 provides some excellent documentation on the Sequence diagram including the flowing examples of the kind of classes which may be involved at the design phase.

Messages

As mentioned above there are a number of different kinds of message which can pass between instances (or classes) in a sequence diagram, these are:

Type	Description	Symbol
Asynchronous	A message is send but the sender does not wait for a response, e.g. an event is sent to an event queue.	
Synchronous	A message is send and the sender waits until the processing is complete in the receiver before continuing. There may be a reply message involved 3.4.1	

Replies	These are messages returned as the result of a call on another object, you can think of them as method returns. There is an alternative syntax (Larman 14 provides an example, we'll stick with one way to avoid confusion.	 
«create»	A special kind of synchronous message resulting in the creation of an object.	
«destroy»	A special kind of synchronous message resulting in the destruction of an object.	

Asynchronous communication is most common with real-time systems where objects have their own threads of execution, e.g. in multi-process or multi-threaded design where there is more than one active object.

All messages have the full form

return = messageName(paramter : paramterType) : return type

However it is not uncommon to see, calls for specific programming languages used here [14] this is common where the diagram has been generated from code, or is being used to generate code Model Driven Development (MDD). Larman 14 also suggests that empty parameter lists and void returns are often omitted, type information may also be omitted if obvious [14]. I'm not sure the latter is a good idea!

3.4.2 Frame Operators

In the earlier section we breifly consdiered Frame Operators here' we'll recap these there are still some we're not going to use right away, but in the next few classes we'll be introducing them as requried (probably along with more complex patterns). The following table is based on Larman 14 and Douglass 2.

Operator	Description
sd	Names an interaction fragment. This could be a system sequence diagram, a realisation of a use case (i.e. how objects from our class diagram will interact to get the job done,) a part of a sequence diagram we want to re-use in other diagrams. **Not covered here in detail, we don't need them until later see ??**.
ref	References another interaction fragment, i.e. where we have parts we wish to re-use (e.g. authenticateUser) we can refer to them with this. **Not covered here in detail, we don't need them until later see ??**.
alt	This provides a number of alternatives, one of which will be taken based on the guard conditions (in [ ]) an else guard is also provided which evaluates to TRUE if all others are false. This is a complex fragment [2] and is used in UML to express both if/else and switch/case type programming structures. **Note:** Larman 14 suggests the guard should be placed over the life-line of the participant who performs the test. See figure 3.11.
opt	A kind of simplified alt, a fragment which will only be executed if its guard is true. See figure 3.13.
loop	Specifies that a fragment should be repeated until some condition (expressed in a guard) becomes FALSE. There is no official **for** syntax though it is suggested one is being considered [14]. See figure 3.12.
par	This is similar in syntax to alt however, here all compartments execute rather than one. **Not covered here in detail, we don't need them until later see ??**.
region	Identifies a region which is atomic, and cannot be interleaved with others, i.e. the actions in this fragment must take place in this order with no possibility of interruption. **Not covered here in detail, we don't need them until later see ??**

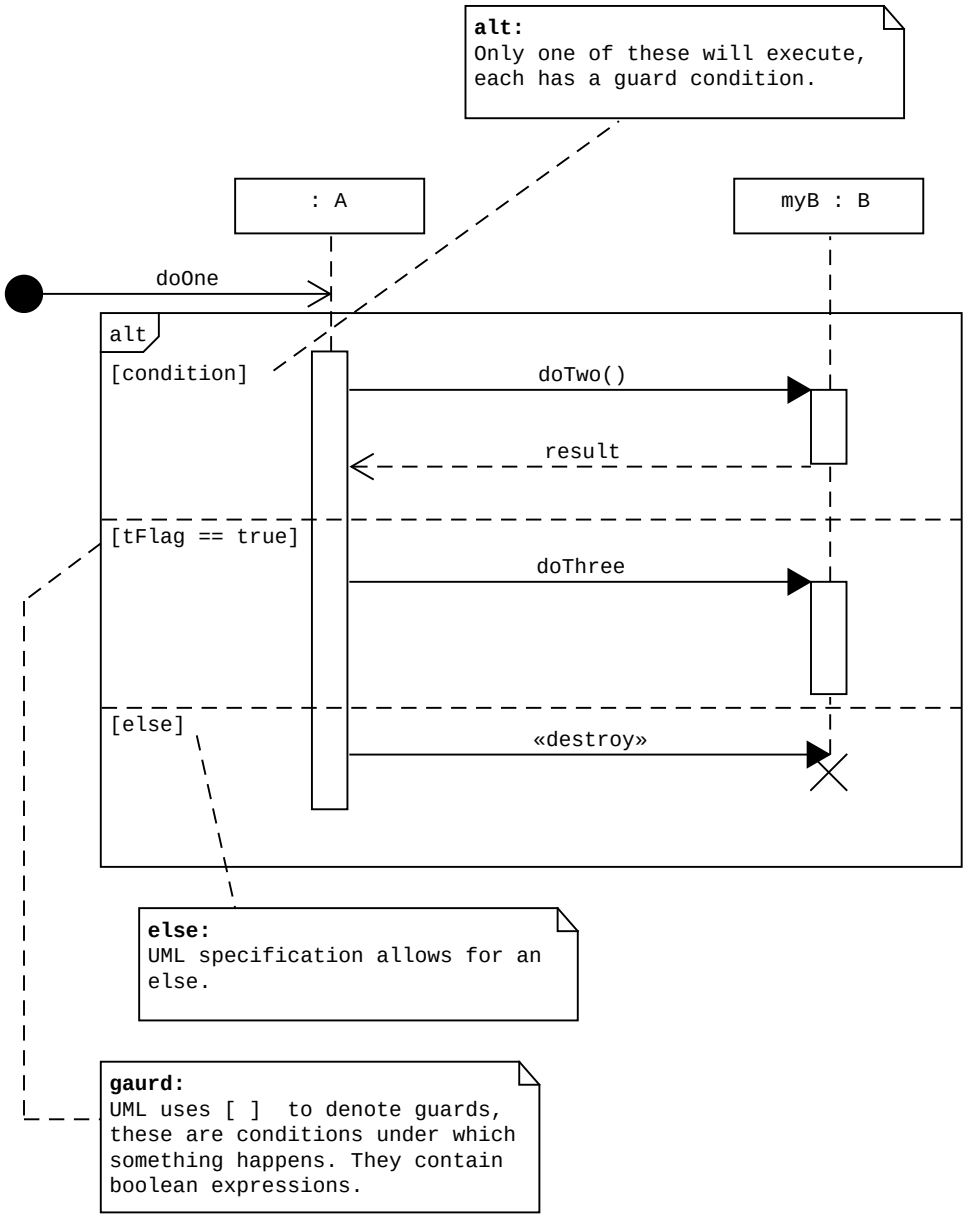


Figure 3.11: Syntax for the alt interaction fragment, this is used for both if/else and switch case type structures.

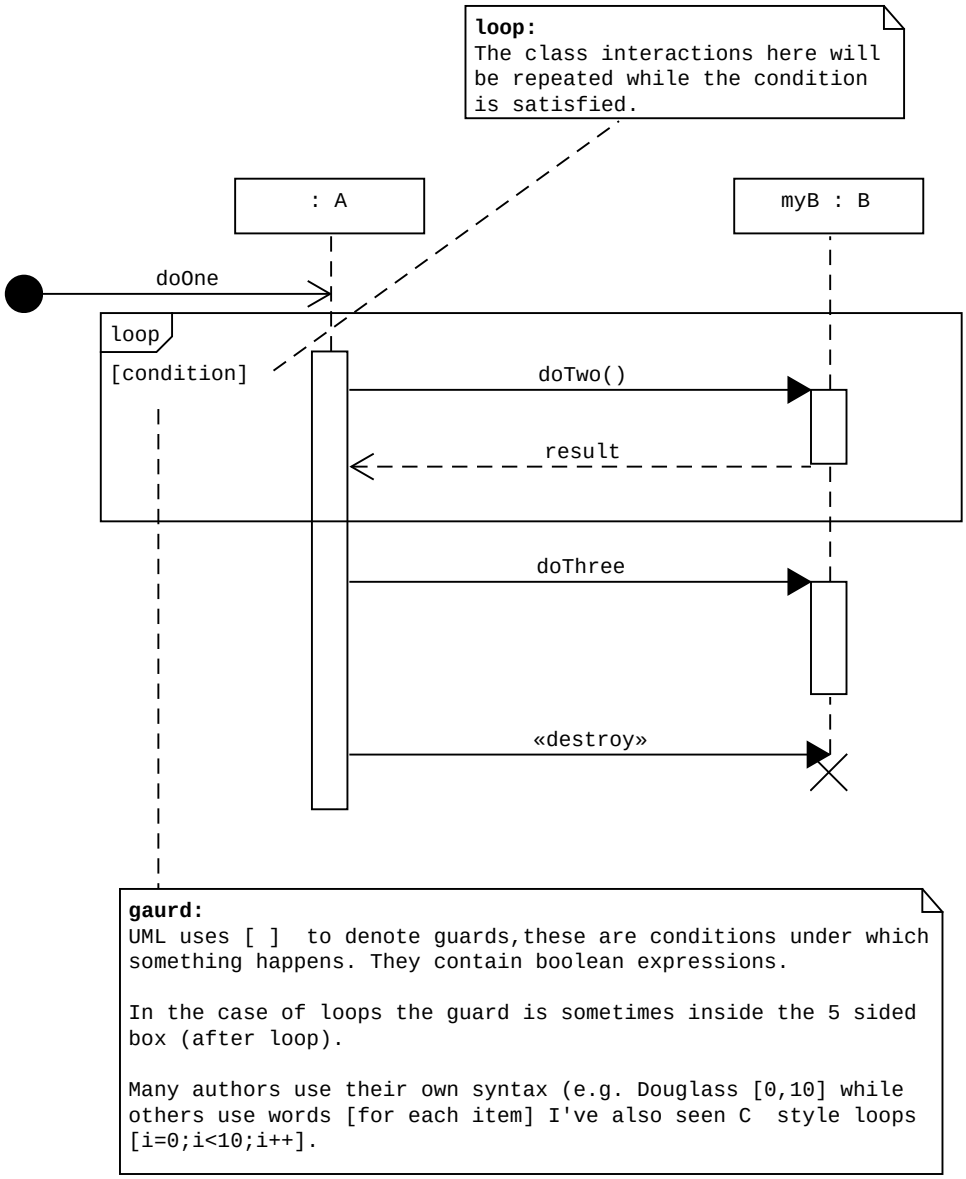


Figure 3.12: Syntax for the loop interaction fragment, this is used for both while and switch for structures.

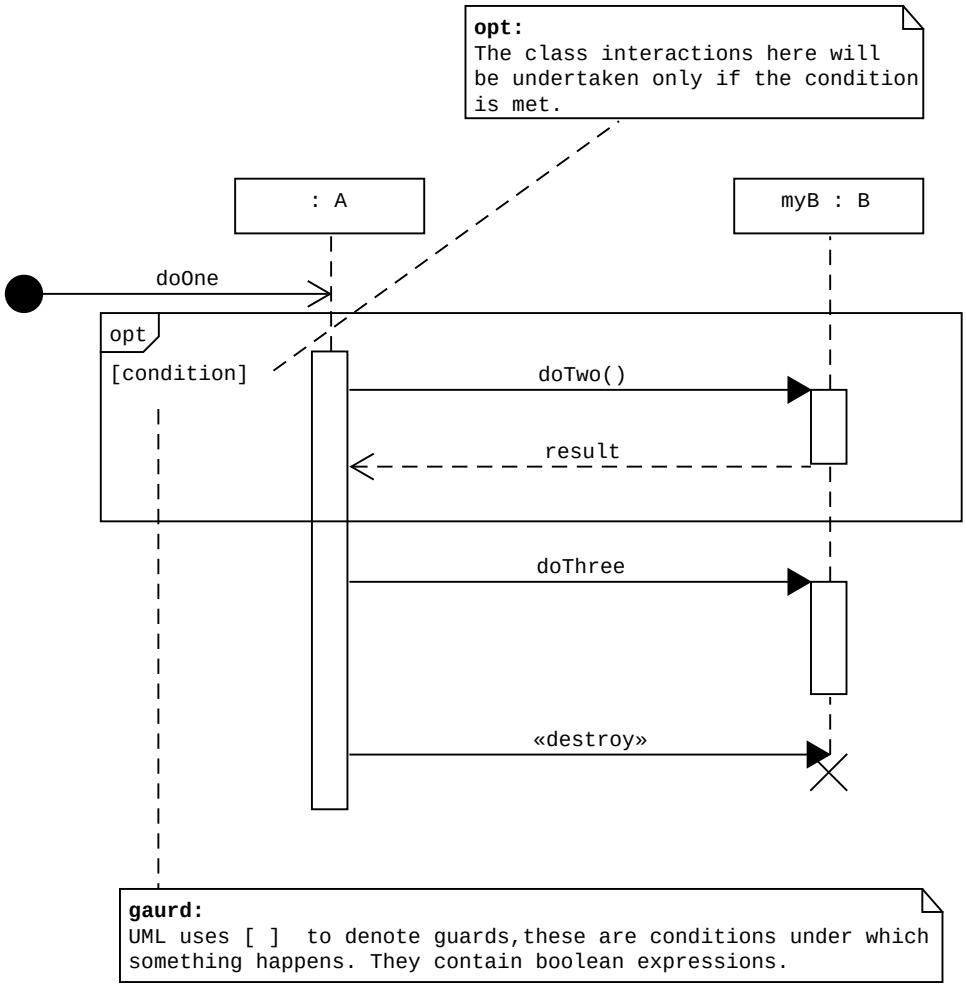


Figure 3.13: Syntax for the opt interaction fragment.

3.5 Tutorial

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

1. Read the case study 3.5.1, and consider the use case 3.5.1 and domain object model3.5.1 then use an appropriate pattern to identify:
   - Who creates Unit?
   - Who creates Format?

2. Read the case study 3.5.2 the domain object model provided 3.5.2 then use an appropriate pattern to identify:
    - Who creates InputPattern?
    - Who creates OutputPattern?

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

- Explain the Model-View Separation Principle and why such an approach would be used in the development of a commercial video game.

3.5.1 Case Study 1

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

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

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

Scenario – Sunny Day 2

A user opens the application and is presented with a screen and appropriate input boxes for the default conversion. The user decides not to perform the

default conversion and uses a button/toggle to swap to metric to imperial conversion. The user interface updates to reflect this (labels change etc). The user enters the feet and inches and presses the covert button. The application performs some validation on the input, the results of which is positive. An output box is filled with the answer in an appropriate format such that it can be copied by the user.

Scenario – Formatting error scenario

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

Use Case

The team have produced a number of use case summarised at the highest level by the following use case diagram. The names of the use case here may differ slightly from those we identified during the tutorial.

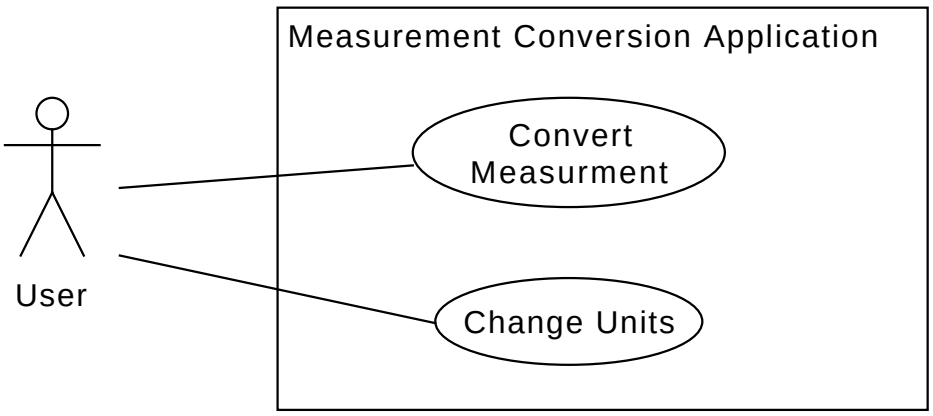


Figure 3.14: This is an incomplete use case based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough.

Use Case: Convert Measurement

UC1: Convert Measurement L2	
Scope:	Measurement Conversion Application
Primary Actor:	User
Level:	user-goal

Stakeholders and Interests:

- User: Wants to convert between imperial and metric measurements of length.

Preconditions:

- Application has launched and the interface is displayed
- Interface is prepared for user input

Postconditions (success guarantee):

- Output is presented in a suitable format

Main Success Scenario:

1. User enters a measurement in the default unit (imperial,)
2. User requests a conversion (indicating they have completed their data entry).
3. System performs the conversion
4. System displays the converted measurement

Extensions (Alternative Flows):

- 1a. User changes units before entering a measurement.
- 2a. User enters nothing
  1. System does not attempt to convert
  2. System highlights the lack of input (e.g. change background of input field to red)
- 2b. User enters an incorrectly formatted measurement for the current conversion.
  1. System Displays an error message
  2. System highlights the incorrect input (e.g. change background of input field to red)
  3. User enters a measurement

Special Requirements:

- Must be possible to enter the measurement in a number of formats, e.g. 1'6", 1.5", 18" etc.
- The output must be such that the user can easily copy the output for use in another application.

Technology and Data Variations List:

1. ...

Frequency of Occurrence:	Expect the user to perform one or more conversions in a single session

---

Open Issues:

- Given that the input can be in a number of formats, which the system is to deduce. How will this be done?
- The output may be required in a specific format, should this option be given to the user and how? This could be a property set though a menu (with a default,) or some means of selection near the output.

Domain Object Model

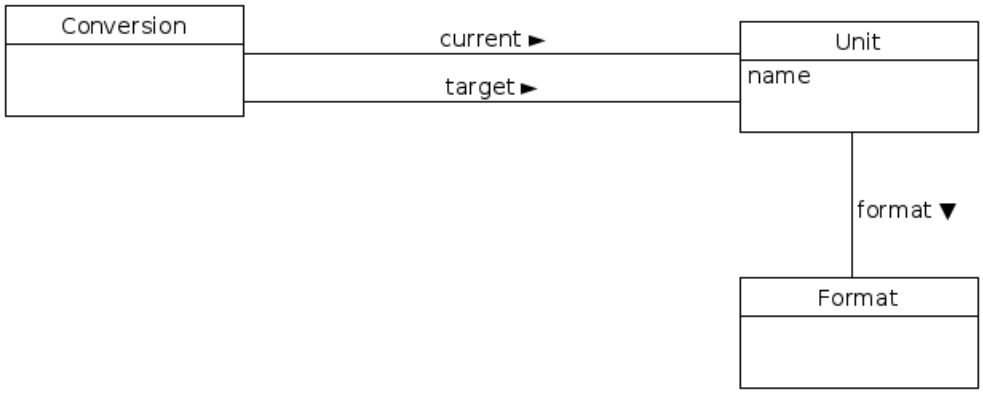


Figure 3.15: This is a domain object model based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough. I've avoided describing this as a draft (suggesting its incomplete,) or using any other term which would suggest it is final. In the spirit of Agile Modelling we are using it to inform sketches of the design so we can discuss and evaluate design options. Changes may be made as a result of this discussion and its probably that future iterations would contain some of the conceptual classes.

3.5.2 Case Study 2

A researcher at the university is looking to write a paper with a colleague they have both datasets and a new neural network to train on them. In order to do this they need to load in files from the original project so that their Neural Network will be trained and tested with the same files. Each file consists of comments (which start with a #,) some meta data (information about the file content) and the data itself (see example file), all in a CSV format. The data is in the form of input and output patterns, each contains t (a time offset). The input contains the values for parameters t, r, g, and b while the output patter has two values t and o. Due to the nature of neural network operations all parameters are floating point numbers between -1.0f and 1.0f further validation is possible as the t (which represent the time-step,) should be identical in the input and output pattern.

DCD for Case Study 2

Another knocked up DCD, this one is similar to this years coursework.

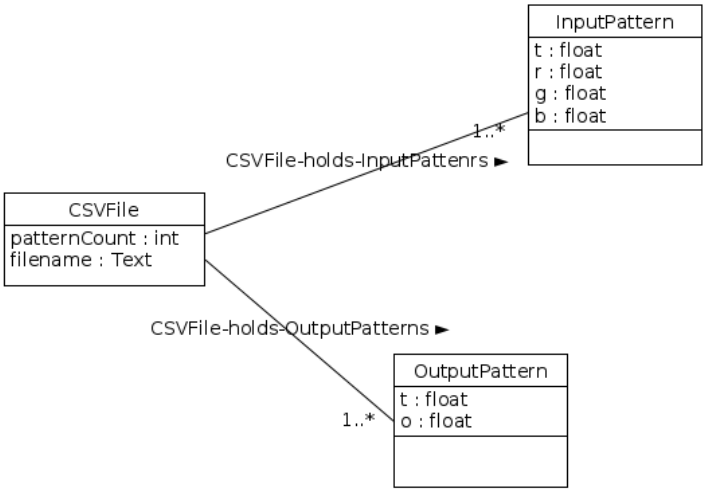


Figure 3.16: Domain Object Model for Case Study 2.

3.6 Summary

The next important part of the design process is the Realisation of the use case, which is the mapping of use case and individual scenarios (descriptions of system behaviour,) to domain (and non-domain) objects. We've started small considering parts of these interactions in isolation and the low level patterns (or OO principles,) we've looked at one of these patterns so far and there are quite a few more to go!
The GRASP patterns are low level simple design patterns/principles named and documented to make them easier to remember, discuss and use. This week we've looked at Creator with some examples of its use, however we've not considered the pattern in full (which we'll do in a later class) at the moment we need to see a few more patterns so we can start using them in the realisation of use case.

3.7 Preparation for Next Week

3.7.1 Essential Reading

An introduction to GRASP and Patterns, this covers this week and next weeks class (maybe the week after).
Larman (2014d). *Applying UML and Patterns*, chapter GRASP: Designing Objects with Responsiblities, pages 312–362. Prentice Hall PTR.

3.7.2 Background Reading

In conjunction with the above you may need to refer to these chapters which provide very detailed coverage of Class and Sequence diagrams. Larman (2014h). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

Larman (2014g). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

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

Responsibility Driven Design and GRASP - Part 2

4.1 Introduction

In the previous notes (3) we'd started object design and were looking to begin mapping the behaviour of the system to these. When we study this in more detail we'll have the analysis artefacts (use case, SSD, etc to guide us,) however, for the simple problems we'll solve between now and then I'm hoping a simple set of patterns will be enough.

In an attempt to systematically design object orientation systems adopted a responsibility driven design approach (RDD). This suggested the use of GRASP patterns in order to assign low level responsibilities and we'd looked at one of these patterns. I'll recap the key concepts before adding more patterns.

4.2 Recap: Responsibilities and Responsibility Driven Design (RRD)

In RRD objects are attributed responsibilities, in terms of what they are responsible for knowing and what they are responsible for doing [10].
There are two types of responsibility knowing and doing [10]:

- **Doing** - Responsibilities may include:
  - Doing something itself, e.g. a calculation, creating and object etc.
  - Causing something to happen in another object, or changing its state.
  - Controlling and coordinating activities in other objects (e.g. **??**)

- **Knowing** - Responsibilities may include:
  - Holding data itself in private attributes
  - Knowing about related objects
  - Knowing about data it can derive or calculate

4.3 Responsibilities and Responsibility Driven Design (RRD)

4.3.1 Recap: What is RRD?

GRASP is a learning aid which names and describes some basic principles to assign responsibilities to objects, proving a simple first step towards RDD [10]. I use this in the second year to provide a foundation for applying more complex patterns such as the Gang of Four (GoF) patterns which are 'larger' and deal with coordinating higher level behaviour.

Some of these (or perhaps all,) will not really be new to you, but you won't have seen them named and their pros and cons discussed. Naming them allows them to be explained and applied [10] leading to a more systematic approach to the delegation of responsibilities (and therefore object behaviour). We're going to come back to these and discuss them in more detail at a later stage, without a name we couldn't do this!

Recap: When do we assign Responsibilities?

Responsibility are assigned either during design when we draw interaction diagrams and decide where the methods which fulfil these responsibilities go or while coding [10]. The GRASP principles can be applied to guide choices about assigning responsibilities and can therefore be applied during design (while drawing interaction diagrams,) or during coding [10].

Patterns

Experienced software developers build up and deploy a "repertoire of both general principles and idiomatic solutions" (that is a generalised bits of solutions,) which if codified in a appropriate format, i.e. named, with a description of the problem and solution give a pattern [10].

According to Larman 10 named patterns provide:

- **A Memory/Learning Aid** It helps 'chunking' and incorporating that concept in to our understanding and memory. That is if we see something described as a pattern (and named,) we can think of it as a pattern and remember it, perhaps associated with other similar pattern.
- **A Communication Aid** This also facilitates communication we can talk about using the Abstract Factory Pattern, to solve a problem and other developers instantly know what we're talking about

Patterns are usually expressed in terms of a general problem and solution which can be applied in many contexts, there may also be some consideration of the trade-offs to be considered [10]. The pattern will then suggest which responsibilities should be associated with which objects (given their roles in the pattern) [10]. The GRASP patterns (or mini patterns,) do not state new ideas they name widely used principles, to an expert they should be fundamental and familiar [10]. To you (second year students,) they'll hopefully be thing which you were starting to get to grips with in your object originated coding, or even guidance for problems which are on the horizon in your coding.

4.3.2 Solving A Simple Design Problems using a GRASP Patterns - Continued

Rather than start with a full scenario (or use case,) which is where we'd normally start in terms of mechanistic design, I'm going to start with some 'parts' of a larger solution. Seeing these simple patterns deployed in a number of examples will allow us to work up to a full scenario, essentially this will be the detail behind the call to system from our SSD (see **??**). I've stolen these examples from Larman 10, we'll work though something similar for other examples later.

Information Expert (or Expert)

Question: Who Knows about the Square object, given a key?

Larman 10 provides a very detailed discussion of this example and the case study it comes from, this is a much abbreviated version and I recommend you read the full one. Larman really explains his though process well.

Another common problem is working out who should *know* something, this is a 'knowing' responsibility which must be assigned to some object [10]. Again

the example we'll use is from a scenario/use-case for the first elaboration iteration of a Monopoly game, where the game is initially set-up, a single piece is added to the board and moved around the board by dice roles (without player interaction). **? ?** covers this in some detail but we're only looking at some elements of it anyway.

The question is:

"Who knows about Square given a key?"

First lets look back at the object domain model to see what classes we have in addition to square (see 4.1).

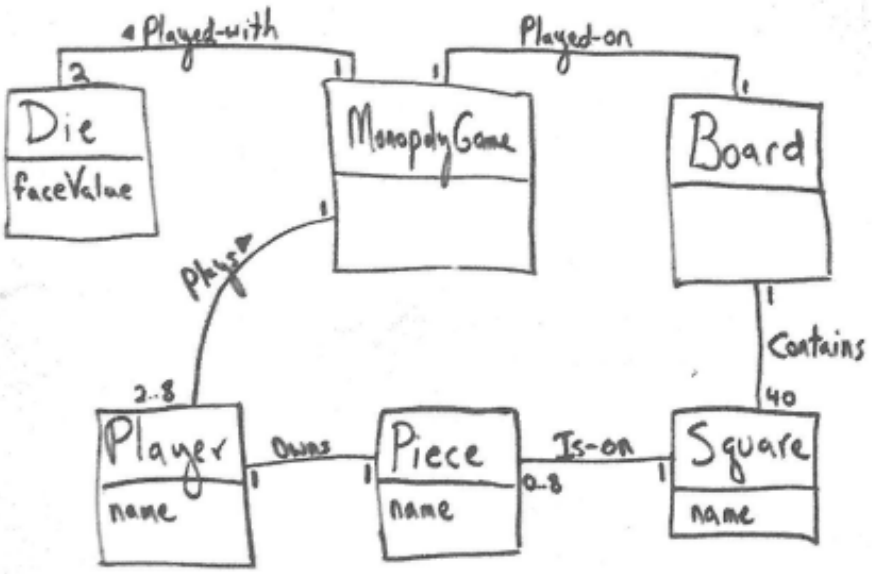


Figure 4.1: The domain model for the first elaboration iteration from Larman 10. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

There are several options according to Larman 10, I've put his discussion of each into a tabular form to aid clarity (see 3.3).

Class Name:	Patterns or Principles	Evaluation
Dog	• Lower Representational Gap (LRG) • Low Coupling (another pattern see **??**)	• An arbitrary class be responsible for knowing create squares, however this doesn't fit well with our mental model of the problem. • Informally coupling refers to a measure of connectedness between elements, minimising this reduces the impact of change on one class on others.
Board	Information Expert	This is the obvious choice (and Larman 10 suggests that of most OO developers,) for an object to find an present one square (given a name,) it must know (have the information) about all the Squares.
Piece	Information Expert	This may fit conceptually, the piece exists in our domain. However in terms of having the information required it is hopefully clear that while piece knows its own square it doesn't know any others. Therefore it cannot fulfil the responsibility.

Table 4.1: Evaulation of the candidates for creating square based on Larman 10.

Note that this is essentially responsibility assignment, we've assigned the responsibility for finding square (with a given name) to Board.

Lets look at the creator pattern (4.2) and then look at the process by which it is used.

Pattern Name:	**Information Expert**
Problem:	What is the basic principle by which we assign responsibilities to objects?
Solution: (advice)	Assign a responsibility (knowing or doing,) to the class that has the information needed to fulfil it.

Table 4.2: The Creator Pattern Larman 10 using the less detailed format.

As suggested above this pattern can apply to either knowing or doing,[10]. For example if we were looking to find the class that reset the squares (in the event of reset or new game,) we would assign the responsibility of doing this to Board, again following the Information Expert pattern.

Software Objects and the Design Class Diagram - DCD .

Again we're working with software objects in the Design Class Diagram (DCD) but we're off the starting line and have some objects already in our model, so the process changes slightly.

This is the adapted from that described in Larman 10, using the same format as we've used earl

Step	In this example.
1. Identify and describe the problem:	*Who Knows about the Square object, given a key?*
2. Decide on a pattern to solve the problem:	*We've chosen **Information Expert.***
3. Look for existing software objects to satisfy the roles of A and B in the pattern: • **iteration *n*:** Look for existing software objects in our DCD. • **iteration 1:** We are just starting the design, using LRG (lower representational gap) identify appropriate domain classes (from the domain model,) and promote them to software classes. Add these to the DCD.	*Board and Square are already software objects, and Board is chosen to be the Information Expert*

We now have to consider the relation between these two objects, fortunately the existing composition relation does not need to be changed. However the ability to fulfil this responsibility must be reflected in the relevant diagrams. Agile modelling suggests that both the dynamic (behavioural - sequence,) and static (structural - DCD) should be worked on in parallel [10]. Larmans images (see 4.2) are cleaned up whiteboard images, these early stages really need whole team involvement.

NOTE: Remember this are part diagrams, no one in their right mind would draw a class diagram for these other than for illustrative purposes. Imagine these fragments as part of a complete sequence diagram for a given scenario or use case.

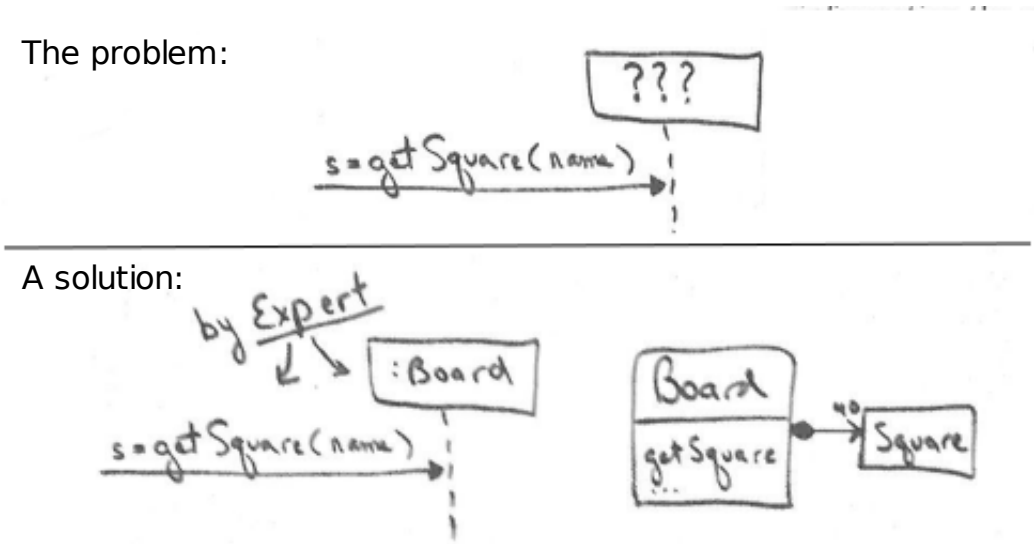


Figure 4.2: The expert pattern being appiled and updated DCD from Larman 10. This is how agile modelling should work, hand drawn diagrams created through group consensus on a whiteboard and digitised.

Low Coupling

Question: Why Board over Dog?

Larman 10 provides a very detailed discussion of this example and the case study it comes from, hopefully my cut down version has left enough context for this to still make sense.

In the Creator (see 3.2.4) and Expert (see 4.3.2) we assigned responsibilities to Board rather than some other arbitrary class, e.g. Dog. The Expert Pattern suggested that Board which knows about all Squares, should be responsible for finding an individual Square given a key. Underlying this is the principle of Low Coupling, which we introduce though this now familiar example, but which is a central goal in the creation of software [10]. Informally coupling is a measure of how strongly one element is connected to, depends on, knows of etc., other elements [10]. For example if element A is tightly coupled (or dependent upon,) B then any change in B will have an impact (require changes in,) A. This is a very general principle which occurs at many levels of software development [10] and one you may have come across before in first year programming.
So the question is:

"Why Board over Dog?"

Lets look at the Low Coupling Principle (4.3) in the same format as the other patterns/principles we've looked at, its important to remember that these all provide advice. In this case Low Coupling supports Information Expert [10], in other cases principles will conflict and we'll need to weigh up the arguments for each.

Pattern Name:	**Low Coupling**
Problem:	How to reduce the impact of change?
Solution: (advice)	Assign a responsibility so that **unnecessary** coupling remains low. Use this principle to evaluate alternative solutions.

Table 4.3: The Low Coupling Principle Larman 10 using the less detailed format.

This is a little different to the first two patterns we looked at, we use Low Coupling to evaluate design choices (or existing designs as targets for refactoring,) unless there are external factors we favour the solution with the lowest coupling [10]. We'll start with an alternative solution to a problem we've already solved "Who knows about a Square object, given a key?", this will allow us to evaluate this against the solution suggested by the Information Expert pattern this is shown in 4.3.

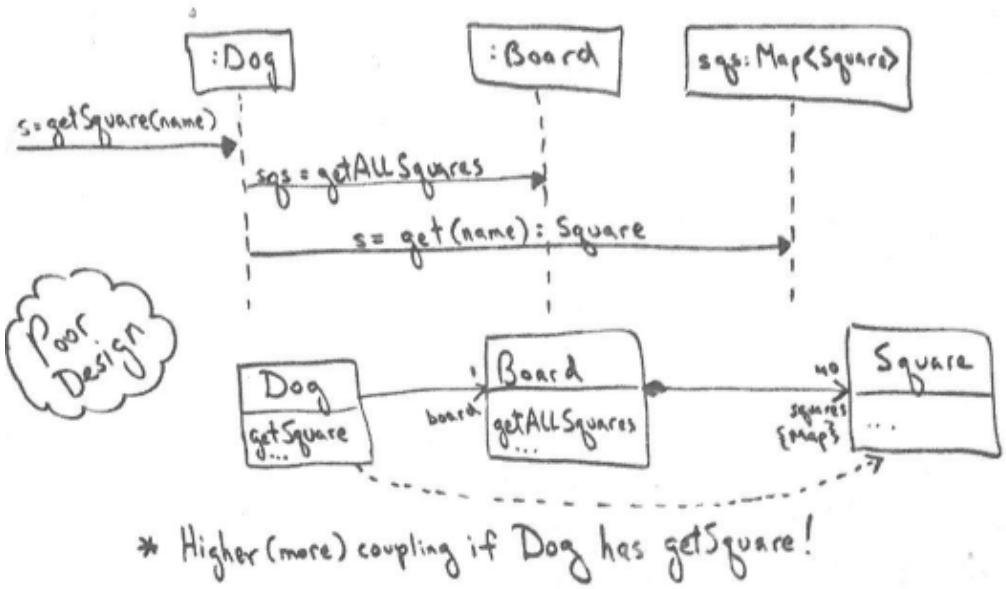


Figure 17.7 Evaluating the effect of coupling on this design.

Figure 4.3: An alternative solution to the problem of "Who knows about a Square object, given a key?" from Larman 10. The original (superior) solution is show in 4.2

Going back to our previous evalaution based on Larman 10, but using the tabular form to aid clarity (see 4.3).

Class Name:	Patterns or Principles	Evaluation
Dog	• Lower Representational Gap (LRG) • **Low Coupling**	• An arbitrary class be responsible for knowing create squares, however this doesn't fit well with our mental model of the problem. • Coupling refers to a measure of connectedness between elements, minimising this reduces the impact of change on one class on others.
Board	Information Expert	This is the obvious choice (and Larman 10 suggests that of most OO developers,) for an object to find an present one square (given a name,) it must know (have the information) about all the Squares.
Piece	Information Expert	This may fit conceptually, the piece exists in our domain. However in terms of having the information required it is hopefully clear that while piece knows its own square it doesn't know any others. Therefore it cannot fulfil the responsibility.

Table 4.4: Evaulation of the candidates for creating square based on Larman 10.

We could assign the responsibility to the Dog class (as shown in 4.3) but this would mean that Dog had to get all Squares from Board, then know enough about the Square class to find the one with the right name [10]. This requires Dog to know about (or be coupled to,) both Board and Square! The solution suggested by Information Expert (i.e. Board has findSquare(name),) requires only that Board knows about Squares i.e. Board is coupled to Square - the coupling is lower for this solution.

Software Objects and the Design Class Diagram - DCD .

This time we're not changing any objects but evaluating existing or alternative solutions [10]

There is no process described in Larman 10, using the same format as we've used earlier, I've tried to provide a framework for the process.

	Step	**In this example.**
1.	Identify and describe the problem:	*Why Board and Not Dog?*
2.	Decide on a principle to evaluate the solutions:	*We've chosen* ***Low Coupling***
3.	Consider the Coupling of each Solution: • **Dog** Requires Dog to be coupled to Board and Square. • **Board** Requires Board to be coupled to Square.	*Board and Square are already coupled (due to creator pattern,) and this solutions gives the lowest coupling Board is chosen. This also fits with Information Expert which itself aims to reduce coupling.*

The diagrams in this section introduces some new syntax (see the section on class diagrams 4.4 and sequence diagrams 4.5 for details).

Controller

Question: What is the first object after (or beyond) the UI layer should receive the message from the UI layer?

Larman 10 provides a very detailed discussion of this example and the case study it comes from. Again the example we'll use is from a scenario/use-case for the first elaboration iteration of a Monopoly game, where the game is initially set-up, a single piece is added to the board and moved around the board by dice roles (without player interaction).

In a layered architecture the UI layer handles all interactions with the user, i.e. showing windows, receiving mouse clicks and keyboard presses etc. [10]. In this case a button press results in a message being sent from the UI layer to some class '???' in the domain layer who controls/coordinates this activity.

From the Model View Principle (see **??**) we know that UI objects should not contain business logic, (e.g. working out the next square etc.,) so this must be delegated to a domain object (in the domain layer) [10].

So the question is:

"What is the first object after (or beyond) the UI layer should receive the message from the UI layer?"

First we review the SSD (see 4.4,) this shows a message generated by the human observer *playGame* [10]. Larman 10 goes on to discuss the potential sources, essentially when the code which the button runs (however it runs it - this will differ from Qt to Java, .Net etc.) this must create a call to the System. I think Larman's choice of example is an issue here, so lets think back to the other SSDs we looked at. Or jump back to **??**, here *addItem(item, quantity)* was the message. While the conversion of an event like a gui button on the screen being converted to a call on a domain object is trivial, the *addItem* example requires some information to be collected from other sources (probably text boxes, filled from the keyboard and bar-code reader).

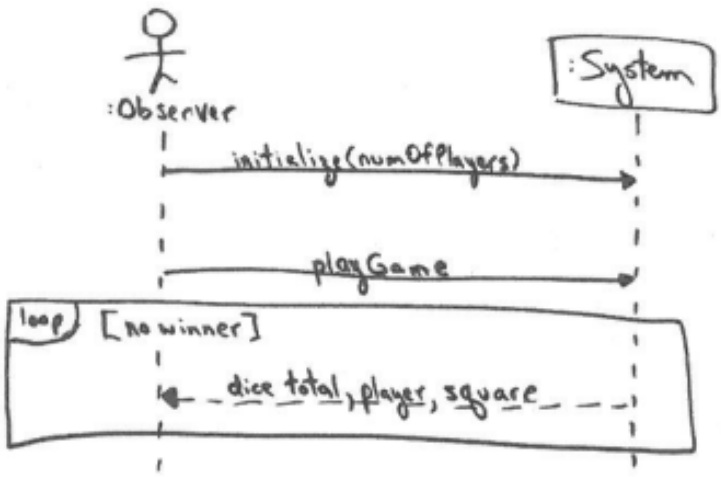


Figure 17.8 SSD for the Monopoly game. Note the *playGame* operation.

Figure 4.4: The SSD for playGame() from Larman 10.

Note the connection between the SSD and this part of the design
Larman 10 is keen to point out the importance of the SSD here, later he explains that the interfacing system may not be a user, but an interface to a device (or group of devices,) a network port or some other outward facing entity.
Next lets look back at the object domain model to see what classes we have in addition to square (see 4.5).

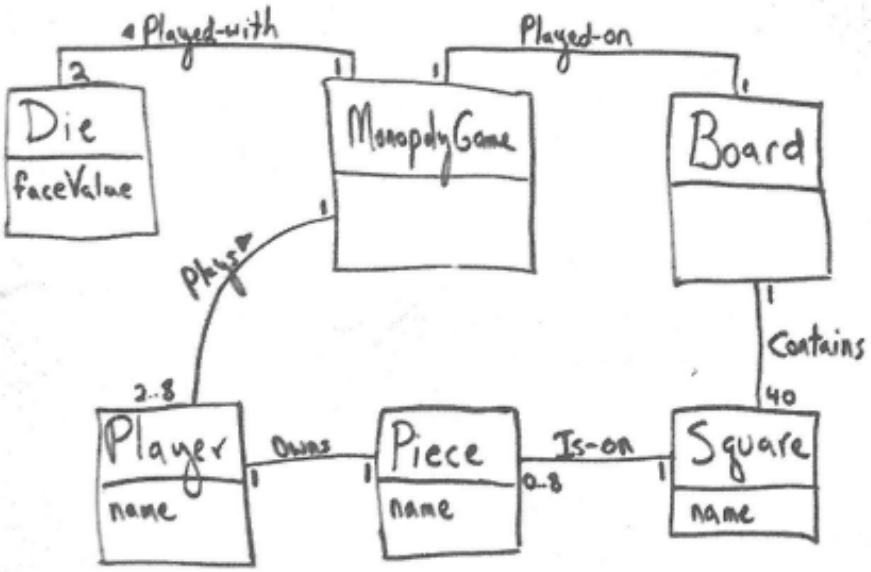


Figure 4.5: The domain model for the first elaboration iteration from Larman 10.

Its kind of obvious we're going to use the Controller Pattern here, (following the pattern from the other sections,) so lets look at the that first (4.2). We'll then evaluate possible implementation of its advice and look at some examples Larman uses to show this being applied during agile modelling.

Pattern Name:	**Controller**
Problem:	Which first object beyond the UI layer receives and coordinates (or controls) the a system operation?
Solution: (advice)	Responsibility should be assigned to an object representing one of the following: 1. An object representing the overall "system" or "root object" (this is a variation of the *facade controller*) 2. A major sub system (another variation of the *facade controller*) 3. A device the software is running within(yet another variation of the *facade controller*) 4. A use case scenario within the system within which the operation occurs (a use case or session controller).

Table 4.5: The Controller Pattern Larman 10 using the less detailed format.

As usuall there are several options according to Larman 10, I've put his discussion of each into a tabular form to aid clarity (see 4.6). One marked difference here is that the choice of object may include the creation of a new object (or objects,) this is one of those were first look at the options controller suggests then a principle (High Cohesion 4.3.2) with which to evaluated the options.

Class Name:	Patterns or Principles	Evaluation
MonopolyGame	• Controller see 4.5 • High Cohesion (another pattern see 4.3.2)	• This is a “system” or ”root object” as suggested. • If the controller becomes to big we violate this see 4.3.2.
A device	Controller	This won’t work - we don’t have a device. If this was running on a phone or a cash machine we could use *Phone* or *BankCashMachine*.
Play Monopoly Game	Controller	This is an option, Play Monopoly Game is the use case that the playGame operation game from. When this is used we normally append “...Handler” or “...Session” to the name.

Table 4.6: Evaulation of the candidates for selecting a controller Larman 10.

Larman 10 cautiously accepts the first of these (MonopolyGame) however he notes that there are trade-offs with the patterns, in this case if the controller dose to much we get a bloated controller. We’re going to consider the patterns in more detail over the next few weeks including the issues to consider when deploying them.

Software Objects and the Design Class Diagram - DCD .

Again we’re working with software objects in the Design Class Diagram (DCD) but we’re off the starting line and have some objects already in our model, so the process changes slightly.

This is the adapted from that described in Larman 10, using the same format as we’ve used earl

Step	In this example.
1. Identify and describe the problem:	*What is the first object after (or beyond) the UI layer should receive the message from the UI layer?*
2. Decide on a pattern to solve the problem:	*We've chosen* ***Controller***
4. Review the SSD, consider the message coming from the UI layer as a system operation (see 4.6)	
5. Look for existing software objects to satisfy the roles of Controller: • Look for existing software objects in our DCD. • If just starting of or if the above fails, using LRG (lower representational gap) identify appropriate domain classes (from the domain model,) and promote them to software classes. Add these to the DCD. • If none of the above fulfil the role we may consider adding a class (this is Pure Fabrication another GRASP pattern **??**).	*MonopolyGame is added as our Controller, this is reasonably only if it ends up with only a few system operations (i.e. doesn't violate High Cohesion)*

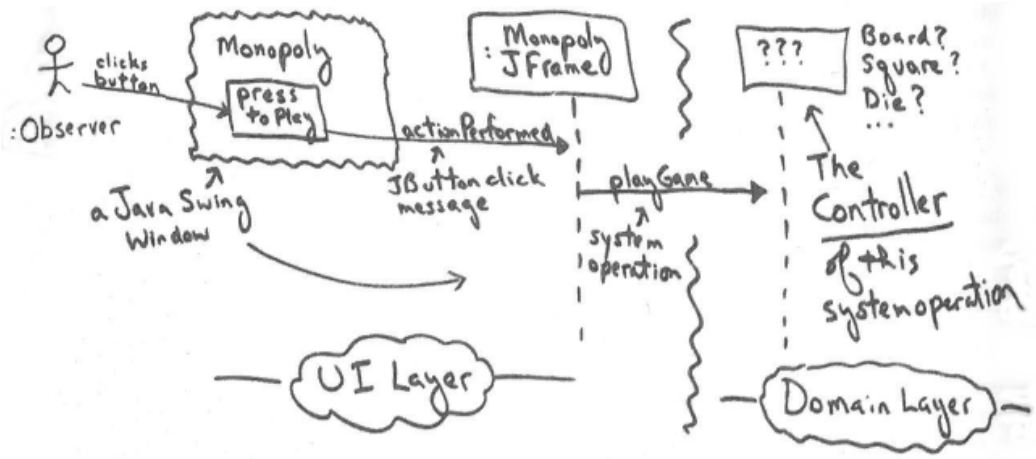


Figure 17.9 Who is the Controller for the *playGame* system operation?

Figure 4.6: Agile Modelling in Action, I like to think this is the moment someone said "who should be the controller?" during the agile modelling session. Actually its taken from Larman 10 discussion so its probably just illustrative.

We now have to consider the relation between these two objects, we add the class to our DCD, we can't connect it up just yet as we need to think about what playGame involves and how it impacts on other objects. Agile modelling suggests that both the dynamic (behavioural - sequence,) and static (structural - DCD) should be worked on in parallel [10]. Larmans image of the completed controller, is a kind of after image of the earlier one 4.7.

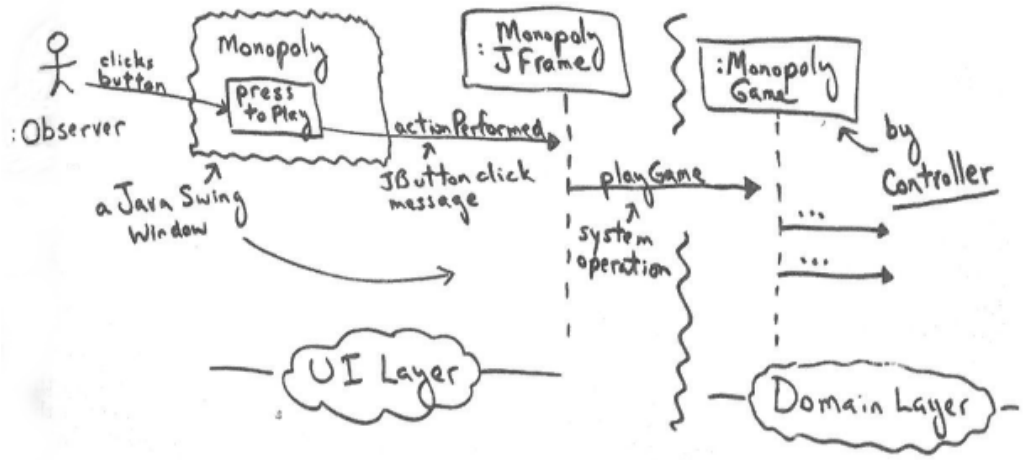


Figure 17.10 Applying the Controller pattern—using *MonopolyGame*. Connecting the UI layer to the domain layer of software objects.

Figure 4.7: This is the controller pattern being applied (taken from [10].

NOTE: This is only a partial scenario, the next step is to work out what objects need to do what. We're not going to go into this in any detail but we will consider High Cohesion a principle which helps us to evaluate our choice (and in part one we've just made).

High Cohesion

The next step is to consider how the domain objects are directed will follow in a few classes time, but one of the key principles we can consider now [10]. This is something we've considered before in the first year - Cohesion. Lets review the problem and possible solutions, we're looking produce the mechanistic design for playGame (that is how do the domain classes collaborate to throw the dice, move the piece etc.) Larman 10 summarises this as a part-complete sequence diagram 4.8.

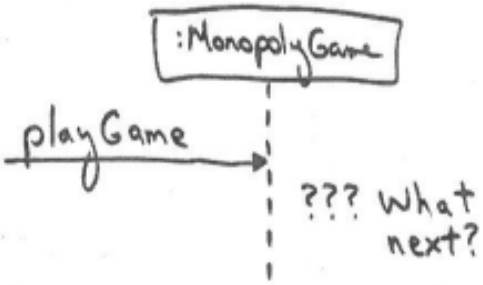


Figure 4.8: Agile modelling - what do we do next?! (taken from [10].

The possible solutions are again show as partial sequence diagrams (see 4.9) but these differ in the level of cohesion.

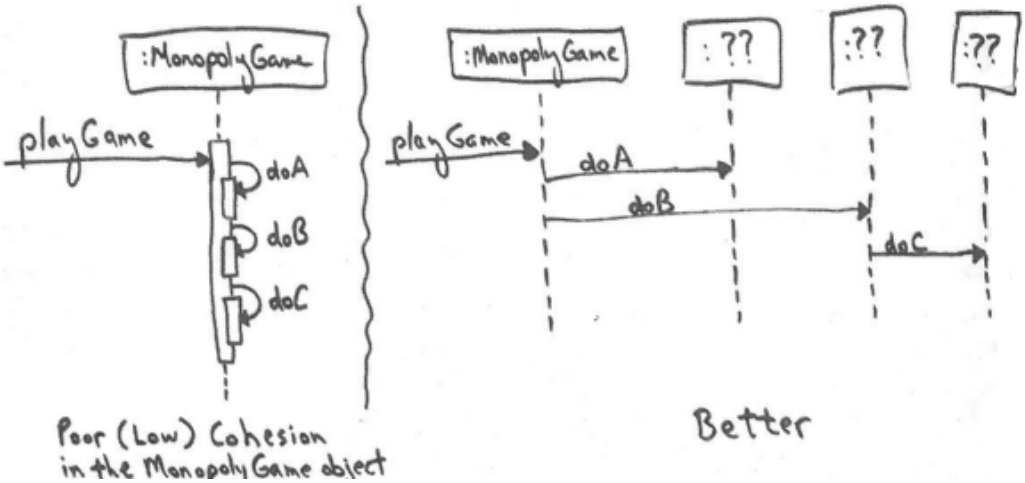


Figure 4.9: High and low coupling and example taken from [10].

Cohesion provides an informal measure of how functionally related operations of a software element are, in this case we're talking about the methods (which fulfil responsibilities) of an object [10]. It also measures the amount of work done by a software element [10]. The book goes for a software based example to begin with but I'll start with an analogy, a reasonably size builders merchants has around 30 staff spread over a head office and several yards. A member of staff Mr. A is responsible for working on the counter at Yard B, organising the payroll for the whole company, completing the tax return and

Health and Safety at yard C. This is not a cohesive set of responsibilities (and hopefully the various issues with this are obvious). So lets re-organise the company, so we have staff at each yard responsible for counter sales, stock control, deliveries and Health and Safety only, in the office we have staff responsible for payroll, finance, etc. Mr C. the manager is responsible for overseeing the office staff, finance, payroll, deliveries, sales, stock control ... in one morning he may authorise the purchase of some bricks from a stockist for Yard A, the transfer of some paint from Yard A to Yard B, a repair to a truck at Yard C, annual leave for a member of the sales team at Yard A etc. Again this is Clearly this is also not quite right, we'd expect to see a layer of management between the overall manager Mr. C and the front line staff. Perhaps a yard manager who took responsibility for the day to day running of the yard?

So the overriding principle here is that an object shouldn't do unrelated jobs and should not do too many jobs (i.e. we should delegate to other objects if there are a large number of related responsibilities).

Going back to the example 4.9 we can see that the right hand version is much more cohesive [10] despite not know what A, B and C are in this context it doesn't matter. If A, B and C relate to different responsibilities its cohesion is low, if they are all related to one responsibility and its just big or complex this is still low cohesion and we need to divide it up. Going a step further if doA() etc. all require at least two collaborating classes (A1 and A2 say) then the left hand version is highly coupled. This leads us to another key point High Cohesion and Low Coupling often go together [10].

Finally we introduce the principle, (and this is more a principle than a pattern,) High Cohesion 4.7 which can be used to evaluate solutions to find the best alternative.

Pattern Name:	**High Cohesion**
Problem:	How to keep objects focused, understandable, manageable and as a side effect support Low Coupling?
Solution: (advice)	Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

Table 4.7: The High Cohesion principle Larman 10 using the less detailed format.

Note: Looking back a little we notice that our use of the controller pattern may cause low cohesion, if we try to use the same controller for everything this is called bloated controller [10].

4.4 More Class Diagrams

4.4.1 Dependency

Associations (and their specialised forms - aggregation and composition,) indicate that one object has an attribute of a type which is another object [13]. Dependency is used to indicate that while the object does not have an attribute whose type is another object, it uses one in some other way [13]. The dependency arrow in UML is an open headed (crows foot,) arrow with a dotted line which may have a stereotype to indicate the nature of the dependency [13].

This shows a kind of coupling between classes [13], is useful when we're analysing solutions for Low Coupling!

Guideline: When to Show Dependency

Larman 14 suggest that we should only show dependencies in the following circumstances (note UML describes the dependent class as the client and the depended upon as the supplier):

- Accessing a local or global variable of the supplier type, this could mean accessing a static constant of a class in C++ or creating a temporary object inside a class.
- Calling a static method of the supplier type, this could mean accessing a static constant of a class in C++, see example 4.10(A)
- Receiving a parameter of the supplier type, see the example 4.10(B)

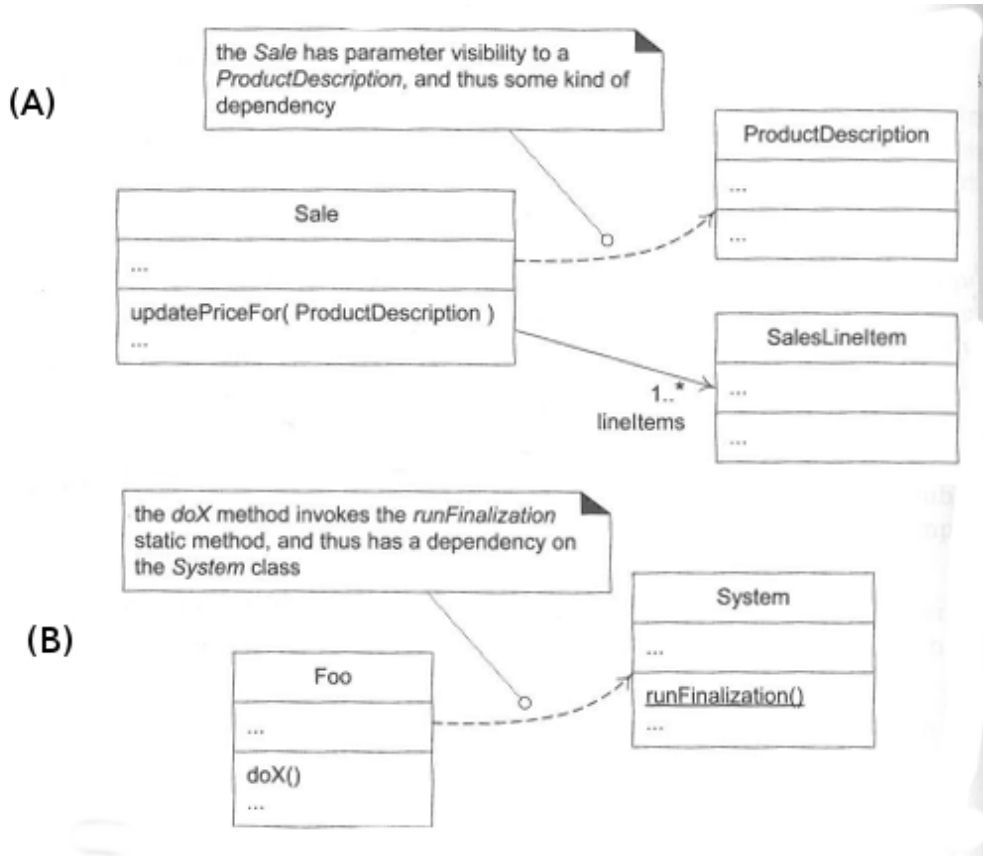


Figure 4.10: Showing dependencies (which are not association/inheritance) in UML class diagrams adapted from[13].

UML provides a number of keywords to help clarify the dependency line (The specification I have (2.5) dates after Larman's book and gives the following) [16]:

- **«instantiate»** - the client creations an instance of the supplier.
- **«use»** - the client uses an instance of the supplier.

4.5 More Sequence Diagrams

4.5.1 Collections and Return Variables

The examples shown above (in the section on Low Coupling4.3) use an alternative syntax for return variables to the one we've used previously (to save whiteboard space). Larman 10 notes the following elements:

- The return variable *sqs* from *getAllSquares* is the same as that named in the 3rd lifeline. This is an object of type Map<Square> a type of map that holds Squares. This referencing of a return variable in the message (sent to it,) rather than as a return is common.
- Similarly the variable *s* in which begins the *getSquare* message and the variable s in the later get message refer to the same object.
- The formatting of the message *s = get(name) : Square* indicates that a Square is returned with the name *s*.

An additional example is show in 4.11.

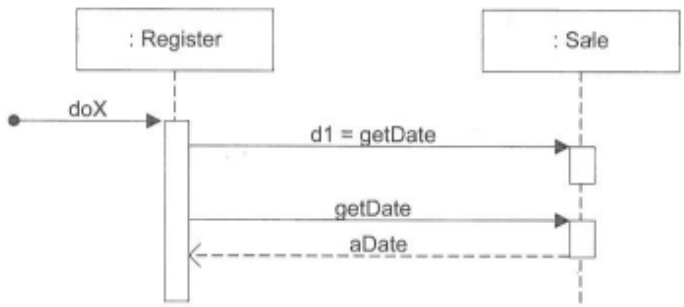


Figure 4.11: Showing return values in sequence diagrams from[14].

4.5.2 Messages to Self

The example for High Cohesion(see 4.9) shows a number of self calls in a sequence diagram, this is used to show an object calling methods from itself (e.g. calling a private helper method). This is shown as a message arrow point back to the originating lifeline with a nested activation bar [14].

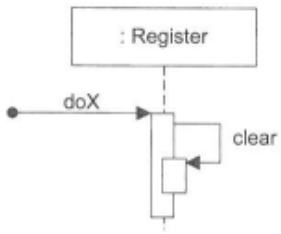


Figure 4.12: Showing self calls in sequence diagrams from[14].

An additional example is show in 4.12.

4.6 Tutorial

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

1. Read the case study 4.6.1, and consider the use case 4.6.1, domain object model 4.14 and SSD 4.15 then use an appropriate pattern to identify:
    - Who is the first object beyond the UI layer who receives the message in the SSD?
    - Who knows how to convert a measure between a given unit and another unit?
    - Who knows how to create a formatted output for a given unit?
2. Read the case study 4.18 see the ODD 4.16 and DCD provided 4.18 and the SSD 4.17 then use an appropriate pattern to identify:
    - Who creates InputPattern?
    - Who creates OutputPattern?
    - Who can find an Input Pattern given an index?
    - Who can find an Output Pattern given an index?
    - Who can load Input and Output Patterns from file, given a filename?

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

- What are the GRASP patterns and why are they useful in the development of object orientated software?

4.6.1 Case Study 1

An application is required to convert between metric and imperial measurements. The application must be capable of performing the conversion from meters and centimetres to feet and inches and vice versa. The application must be written for Windows compatible PC however porting at a later stage is being considered and my include Linux (Gnome based), MacOS, iOS and

Android. The user must be able to enter their measurement in an appropriate format (depending on whether they are converting from imperial to metric or metric to imperial). In response to some input from the user the application should then produce the conversion in a format that allows them to copy and paste this to another application.

The default conversion will be imperial to metric, a switch or toggle will be provided to change to metric to imperial. The interface will consist of a box for each of the units (feet/inches or meters/centimetres) depending on the conversion option selected the application will display appropriate boxes as input and output.

Scenario – Sunny Day Scenario (everything goes right)

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

Scenario – Sunny Day 2

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

Scenario – Formatting error scenario

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

Use Case

The team have produced a number of use case summarised at the highest level by the following use case diagram. The names of the use case here may differ slightly from those we identified during the tutorial.

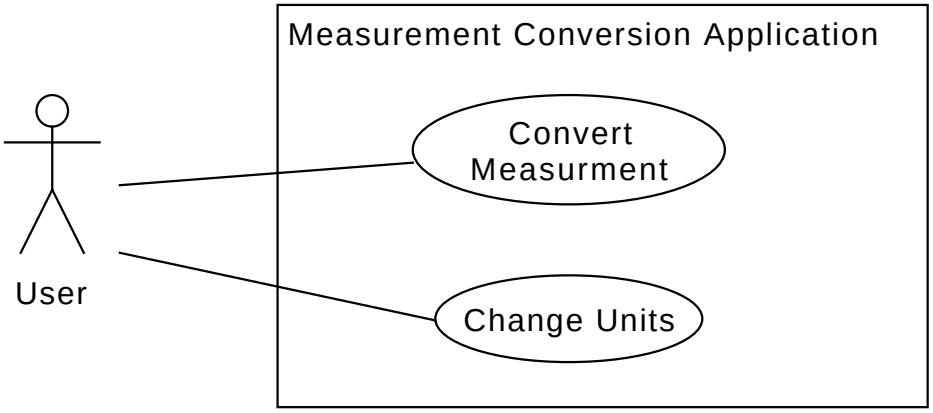


Figure 4.13: This is an incomplete use case based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough.

Use Case: Convert Measurement

UC2: Convert Measurement L3	
Scope:	Measurement Conversion Application
Primary Actor:	User
Level:	user-goal

Stakeholders and Interests:

- User: Wants to convert between imperial and metric measurements of length.

Preconditions:

- Application has launched and the interface is displayed
- Interface is prepared for user input

Postconditions (success guarantee):

- Output is presented in a suitable format

Main Success Scenario:

1. User enters a measurement in the default unit (imperial,)
2. User requests a conversion (indicating they have completed their data entry).
3. System performs the conversion
4. System displays the converted measurement

Extensions (Alternative Flows):

- 1a. User changes units before entering a measurement.
- 2a. User enters nothing
    1. System does not attempt to convert
    2. System highlights the lack of input (e.g. change background of input field to red)
- 2b. User enters an incorrectly formatted measurement for the current conversion.
    1. System Displays an error message
    2. System highlights the incorrect input (e.g. change background of input field to red)
    3. User enters a measurement

Special Requirements:

- Must be possible to enter the measurement in a number of formats, e.g. 1'6", 1.5", 18" etc.
- The output must be such that the user can easily copy the output for use in another application.

Technology and Data Variations List:

1. ...

Frequency of Occurrence:	Expect the user to perform one or more conversions in a single session

Open Issues:

- Given that the input can be in a number of formats, which the system is to deduce. How will this be done?
- The output may be required in a specific format, should this option be given to the user and how? This could be a property set though a menu (with a default,) or some means of selection near the output.

Domain Object Model

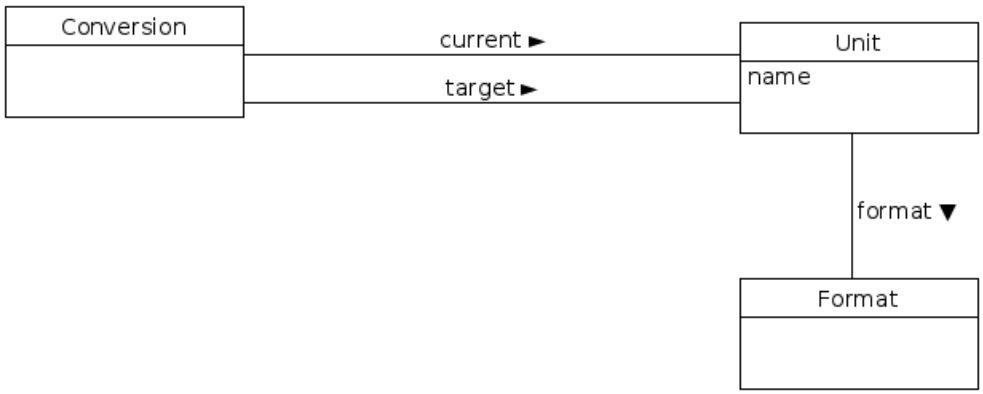


Figure 4.14: This is a domain object model based on the tutorial from a few weeks ago, I'd like to have used the one we drew on the board however the picture I had wasn't clear enough. I've avoided describing this as a draft (suggesting its incomplete,) or using any other term which would suggest it is final. In the spirit of Agile Modelling we are using it to inform sketches of the design so we can discuss and evaluate design options. Changes may be made as a result of this discussion and its probably that future iterations would contain some of the conceptual classes.

System Sequence Diagram

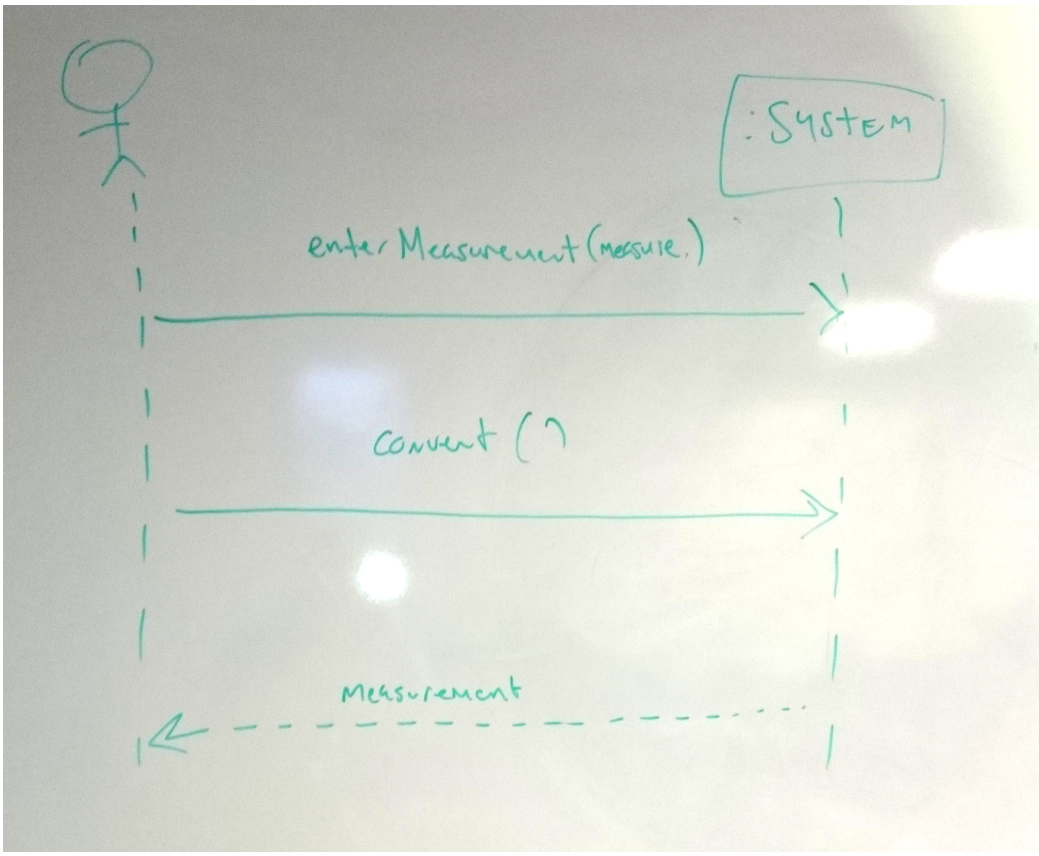


Figure 4.15: This is the one we did in class, I'm not sure if its going to be usable!

4.6.2 Case Study 2

A researcher at the university is looking to write a paper with a colleague they have both datasets and a new neural network to train on them. In order to do this they need to load in files from the original project so that their Neural Network will be trained and tested with the same files. Each file consists of comments (which start with a #,) some meta data (information about the file content) and the data itself (see example file), all in a CSV format. The data is in the form of input and output patterns, each contains t (a time offset). The input contains the values for parameters t, r, g, and b while the output patter has two values t and o. Due to the nature of neural network operations all parameters are floating point numbers between -1.0f

and 1.0f further validation is possible as the t (which represent the time-step,) should be identical in the input and output pattern.

4.6.3 Domain Object Diagram for Case Study 2

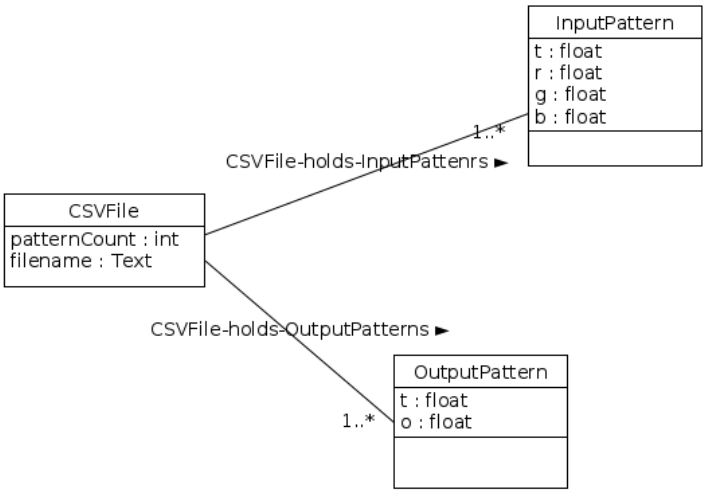


Figure 4.16: Domain Object Model for Case Study 2.

4.6.4 System Sequence Diagram for Case Study 2

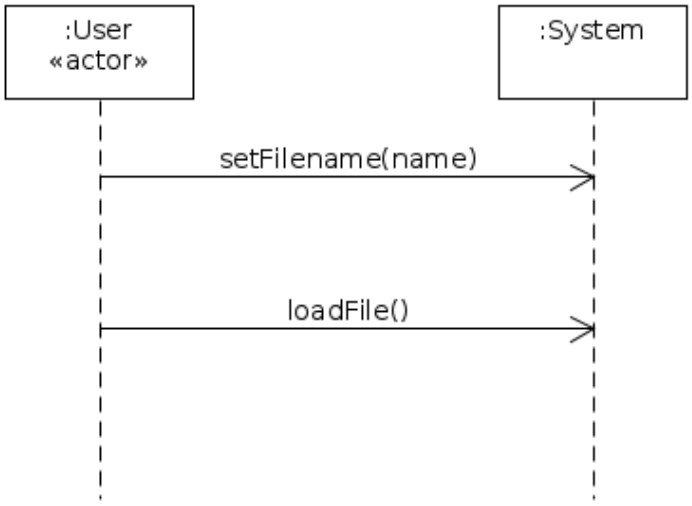


Figure 4.17: SSD for Case Study 2.

DCD for Case Study 2

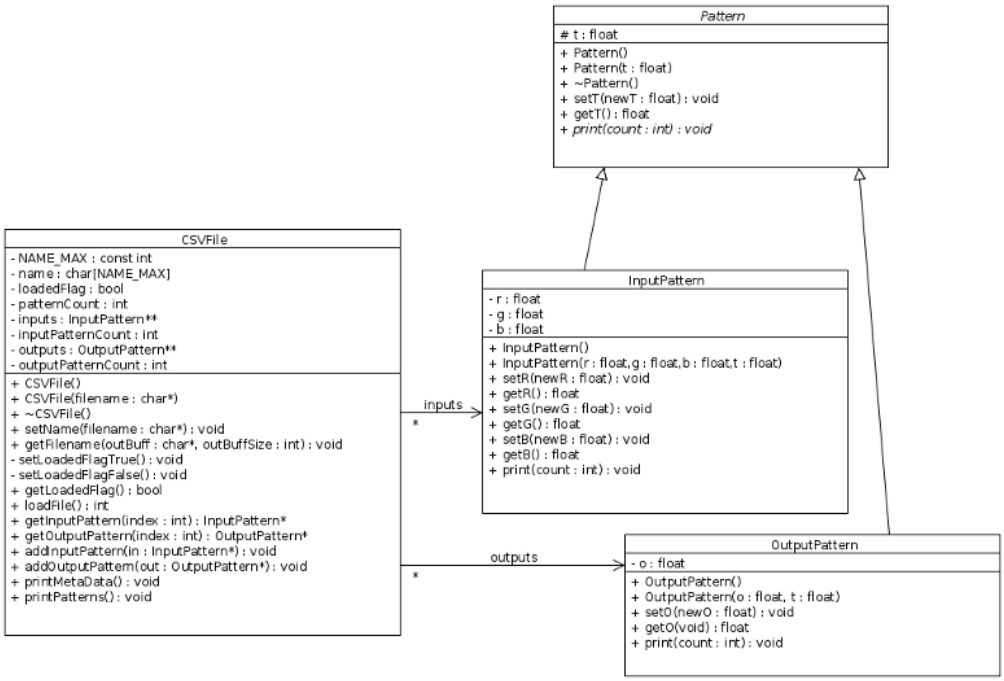


Figure 4.18: I happen to have this as I set is as a coursework, to end up with this we'll need one pattern I've not covered yet!.

4.7 Summary

The next important part of the design process is the Realisation of the use case, which is the mapping of use case and individual scenarios to object behaviour. We've started small considering parts of these interactions in isolation and the low level patterns (or OO principles,) we've looked the most essential of the patterns (including creator we did last week). There are only a few more to do, but we're going to use these to realise some use case these before looking at them.

The GRASP patterns are low level simple design patterns/principles named and documented to make them easier to remember, discuss and use. This week we've looked at Information Expert, Low Coupling, Controller and High Cohesion with some examples of their application, however we've not considered the pattern in full (which we'll do in a later class) at the moment we

need to see a few more patterns so we can start using them in the realisation of use case.

4.8 Preparation for Next Week

4.8.1 Essential Reading

An introduction to GRASP and Patterns, this covers this week and last weeks material..
Larman (2014d). *Applying UML and Patterns*, chapter GRASP: Designing Objects with Responsiblities, pages 312–362. Prentice Hall PTR.
For next week read as much of chapter 18 as you can! I'll not be going into as much detail (or doing as many examples as Larman)!
Larman (2014i). *Applying UML and Patterns*, chapter Object Design Examples with GRASP, pages 312–362. Prentice Hall PTR.

4.8.2 Background Reading

In conjunction with the above you may need to refer to these chapters which provide very detailed coverage of Class and Sequence diagrams. Larman (2014h). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

Larman (2014g). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

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

Bibliography

[1] Douglass, B. P. (2004a). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Requirements Anlysis of Real-time Systems, pages 331–399. Object technology. Addison Wesley, 3rd edition.

[2] Douglass, B. P. (2004b). *Real Time UML: Advances in the UML for Real-Time Systems*, chapter Object Orientation with UML 2 - Behavioural Elements, pages 139–215. Object technology. Addison Wesley, 3rd edition.

[3] Fowler, M. (2003a). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Introduction, pages 14–26. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[4] Fowler, M. (2003b). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Class Diagrams, pages 79–85. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[5] Fowler, M. (2003c). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Class Diagrams: Advanced Concepts, pages 56–72. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[6] Fowler, M. (2003d). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, chapter Sequence Diagrams, pages 47–56. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

[7] Larman (2014a). *Applying UML and Patterns*, chapter Domain Models, pages 131–172. Prentice Hall PTR.

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

[9] Larman (2014c). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 173–181. Prentice Hall PTR.

[10] Larman (2014d). *Applying UML and Patterns*, chapter GRASP: Designing Objects with Responsiblities, pages 312–362. Prentice Hall PTR.

[11] Larman (2014e). *Applying UML and Patterns*, chapter On To Object Design, pages 213–220. Prentice Hall PTR.

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

[13] Larman (2014g). *Applying UML and Patterns*, chapter UML Class Diagrams, pages 249–270. Prentice Hall PTR.

[14] Larman (2014h). *Applying UML and Patterns*, chapter System Sequence Diagrams, pages 221–248. Prentice Hall PTR.

[15] Larman (2014i). *Applying UML and Patterns*, chapter Object Design Examples with GRASP, pages 312–362. Prentice Hall PTR.

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