Ariadne training

UML Applied – Second Edition

Object Oriented Analysis and Design

www.ariadnetraining.co.uk

About Ariadne

Ariadne Training Limited was established in September 2000 and since then we have built a reputation based on our excellent courses, our responsiveness to our customer's needs and our "in the field" experience.

Our Courses

Ariadne's courses are designed to be fun to attend, whilst at the same time covering the topics in depth. During the course, you will have plenty of time to practice and apply your new found skills.

The courses are all written in-house, by experienced IT professionals with genuine "in the field" experience of their subject. We also ensure that our trainers are first class teachers, and able to explain the most difficult concepts clearly. Above all, your trainer will work hard to make the course enjoyable and productive.

Ariadne's Mentoring Scheme

For many of our customers, what really matters is: "can I really apply what I have learnt to my real work?".

Part of our solution is to ensure that as many of our courses as possible are backed up with a **Mentoring Scheme**, as an integral part of the course. The scheme means that after you have attended an Ariadne course, your project are entitled to an equal number of days of free mentoring, where the trainer will work with your delegates to help them apply their knowledge. Exactly how the trainer does this is up to you, but examples include:

- The trainer works with you to kick-start a project, such as helping define your requirements or creating an initial architecture.
- Leading a team of (for example) programmers and helping them through their first real attempt at applying the work

The goal of the mentor is to get you, as a project, to the point where you can "fly solo", and their services are no longer required!

Ongoing Backup

After the course, every attendee is also entitled to unlimited support from their trainer, via direct contact by email and telephone. Although we can't promise to answer queries immediately, we do promise that we will always be there to help out once you're back "in the real world".

Ariadne's Case Studies

On an Ariadne course, you will spend half of your time following an in-depth **Case Study**, a major practical exercise designed to simulate the kinds of problems you are likely to face when you return to work.

Why the name 'Ariadne'?

We get this question on every course! Ariadne is the name of an ancient Greek princess, who guided her lover Theseus out of the labyrinth and the dangers of the Minotaur with a ball of thread. This labyrinth can still be visited today in the beautiful Knossos Palace in Iraklion on the Mediterranean island of Crete.

Our Mission

We are dedicated to bringing understanding to the techniques used by the software industry - the simplicity of which is so often shrouded by a fog of detail.

In short, we aim to show that software engineering can be very simple.

Introduction

This book is provided as a companion to the Ariadne training course *UML Applied – Object Oriented Analysis and Design*. Although some of the book is written in standard book format and can be read without attending the course itself, readers using the book without attending the course may find some details of the course are omitted, or some aspects are not explained in depth. Usually this will be because the lecturer covers some material not covered in the book.

In particular, the **case study**, which makes up 50% of the course, is not mentioned in this text (we often rotate our case studies, so it would not be possible to do so).

References to other books are denoted by a number in square brackets – for example, [4] means to check reference 4 in the bibliography at the back of this book.

Intended Audience

There are no specific prerequisites to attend the UML Applied course.

Experienced Analysts who are looking for information about the UML will find the accompanying course *Business Systems Analysis with the UML* to be more useful.

The course itself is hands-on and fairly intense, so it essential that all attendees take an active and enthusiastic part in the case study.

To submit comments or questions, please email info@ariadnetraining.co.uk, or see our website at www.ariadnetraining.co.uk.

The Cover Art

The cover art by Laura Morgan was one of the first she did for Ariadne. The members of the orchestra are playing different instruments, and yet are all reading their instructions written in the same language. The language might look a bit archaic to non-musicians, but for the musicians, it is a *standard* language. This is the aim of the UML.

© 2000 Laura Morgan

Running Order

Detailed Contents

Each heading listed here coresponds to a screen used during presentations on the training course.

Chapter 1

Introducing the UML

In this session, we will describe the **UML**, or the **Unified Modelling Language**. As well as charting the history and development of the UML, we will explain what the UML tries to do – and perhaps more importantly, we will explain what the UML does **not** do.

What is the UML?

"UML" stands for "Unified Modelling Language", and is designed to be a graphical language. The UML user guide claims that the UML is designed for "specifying, visualising, constructing and documenting the artifacts[1] of software systems". A rather grandiose definition – what does it really mean?

In everyday English, an "Artefact" is a product of human workmanship. So in the context of building software systems, an Artefact will be the pieces of work that we generate in the course of building the system. Think of the diagrams, documents, test plans, code and so on.

So, to rephrase the definition from the UML User Guide:

"The UML is a graphical language that enables us to write down on 'paper' the work that we produce in the course of building a software system."

We don't necessarily mean 'on paper'; in fact the UML is designed to be electronic as well as easy for humans to sketch on paper (or on the back of cigarette packets).

What do we mean by a "graphical language"?

[1] The user guide uses the American spelling. We'll use English from now on.

A Graphical Language

Consider a normal, everyday language (not a programming language, but a language such as English or French). The language will consist of:

- A notation; the symbols we can use to write down the language
- A syntax and grammar; a collection of rules that determine what we can and cannot do with the notation to produce meaningful results

The UML is no different. It contains a collection of symbols we can use to write down the language. The difference between a natural language like English and the UML is that the UML's symbols are graphical; rather like icons. Here are some examples of the UML's notation:

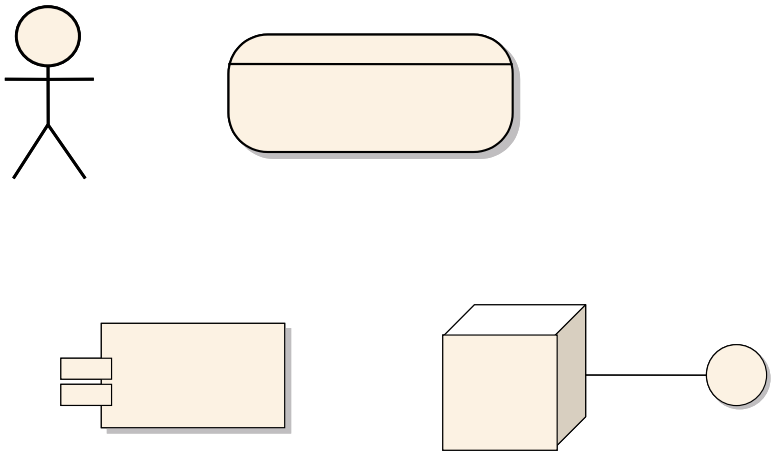


Figure 1 - Some UML Symbols

(Don't worry about what these symbols mean for now – we'll be explaining them in detail (if we need them) as we progress through the course).

The UML 'Grammar'

The symbols on their own are not a great deal of use; we have to connect them together to construct meaningful statements from them (in the same way as connecting letters together to make meaningful sentences).

Here is an example of three UML symbols connected together – a stick man, an oval and a line joining the two together:

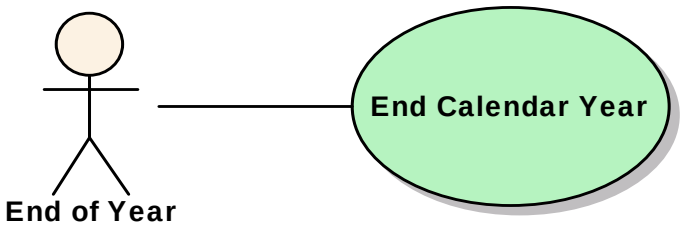


Figure 2 - UML Symbols joined together

Does the picture in ke any sense? If so, what does it make any sense? If so, what does it mean?

Learning the UML symbols is easy; we could sketch them down in a couple of minutes and explain most of them in an hour or so. This is no different to learning an alphabet – all very well but you can't read or write a book after learning the alphabet.

So in order to understand the UML, it is necessary to learn its grammar; in other words, how to put the symbols together…

Building Floorplan

To digress for a second, have a look at the figure below – an extract from an architectural floor plan. What does the symbol inside the oval represent?

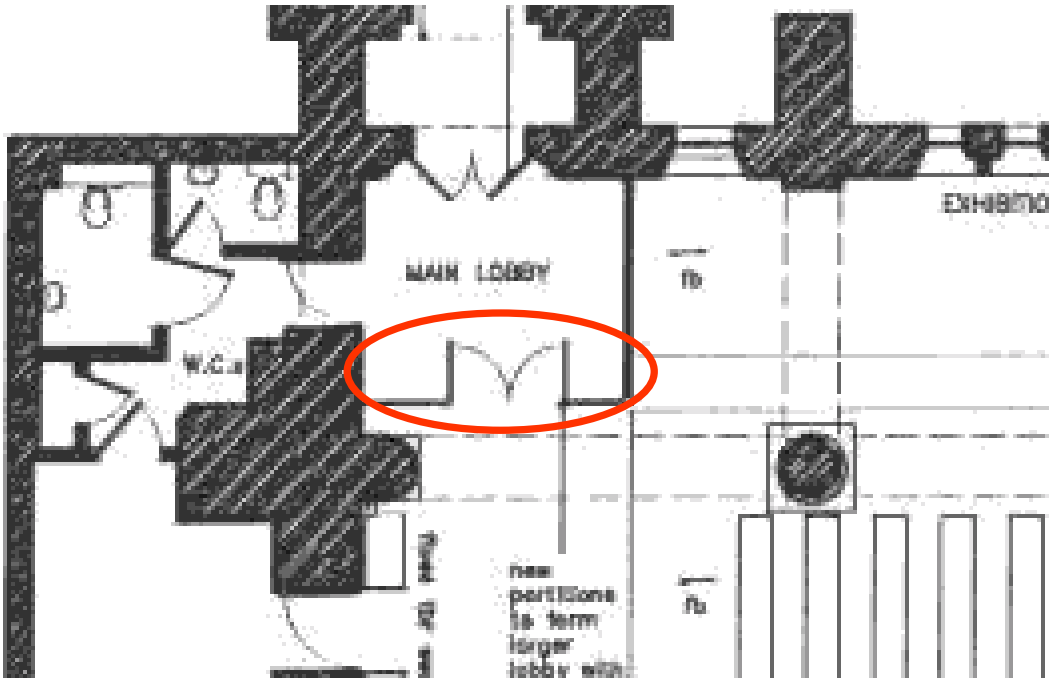


Figure 3 - Extract from an architectural floor plan

Full marks if you recognised the symbol as a door (or in this case, double doors). Most readers of this book won't be au fait with architectural plans, yet the symbol is so simple, it is easy to recognise what it is representing. The UML has similar aims…

Modelling Notations

If we were a building construction professional, we would not consider performing a task without using a standard notation to create plans. Ideally the plans would need to be understood by the architect, the builders, the planners and even (with a little help, and at a fairly high level) by the future owners. Although there is only one house to be built, there may be many different versions of the plan representing different levels of detail – the electrical contractors would require a very different (and more detailed) plan than the future owners.

The IT Industry has, until recently, lacked such a standard notation. It was impossible to “draw up the plans” for an IT System in any standard manner – any project had a bewildering choice of conflicting modelling notations…

The “Method Wars”

Between 1989 and 1994, there were more than fifty modelling languages in *common* use – goodness knows how many more there were. This period was known as the *Method Wars* (perhaps *Notation Wars* would have been a more accurate term).

Any project starting up in this period would be faced with a daunting choice – which one to adopt?

Which one was the best? Well, none of them really. They all had strengths and weaknesses; they all did pretty much the same thing, but probably all in slightly different ways.

As an example, here is an item of notation from the “Booch” syntax, developed by industry Guru and one of the original members of Rational Software, Grady Booch:

Figure 4 - The Booch "Cloud" Icon

If you have never worked in Booch before, the “Cloud” icon will be meaningless to you. In fact, it merely represents a concept that is present in *all* modelling languages, but using a very strange and bizarrely shaped icon.

Anyway, through the mid 1990’s, three of the notations began to gain strong industry acceptance.

- Booch, the notation developed by Grady Booch, as mentioned above.
- OMT, the “Object Modelling Technique”, developed by James Rumbaugh during his work with General Electric
- OOSE, or “Object Oriented Software Engineering”, a technique developed by Ivar Jacobson at Eriksson.

There were other methods and notations, such as Jackson (JSD), SSADM and Shlaer-Mellor, but the three notations listed above were to become the most significant…

The Three Amigos

In 1994, James Rumbaugh moved to Rational to join Grady Booch, and to begin an effort towards combining their ideas into a single “method”, which they originally called the “Unified Method”. In 1995, Ivar Jacobson joined the unification team, adding the ideas behind his OOSE method into the mix. As we’ll see later, one of the key ideas in OOSE was a *Use Case*, which is now the cornerstone of the modern day UML.

The Object Management Group, or OMG[2] are an industry standards body, run as a not-for-profit organization. Concerned at the lack of an industry standard modelling language, the OMG issued a challenge to the software industry to develop one. The UML was eventually adopted by the OMG as the standard modelling language.

Although Booch, Rumbaugh and Jacobson are credited with the initial development of the UML (and they are certainly the major contributors to the language), in fact the industry in general have contributed to the UML, with many organizations and individuals adding their own ideas.

Although Rational may have been credited with the initial impetus for its creation, the UML is now “owned” by the OMG; it is non-proprietary and is

[2] www.omg.org

available to all. The specification of the UML is available as a free download from the OMG website.

The UML is Simple

Ariadne are guaranteed to get floods of complaints about this statement. It is certainly true that the UML specification is a complex and difficult read – but then the document isn't really intended for use by users of the UML – it is targeted at tool developers or academics.

To put the above assertion into less controversial terms, we mean "the UML aims to allow us to produce uncluttered, focused, clear and understandable models produced at the correct level of abstraction". In a similar way to the floor plan in a door (or in this case, double d, the UML symbols should be recognizable and understandable to all – even to those not connected to IT.

As the project progresses, the designers and programmers may well need to build more detailed diagrams using the UML; but these diagrams will complement the more general and high level diagrams that we are likely to produce in Analysis.

Some people seem to want to make the UML as complex as possible, using the logic that Complex Systems => Complex Method.

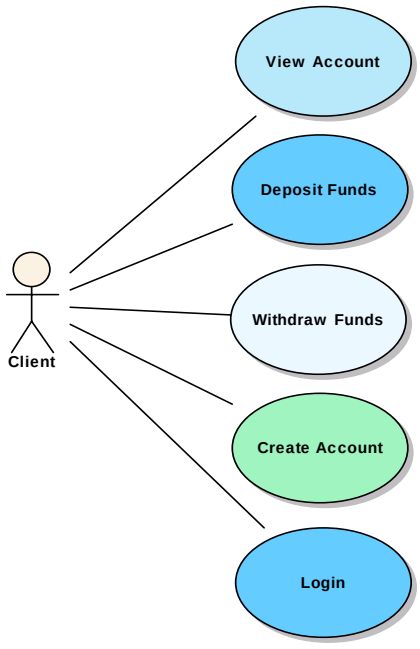


Figure 5 - One of the UML models, shown here almost in its entirety. Don't worry about its meaning – we'll be studying it later, but hopefully it should be obvious that it is very simple and straightforward.

The UML is a communication tool, and we should take care to produce models that are clear and understandable to our target audience.

UML Penetration

All of the above is all very well, but the UML would be useless if the industry had ignored it. In fact, the UML seems to be achieving its goal of becoming the 'lingua franca' of the software industry.

Many tools support the UML as their primary notation, and most of them support the full range of UML models, with Business Process Models, Analysis Models and Architecture models.

Some of the tools also support "Round Trip" engineering, where program code can be generated from the models, and models generated from the code.

Summary

- The UML is a notation for capturing the "artefacts" of software intensive projects
- A graphical language, and one that is aiming to become a "standard" language in the software industry
- The UML is overseen by the OMG; membership is required to have voting rights
- The UML aims to be simple and a versatile toolbox rather than a prescriptive method or process

Chapter 2

The Unified Process

It is very, very important to remember that the UML is a **bare language**. We hope that we have convinced you so far that this isn't a bad thing; the UML aimed to gain industry acceptance as a common language, and it seems to have done so. If the goals of the UML had been broader and had incorporated a process, it would not have stood a chance – every project on the planet does things in different ways, and processes are very contentious[3].

The UML does not tell you how to develop software; it is definitely not a method, a methodology or a process.

Omitting any question of process from the UML is generally a laudable move – but it does leave us with the question of how to apply the UML on our projects.

In fact, although the UML makes no mention of processes, it was designed to be used alongside **Object Oriented** (or **OO**) projects; many of the notations in the UML are heavily influenced by the OO way of thinking. That said, it can be certainly applied to non-OO projects as well.

Similarly, the UML can be used alongside any project lifecycle[4], but the most natural fit for the UML is an *iterative* lifecycle. We'll look at the iterative

[3] A loose definition of process; a method of developing software that rigorously defines the steps to be followed and the tasks to be carried out)

[4] They include Waterfall, V-Model, DSDM, RAD, RUP, Iterative, Incremental, Spiral et al

lifecycle in a few moments, but first we'll look at the most common lifecycle in use today, the *waterfall lifecycle*…

The Waterfall Lifecycle

In the waterfall lifecycle (first proposed by Winston Royce in 1970 – although he didn't call it the waterfall at the time), the project is broken down into a series of discrete steps. These steps vary from project to project, but we would expect to see at least the following on most projects:

- **Analysis;** where the problem is analysed and requirements are gathered
- **Design**; where a solution to meet the requirements is planned
- **Implementation;** where the solution is realised in the form of code
- **Testing;** where the solution is tested against the requirements
- **Deployment**; where the solution is delivered to the client site

These steps are perhaps a little simplistic; on many projects we would expect to see activities such as Business Analysis, high/low level design, unit testing, system testing, acceptance testing, integration and so on.

The lifecycle gets it name because the steps are carried out in strict sequence; it is not acceptable to move on to the next stage until the current stage is 100% complete. Once the project has moved on, it is difficult (or impossible) to move back "up" the lifecycle:

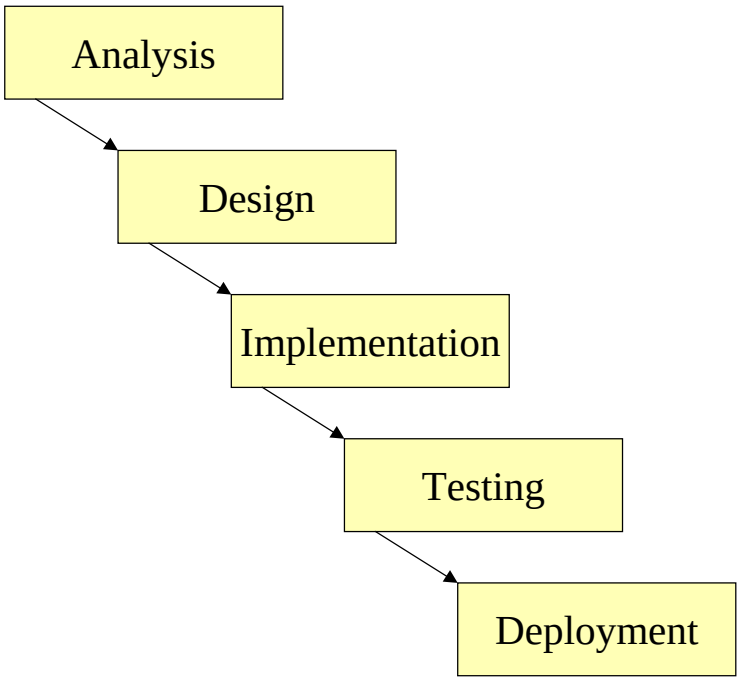


Figure 6 - The Waterfall Lifecycle

Disadvantages of a Waterfall Lifecycle

This simplistic (and easy to manage) process begins to break down as the complexity and size of the project increases. The main problems are:

- Even large systems must be fully understood and analysed before progress can be made to the design stage. The complexity increases, and becomes overwhelming for the developers.
- Risk is pushed forward. Major problems often emerge at the latter stages of the process – especially during system integration. Ironically, the cost to rectify errors increase exponentially as time progresses.
- On large projects, each stage will run for extremely long periods. A two-year long testing stage is not necessarily a good recipe for staff retention!

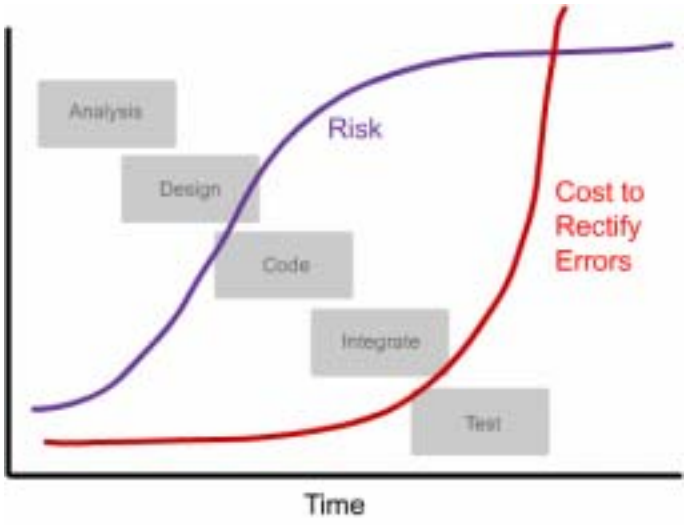


Figure 7 –Over time on the waterfall, both the risks and the cost to rectify errors increase

the wate46.2(r(allmoe46.2odel)5.3(he)5.3(e)-648(a)-1.1syg to u)4.2(nderstu)4.2(a)-1.1de aa aa theadvsthe(m)4.6(o)-6.5dpe-4.1( )-5.7(b)4.1egima r46.2(e43.1(a))1.2sk ne .5(e)-6.5xs

Also, as the analysis phase is performed in a short burst at the outset of the project, we run a serious risk of failing to understand the customer's requirements. Even if we follow a rigid requirements management procedure and sign off requirements with the customer, the chances are that by the end of Design, Coding, Integration and Testing, the final product will not necessarily be what the customer wanted.

Waterfall – Advantages

Having said all the above, there is *nothing wrong* with a waterfall model, providing the project is small enough. The definition of "small enough" is subjective, but essentially, if the project can be tackled by a small team of people, with each person able to understand every aspect of the system, and if the lifecycle is short (a few months), then the waterfall he vsaablepoe ieihbetter( )-5.3(tanh)-7.5(cj)-5.2hohing

The Unified Process (UP)[5]

We'll leave the waterfall model behind for a short while (but bear in mind its advantages and disadvantages). The UML is often applied alongside a lifecycle that is radically different from the waterfall – although is does feature some of the waterfall's ideas.

The Unified Process[6] (or UP) is divided into four major phases: **Inception**; **Elaboration**; **Construction** and **Transition**. These phases are performed in sequence, but the phases must not be confused with the stages in the waterfall lifecycle. This section describes the four phases, and outlines the activities performed during each one.

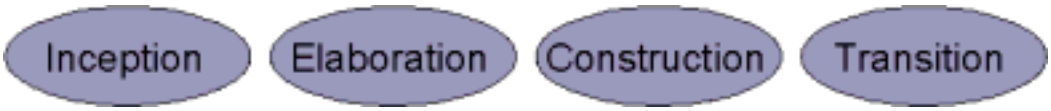


Figure 8 - the four phases of the Unified Process

Inception Phase

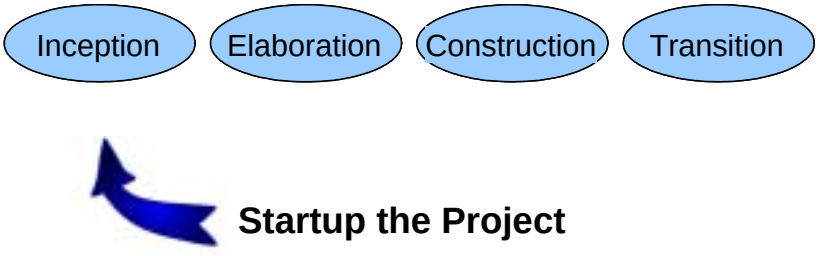


Figure 9 - Inception; the starting phase for the project

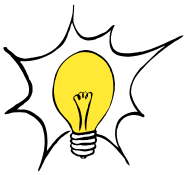
---

[5] See http://www.unifiedprocess.org/ for more information

[6] Development of an industry standard process is some way behind the development of a unified notation. Many of the topics we present here are in a state of flux at the time of writing (after all, developing an industry standard process is a *massive* task). You may see the term *Rational Unified Process*, or RUP mentioned elsewhere – the RUP can be thought of as the UP, with proprietary extensions offered by Rational. Thus, the RUP is a commercial product. A newer process for Enterprise Systems called the EUP (http://www.enterpriseunifiedprocess.info/) is also being constructed; the EUP has exactly the same base as the UP we study here – it merely adds some extra concepts which are useful for enterprise systems development.

The inception phase is concerned with establishing the scope of the project and generally defining a vision for the project. For a small project, this phase could be a simple chat over coffee and an agreement to proceed; on larger projects, a more thorough inception is necessary. Possible deliverables from this phase are:

- A Vision Document
- An initial exploration of the customer's requirements
- A first-cut project glossary (more on this later)
- A Business Case (including success criteria and a financial forecast, estimates of the Return on Investment, etc)
- An initial risk assessment
- A project plan



Inception in a sentence: Determine what needs to be done, examine the ways it could be achieved and decide whether it makes business sense to go ahead

Elaboration

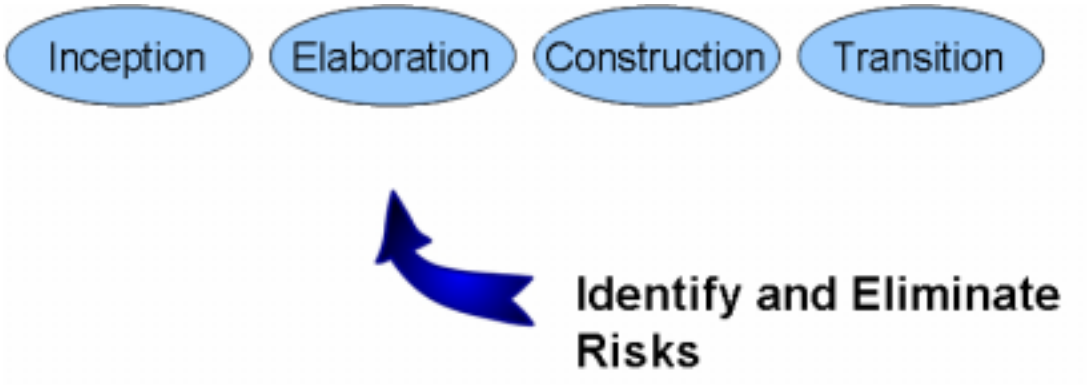


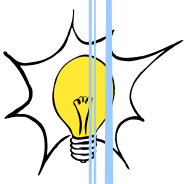
Figure 10 - The general goal of the elaboration phase is to identify and eliminate risks

The purpose of elaboration is to analyse the problem, develop the project plan further, and eliminate the riskier areas of the project. By the end of the elaboration phase, we aim to have a general understanding of the entire project, even if it is not necessarily a *deep* understanding (that comes later, and in small, manageable chunks).

You may recognise that this phase sounds very mu(oje)-6nvr [(p)-7.7rvona(e)-1.2(r)-Tona(e)6.4(t) 2r

different tasks and activities (and we'll see what these are in detail shortly). One of the activities is indeed Analysis, but it is certainly not the only job to do in elaboration.

So we'll need to brea



Transition Phas

Figure 12 - In Transition, we make the client take ownership of the finished work

The final phase is concerned with moving the final product across to the customers. Typical activities in this phase include:

- Beta-releases for testing by the user community
- Factory testing, or running the product in parallel with the legacy system that the product is replacing
- Data takeon (ie converting existing databases across to new formats, importing data, etc)
- Training the new users
- Marketing, Distribution and Sales

The Transition phase should not be confused with the traditional test phase at the end of the waterfall model. At the start of Transition, a full, tested and running product should be available for the users. As listed above, some projects may require a beta-test stage, but the product should be pretty much complete before this phase happens.

Transition: Hand the product over to the customer

Iterative Phases

Any of the phases, but in particular Construction, can (and should) be carried out as a series of “mini waterfalls”. The idea is that we take the work that needs to be done and break it down into a series of separate “chunks”, and build each “chunk” in turn.

At the end of Iteration 1, the first chunk of the system will be complete. It won’t be very much of the system, but it probably will be running and stable code (albeit with lots of “to-dos”).

How we define these “chunks” is critical and difficult; thankfully the UML provides a tool that provides an ideal way of defining the sizes of the chunks; we’ll be studying this shortly.

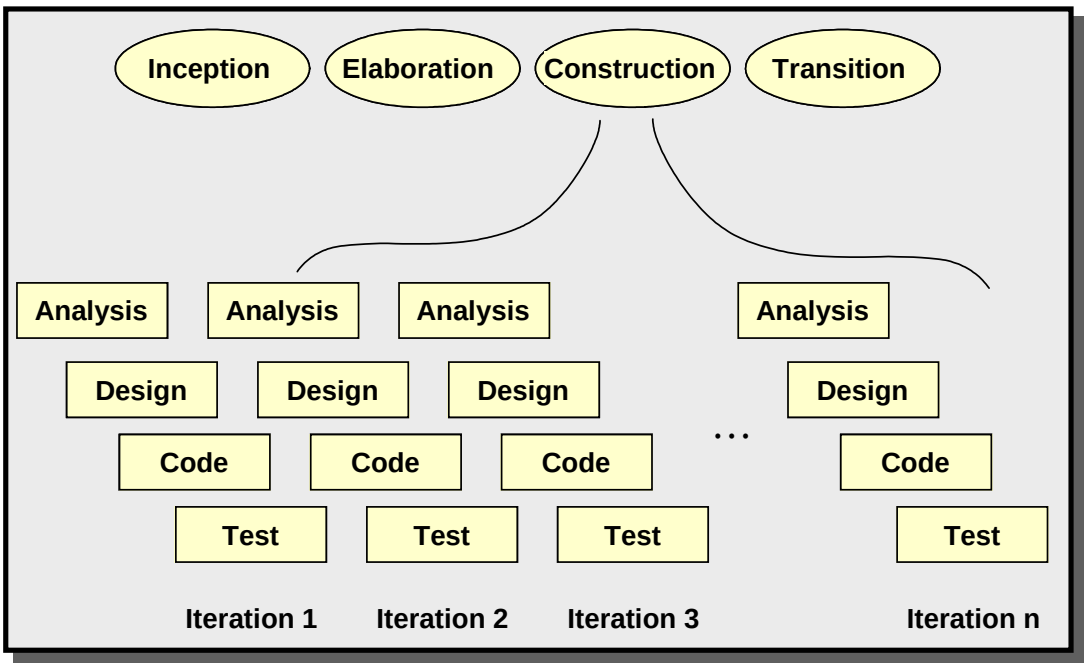


Figure 13 - The Construction phase is performed in an iterative fashion

Typical Timings (eg 2 year project)

How long should each of the four phases last? This is entirely up to individual projects, but a loose guideline is 10% inception, 30% elaboration, 50% construction and 10% transition.

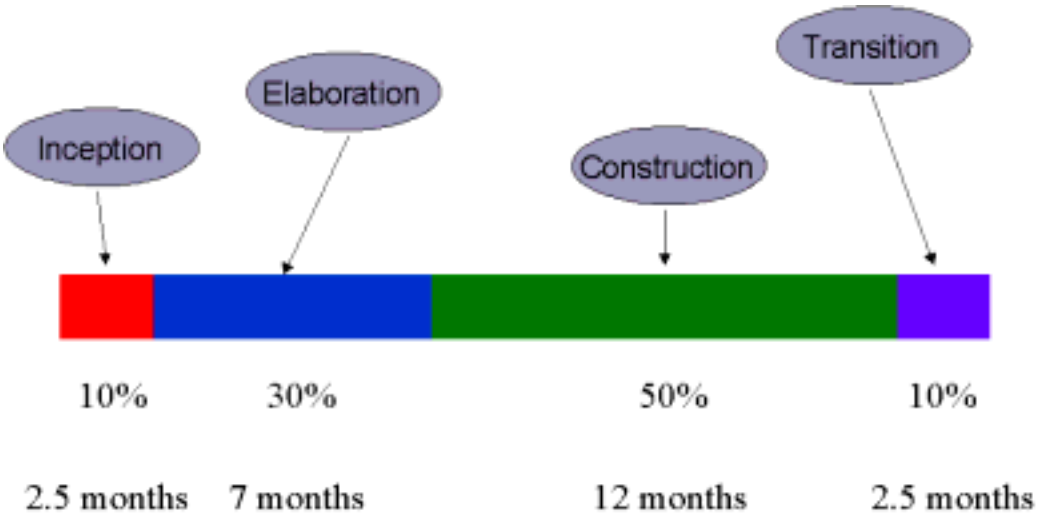


Figure 14 - Possible timings for each phase. This example shows the length of each phase for a two year project.

The timing of 30% for elaboration surprises many people; this effectively means that after 40% of the total development time, construction of the system has not begun.

However, bear in mind that this chart represents the *elapsed time* and not the *expanded effort*. Generally, the inception and elaboration phases are lightly staffed. The Elaboration phase is a specialist job and will require a tight and focussed team of domain experts or senior analysts. By the time elaboration is complete, relatively few “person hours” will have been spent, but the project team (as part of the elaboration deliverables) should have confidence that it is safe to proceed.

The commencement of construction usually sees the staffing profile increase, as separate teams are allocated to different chunks of software.

Project Activities

Let’s stamp out a major and common misconception:

- Inception is **not** Analysis
- Elaboration is **not** Design
- Construction is **not** Coding
- Transition is **not** Testing

All of these phases are in fact a collection of related activities – we have seen for example that construction features analysis, design, coding and testing.

The following diagram is a chart describing the activities that take place through each phase, and how intense that activity is:

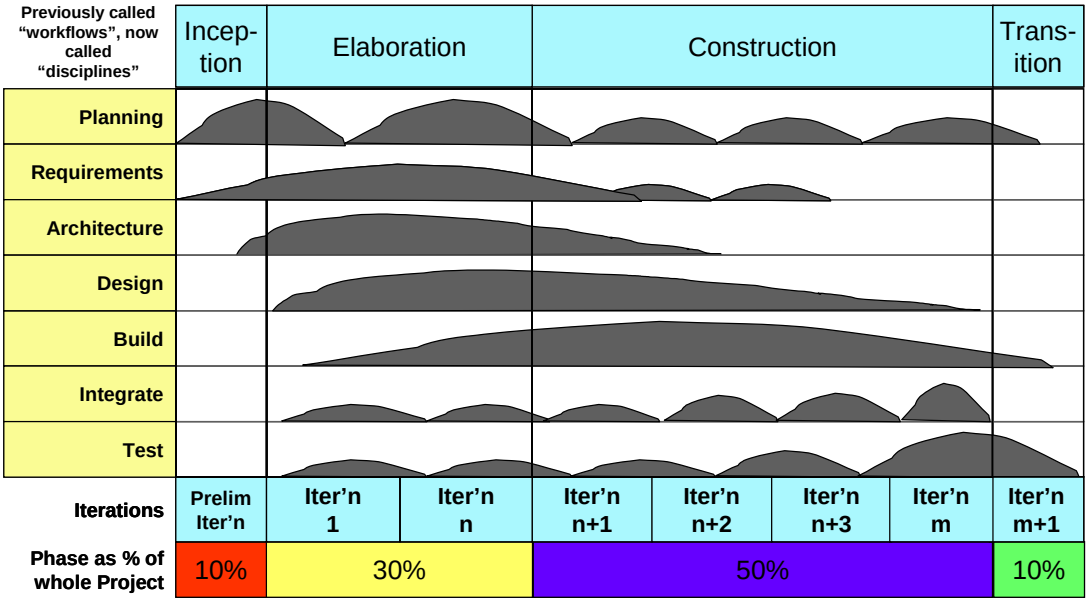


Figure 15 - The activities performed through the lifecycle

The diagram above is just an extract; there are other activities such as "Project Environment" (ie setting up the equipment for the project), and so on.

Let's look at each activity in turn:

- **Planning** is a continuous, on going activity through the entire project. This is in stark contrast to the waterfall set up, where the project would often be planned using Gantt charts and fixed milestones. The project is constantly reviewed – particularly at the end of each iteration, and the progress is checked against the plans.

  Note, however, that the planning effort will get a little less intense as the project progresses; in theory after the first couple of construction iterations, the project should get into a "groove". This is a generalisation though, and the needs of individual projects will vary.

- **Requirements –** the analysis of requirements is an analysis activity that may well be started during inception, but the bulk of the effort is done during elaboration. Due to the iterative nature of the process, some tinkering may need to be done during construction, but these changes should be minor.

- **Architecture** is a term that has many meanings in the IT industry; here we use it in the sense of "the plans for the overall structure of the software". For example, we may decide that the solution will be coded as three distinct "subsystems".

  In the UP, one of the main activities in the Elaboration phase is the development of a sound architecture. The means for developing the architecture is usually though developing **prototypes** that exercise the proposed architecture.

  Ideally, once elaboration is complete, we would have a complete and perfect architecture – meaning that Construction becomes a much simpler process of "slotting in" the code developed in each iteration. This ideal never happens of course, and so the refinement of the architecture continues throughout construction – and will often be done by a specialist "architecture team".

- **Design** begins in elaboration (and will often be the design of the prototypes), and this effort naturally continues through construction. The design effort begins to peter off as time goes on, because the later iterations will tend to be simpler, more trivial blocks of functionality.

- **Building**, similarly commences in Elaboration (the prototypes), and continues throughout construction (building in this context means the same as "coding")

- **Integration** is the process of taking a collection of separate pieces of the system and making them run together. In the Unified Process, integration is a major job, due to the iterative lifecycle – at the end of each iteration, the system has to be integrated.

  Although integration is a big task, and projects following the Unified Process may do more integration throughout the lifetime of the project, the integration is done is small chunks – ie at the end of each of the "mini waterfalls". Therefore, the integration is relatively straightforward – if you have worked on a large waterfall project before, you can compare this to the "big bang" approach used on those projects, where integration is a long, tedious process.

- **Testing** is performed at the end of each iteration; the testing processes gets progressively more intense as time progresses, as the system is getting bigger and bigger (and more regression testing is needed).

Unified Process – The Pros

- Early, regular feedback (from customer)

If required, we can work very closely with the customer on a UP development; we can involve them in the testing process at the end of each iteration and give them sight of the developing product. Whether this is practical or not is down to the nature of your project, but if it is possible, it is an excellent way of killing small problems before they turn into project killers.

- Risks attacked early in the process

  The elaboration phase is committed to the attacking of risk; rather than hiding them and hoping that they go away later on, risks are exposed through the construction of prototypes.

- Scale and complexity of work discovered earlier

  Again, due to the elaboration effort, the scale and complexity of the work can be discovered earlier (ie before construction and not during). Of course, this isn't a given – it will depend on the skill and experience of the elaboration team – and we'll also need a collection of tools and techniques to make this happen. Perhaps the UML will help here!

- Development Process tested early

  This is a huge boon. By the end of the first construction iteration (ie just a little over 50% of the way through the project), the **entire** development process has been exercised[7]. Say for example that our QA process was badly thought out and implemented – we made the whole process too intrusive, many forms had to be signed and the coding effort was delayed by over two weeks as a result. Well, we have found out straight away and can easily modify it in time for iteration 2.

- Regular Releases => Morale boost!

  This advantage cannot be understated. The development teams get to see a regular expansion in the functionality of the system, and get to feel that progress is constantly being made (as opposed to waterfall projects, where running code might not emerge for years!) UP projects should ensure that they celebrate hard at the end of each iteration!!

[7] Well, apart from Transition but it is fair to say that Transition is *relatively* simple and shouldn't worry us here.

Unified Process: The Cons

The big downside of the Unified Process cannot be emphasized enough – it is **much** harder to manage than linear processes. Check the "planning" activity in the diagram on page 32; it is a constant effort and when things go wrong, it will be immediately obvious.

Much of the management effort is concentrated on the running of the process (ie the progression of the phases and the iterations within) rather than the micromanagement of what the development teams are actually up to. Successful UP projects will generally have an attitude of empowerment for their development teams – and **genuine** empowerment too, rather than paying lip service to it.

It is also harder to understand than the linear processes. We have taken several pages to describe it here, and we have only scratched the surface. Yet we can explain most of the waterfall lifecycle in a few sentences.

It ought to stand to reason that a process that is robust enough to support a project of any duration and complexity is going to be harder to take on board – but it should be recognised that learning how the Unified Process works is hard.

Since the UP is a different way of thinking, it can be an extreme and painful culture change for many organisations. It is often introduced at the same time as other changes (for example, OO development, new operating systems), and it can easily cripple a project if things aren't done properly. Bearing in mind that the Unified Process is harder to understand, it is not surprising that many projects go off half cocked.

Finally, although at the time of writing the UP is considered to be the enlightened method of software development and is very much "state of the art", it is not the most used process – the waterfall will continue to hold that crown for many years to come. There is far less "in the field" experience of the UP.

Iteration Length?

A single iteration should typically last between *2 weeks* and *2 months*. Any more than two months leads to an increase in complexity and the inevitable "big bang" integration stage, where many software components have to be integrated for the first time.

A bigger and more complex project should **not** automatically imply the need for longer iterations – this will increase the level of complexity the developers need to handle at any one time. Rather, a bigger project should require **more** iterations.

Some factors that should influence the iteration length include: (see Larman [2]).

- Early development cycles may need to be longer. This gives developers a chance to perform exploratory work on untested or new technology, or to define the infrastructure for the project.
- Novice staff
- Parallel developments teams
- Distributed (eg cross site) teams [note that Larman even includes in this category any team where the members are not all located on the same floor, even if they are in the same building!]

To this list, I would also add that a *high ceremony* project will generally need longer iterations. A high ceremony project is one that might have to deliver a lot of project documentation to the customer, or perhaps a project that must meet a lot of legal requirements. A very good example would be any defence related project. In this case, the documentary work will extend the length of the iteration – but the amount of software development tackled in the iteration should still be kept to a minimum to avoid our chief enemy, complexity overload.

Some Important Points

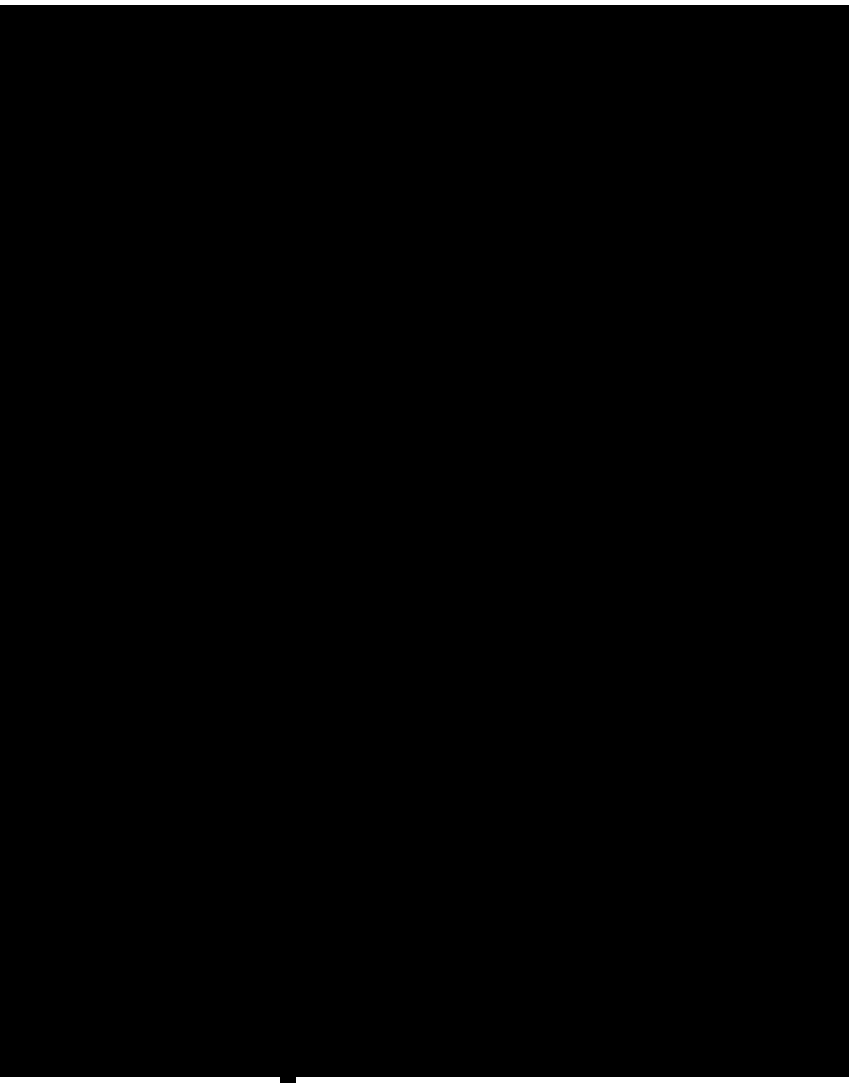
It is worth fixing some common UP misconceptions…

As we previously stated, it is a common misconception to assume that Inception = Requirements; Elaboration = Design and so on. As the previous discussion has proved, each phase in the UP is a combination of activities, and requirements, design, coding and all the other disciplines required on a project are “threaded” around the four phases.

It is not necessary to define most of the requirements before starting design or implementation – in other words, a long and tortuous “requirements analysis” is not required before any work can be done on the design or build. As the diagram on page 32 shows, requirements is an on going activity that needs to be revisited through the elaboration phase (and possibly through construction too).

Complex projects do **not** mean long iterations. Long iterations means waterfalls (if your project wants to use a waterfall, then fine – but if you are using UP then short iterations are the order of the day). Complex projects mean **more** iterations.

Despite the apparent complexity of the UP (and we cannot hide the fact that it is harder to understand, and there is a lot to learn), the UP is not intended to be a formal, heavyweight and complex process - it can be as agile or as prescriptive as your project requires. There is no law stating that your project



XP Rules and Practices – Planning

"User Stories" are written

User Stories are similar to Use Cases. There are some differences, but to be honest it is a bit hard to discern exactly what those differences are. It is a little worrying that there are breakaway camps claiming to be radically different when there isn't really. One interesting idea is that the User Stories are written by the customer (directly). In XP, the customer (or a representative) is part of the team (not a visitor, a full blown member).

Make frequent small releases

The project aims to deliver small releases to the customer, and often.

The "Project Velocity" is measured

The project velocity is a measure of how much progress is being made on the project. The user stories (and not lines of code) are used as the measure of progress.

The project is divided into iterations

XP suggests a schedule of about 10-12 iterations of a duration of about 1-3 weeks each. They also suggest that the iteration length be a constant.

Iteration planning starts each iteration

XP is often mistaken as a hacker's paradise, but really this is a long way from the truth. One example of the control aspects of XP is the concept of an iteration planning meeting - this is held at the start of each iteration, and plans the tasks to be carried out during iteration.

Move people around

Staff on an XP project are regularly moved around - to avoid excessive knowledge being held by individuals and to reduce the "bus count" of the project.

A Stand-Up meeting starts each day

The "stand up meeting" is held each day, a short meeting to which all project members must attend. The style of a standup meeting is that it is extremely short, problems are aired rather than solved (solutions should be found *outside* of the meeting), and attendees must (as the name suggests) stand up when they speak. By making people stand up when they speak, the meeting is less likely to sag and drown in waffle.

Fix XP when it breaks

XP encourages projects to change the rules when they don't work!

XP Rules and Practices – Designing

Simplicity

The XP mantra is "find the simplest thing that could possibly work". XP encourages that functionality should NEVER be added until it is needed.

Choose a “system metaphor”

System Metaphor means to have a consistent naming convention across the whole project. The UP has similar ideas (in the form of “project glossaries”.

Use CRC cards for design sessions

XP encourages the use of CRC (Class Responsibility and Collaboration Cards). This technique (invented by the same people driving the idea of XP) is a common Object Oriented design technique - we'll be doing something similar later.

Create “spike solutions” to reduce risk

Spike solutions are kind of throwaway non-production quality "prototypes", to reduce technical risk.

No functionality is added early

XP encourages that only the work required for the current iteration should be done now. They argue that only 10% of the extra work that you put so much effort into will ever be used.

Refactor whenever and wherever possible

Refactoring, a technique made popular by Martin Fowler is the process of improving the quality of code (or design) that already works. XP encourages constant refactoring. By putting refactoring at the heart of the project, designs are less likely to become unmaintainable, stale or unwieldy.

XP Rules and Practices – Coding

The customer is always available

XP requires that the customer, or a representative (not one of their trainees - an expert is required!), is a part of the development team.

Code must be written to agreed standards

Coding standards must be set and adhered to – again proof that XP is not and has never been a “hacker’s charter”

Code the unit test first

In XP, the idea of writing automated unit tests (that can be run as part of the build process rather than manually) is central. Also of importance is that the unit tests are written **before** the code!

All production code is pair programmed

Perhaps the most famous XP practice. All code that is intended to be included in the production system must be coded by two programmers working together. XP’s findings show that two people working at a single computer will do as much work as the two would have achieved separately – and critically the work will be of a higher standard. The idea is that whilst one programmer is “heads down”, the other programmer can think tactically and objectively.

Leave optimisation until last

Code is only optimised at the end of the project – not as the code is being written. This is an ancient practice and is included in XP for good reasons. “Premature optimisation is the root of all evil” according to Donald Knuth, and we’re not going to argue with him!

No overtime

My favourite (and a rule that I’ve applied on non XP projects too). The extremeprogramming.org website explains that working overtime sucks the spirit and motivation out of a team. We’ll also add to that the wise law invented by the late great C Northcote Parkinson : “Work expands to fill the time available” – so overtime is a waste of time, anyway.

XP Rules and Practices – Testing

All code must have unit tests

As we said earlier, automatic unit tests are of vital importance to XP. All production code is unit testable in this fashion.

All code must pass all unit tests before it can be released

This rule, I think, speaks for itself.

When a bug is found tests are created

On discovery of a bug, tests are written to guard against it returning.

Acceptance tests are run often and the score is published

Acceptance tests are derived from the user stories, and rather than delaying running them until the end of the project, the acceptance tests are run from an early stage. Although many of the tests will fail in the early stages of a project, as time progresses more and more of the tests will pass. The "score" of how many tests have passed are published regularly to the development team, so a clear view of progress made can be seen by all. Contrast this with the approach of counting lines of code written per day as a metric of progress…

Session Summary

- The UML is a bare language and must be implemented alongside a solid process
- An iterative lifecycle offers many advantages over the traditional waterfall
- The Unified Process (UP) is built around four phases - Inception, Elaboration, Construction, Transition
- XP is an interesting movement; it is really only intended for smaller projects (<30 developers), but it is gaining momentum and you cannot fault the core ideas - although it does seem to be in head to head competition with UML style thinking….

Chapter 3

Introducing the Diagrams

Conscious of the fact that we are well into the course now and we've hardly seen any UML, we are going to briefly introduce the diagrams so that you have least seen most of them. You don't need to understand what they are about yet – of course, we'll be working through them in detail through the rest of the week.

We will give a quick description of what each diagram is usually used for - don't take the quick descriptions too literally as we'll be a little bit technically imprecise for the moment!

Although there are about nine different diagrams, there are only five classes of diagram:

- The Use Case Model
- Interaction Models
- State Models
- Implementation Diagrams
- Static Model (Class Diagram)

Note: This material is applicable to UML1.5 (and previous versions). UML 2.0 (only just released at the time of writing and not supported by all tools) adds extra diagrams and renames some of the existing ones. However, the changes are relatively minor and we will cover these in a later chapter.

Models vs Diagrams

Before we begin, it is worth clearing up a common confusion in the UML about the difference between *Models* and *Diagrams*. Many practitioners use the terms interchangeably, but there is a difference.

A **Model** is a collection of information about your project, its artefacts and its relationships. The model could be stored in a database.

A **Diagram** is a visual representation of *some* of the data in your Model.

Many UML tools are quite keen on the difference; often changing the Diagram does not alter the Model.

Take this example from the Enterprise Architect tool. Here, we have two classes on a Class Diagram (Customer and Order). Note that the two classes appear in the tree on the right hand side – this tree is showing the **Model** and how it hangs together…

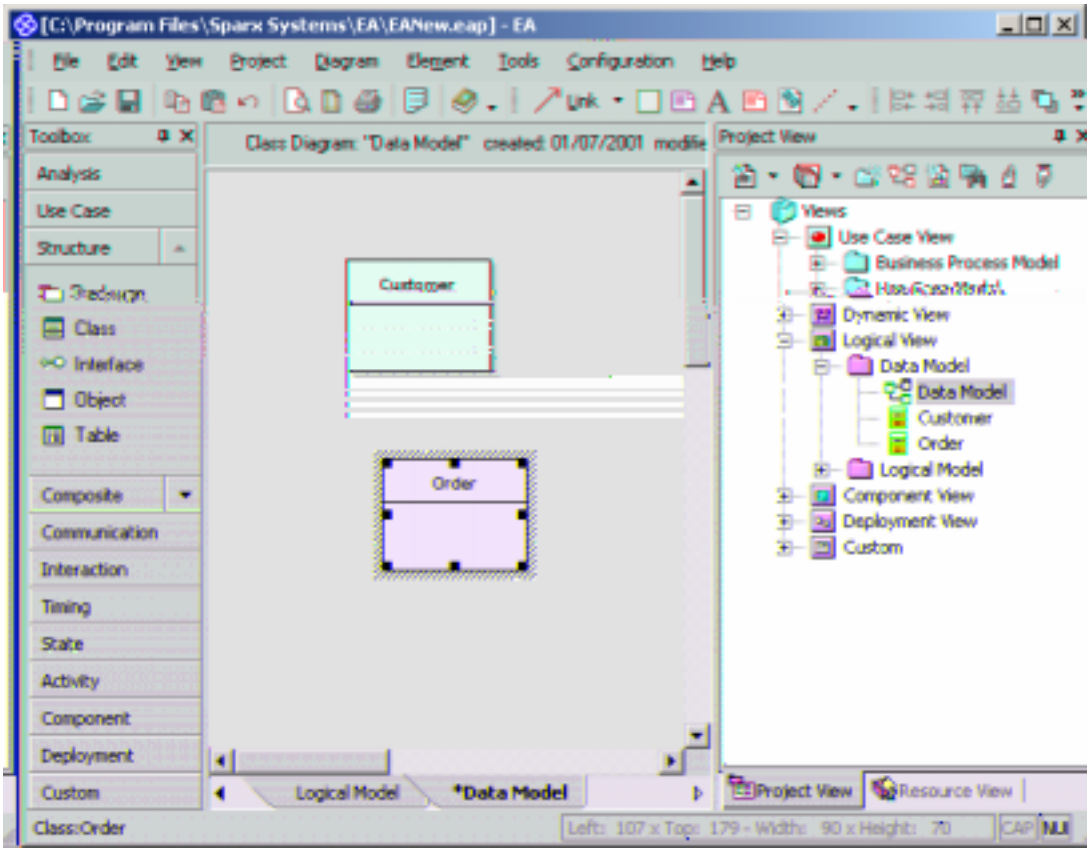


Figure 16 - Two Classes in the Model and the Class Diagram

However, look what happens when we delete the “Order” class from the diagram (using the del key)…

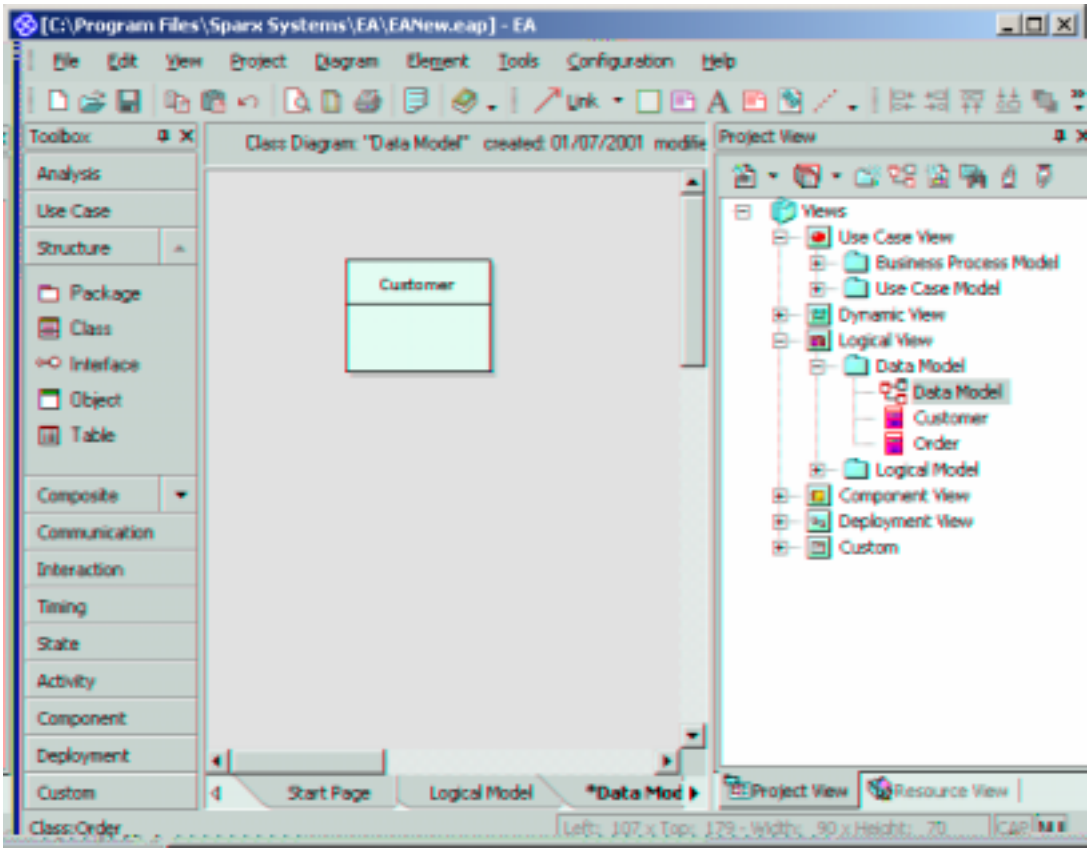


Figure 17 - Deleting the Order class from the diagram

However, notice that the **Order class is still in the Model!** This is because EA is very strict on the distinction between the Model and the Diagram. By pressing delete, we are merely asking EA not to display the class on the Diagram. The class itself is still preserved entirely, including the relationships it has with other classes.

Although it is an important difference (especially when working with tools), don't get too precious about it. We'll often refer to a Model as a Diagram, when we really should say "The Model which is visualized by the following diagram", or vice versa.

Class Diagrams

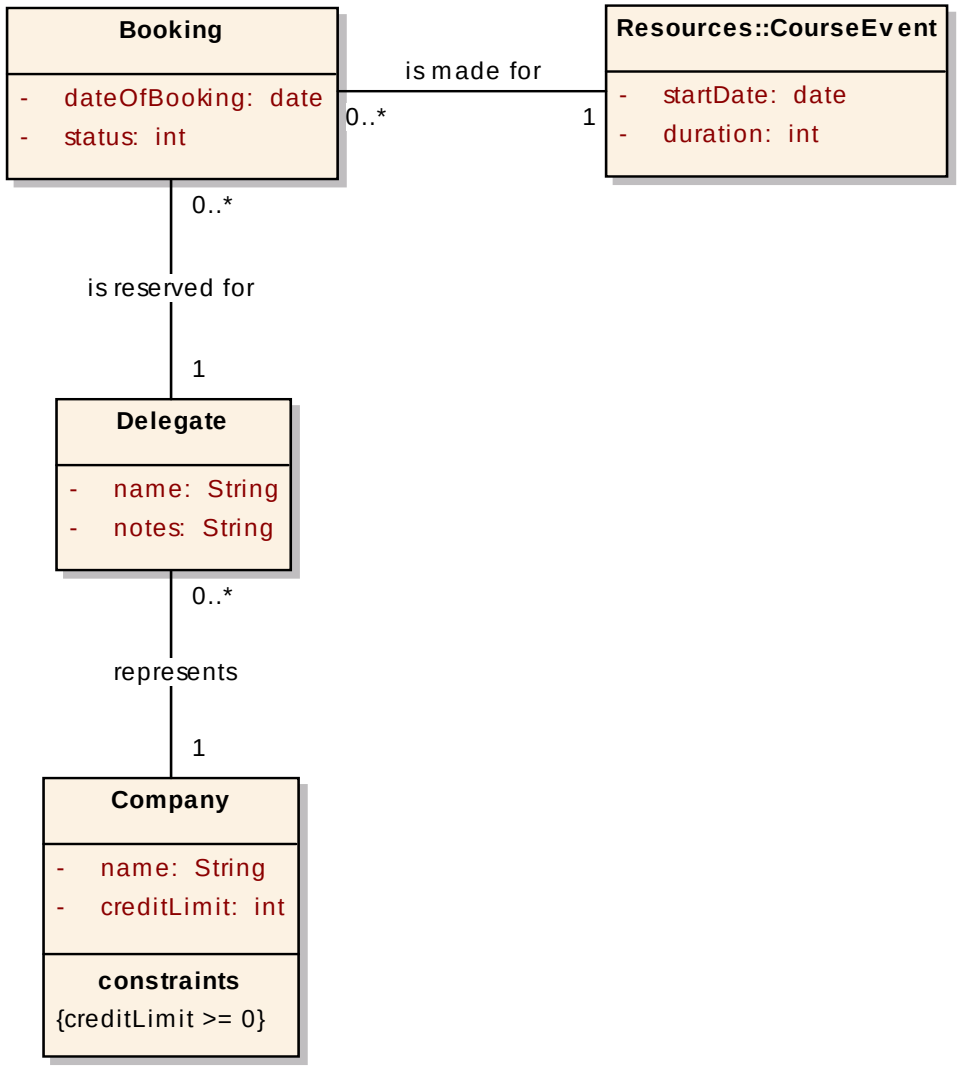


Figure 18 - A Class Diagram

Class Diagrams enable us to capture details about the objects that exist in the problem and/or solution. They are built initially during analysis. This model is transferred into design where it is expanded and eventually transformed into code.

Class diagrams are perhaps the most commonly used UML models.

Statecharts

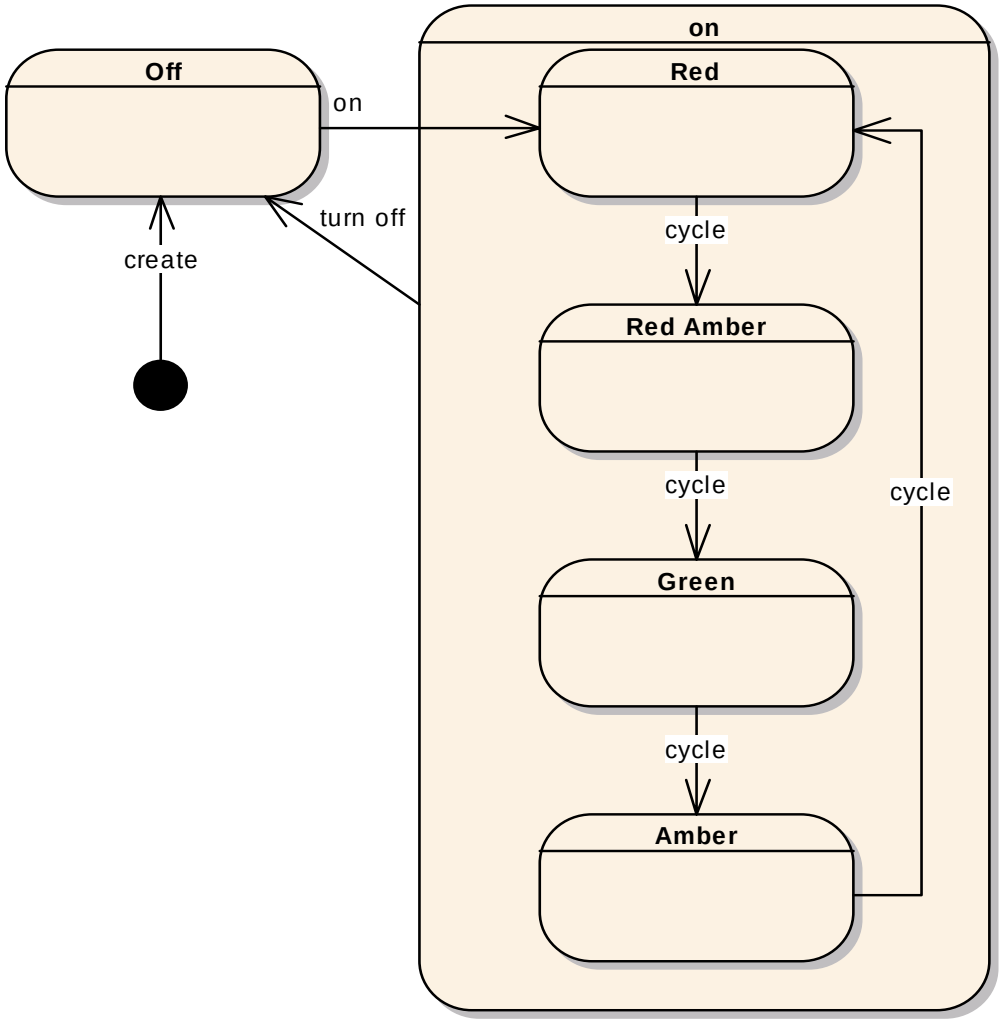


Figure 19 - A State Model

State Charts enable us to ensure that information that we hold is held in the correct state. They are used extensively during analysis and are usually transferred to design.

Projects with safety or time critical behaviour will rely on these diagrams during design.

Interaction Diagrams

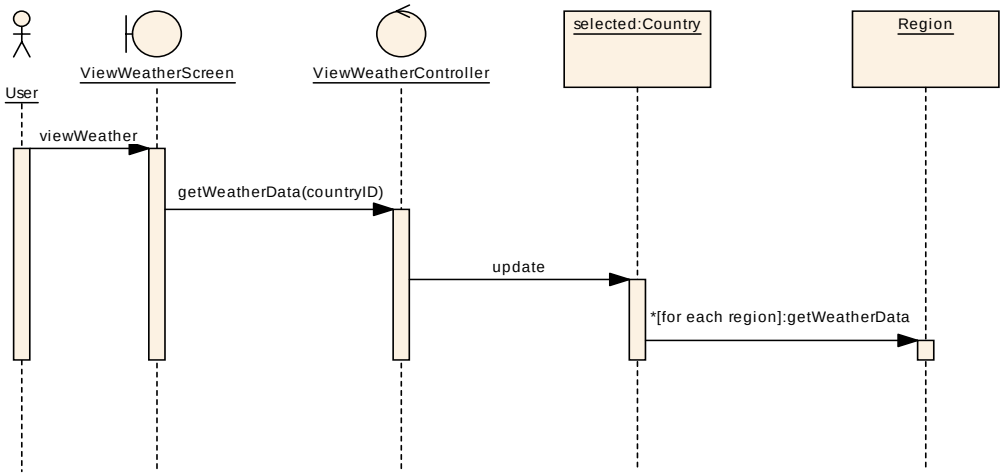


Figure 20 - An Interaction Diagram

Interaction Diagrams allow us to plan how the software will work “under the hood”. They are used in detailed software design, and the “core” of object oriented design work goes on here.

Use Case Diagrams

The Use Case Diagram holds the whole model together…

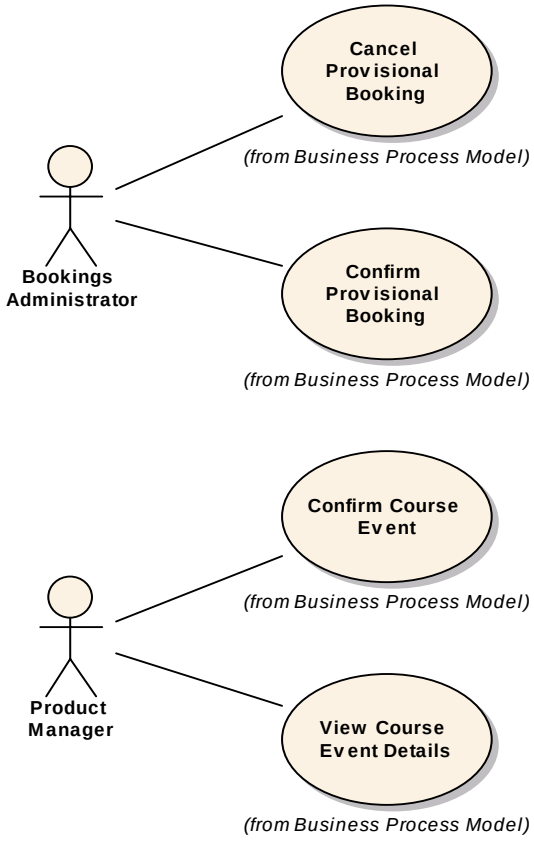


Figure 21 - A Use Case Model

Deployment Model

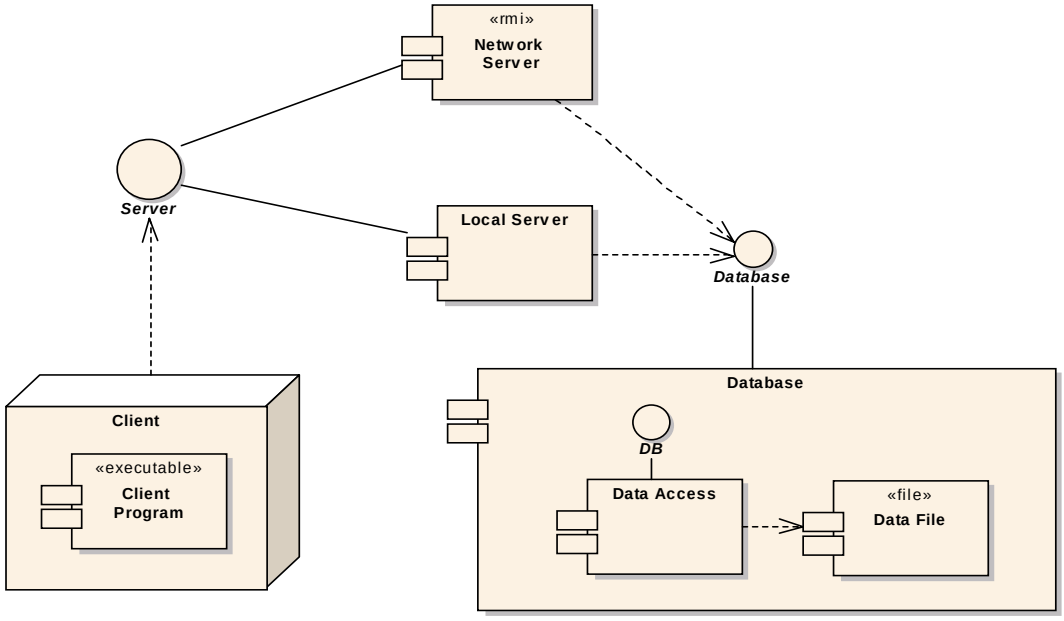


Figure 22 - A Deployment Model

The Deployment Model describes the configuration of the physical artefacts. It is used by architects and hardware engineers.

Activity Diagrams

That's it for the five classes of UML model. However, there is another diagram available in the UML that doesn't neatly fit in to any of the five classes. It is a general purpose diagram called an "Activity Diagram". You will probably recognize it as a flow chart – it has 101 uses – it is usually used wherever there is a need to capture "flow" – but it is not used for Object Oriented Design!

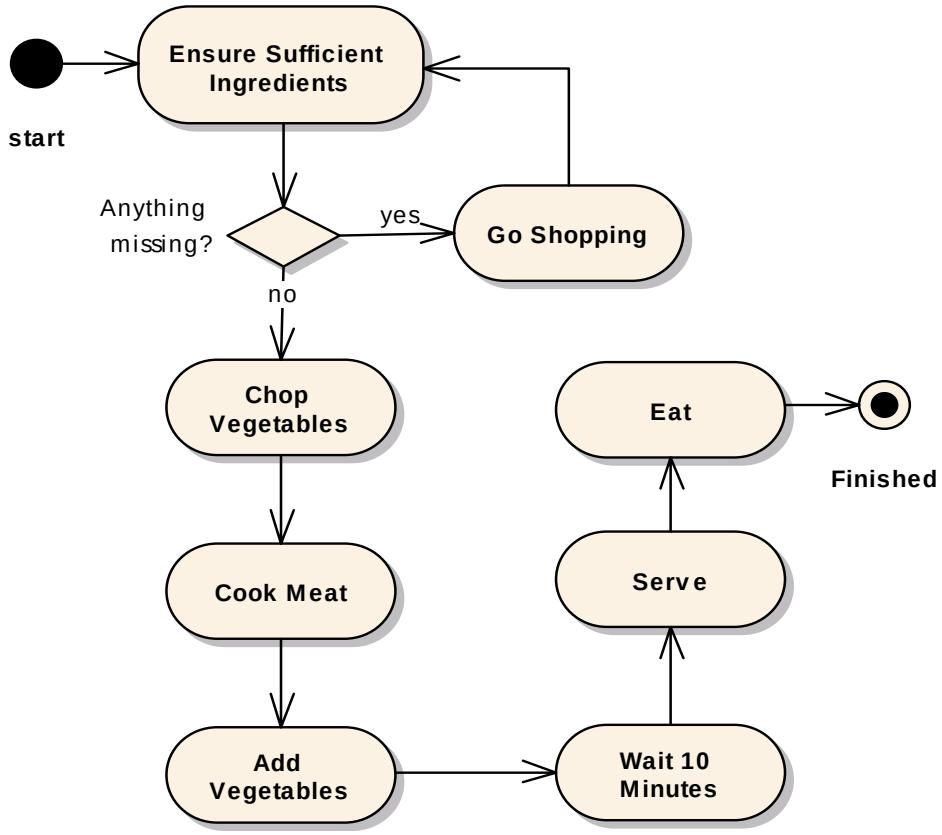


Figure 23 - An Activity Diagram

Linking Together

So we've seen the diagrams, but how do they fit together? The real answer is that the UML specification doesn't explicitly say – in order to allow us to use the UML Diagrams as tools for us to exploit.

However, throughout this course we will present a method of linking the diagrams together to suit our software development…

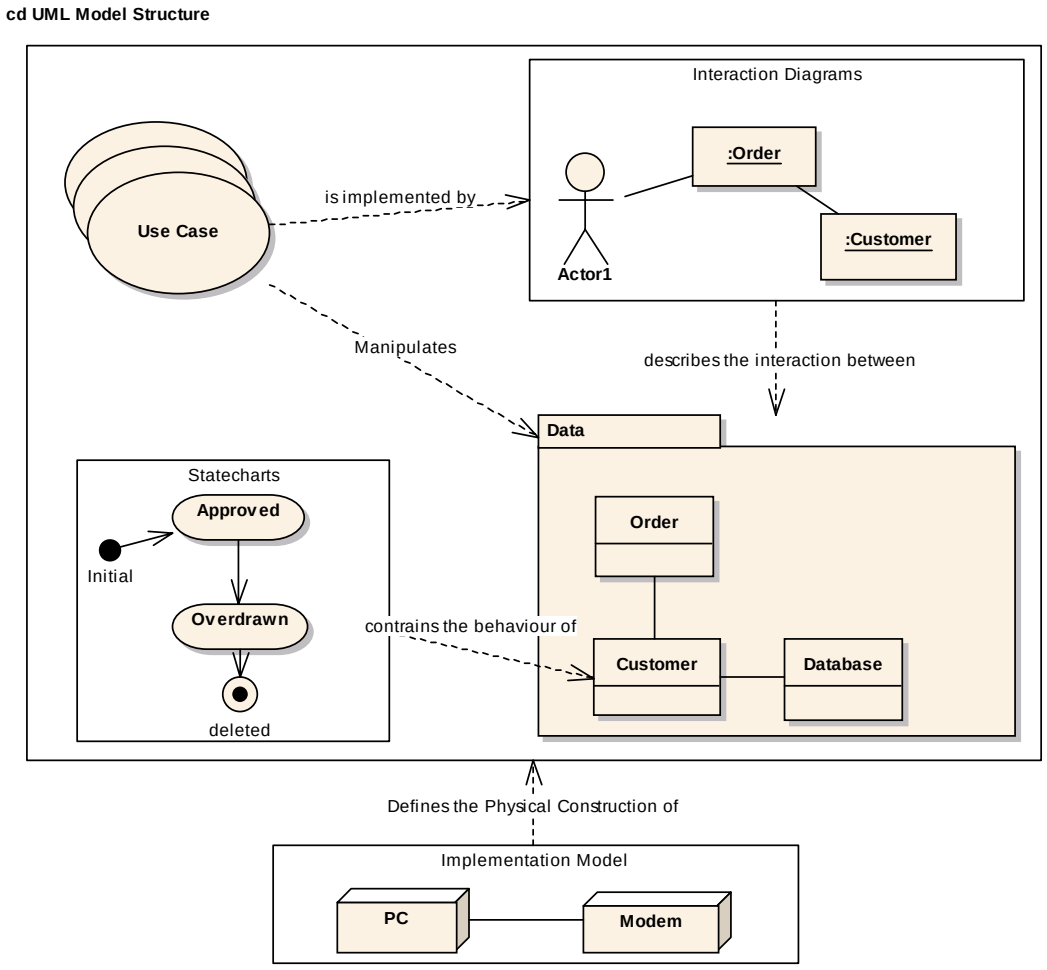


Figure 24 - Possible UML Framework for Software Implementation

Summary

- There are five classes of UML Diagram:
  - Use Case Model
  - Class Model
  - Interaction Model
  - Implementation Model
  - State Model
- There is also a “catch all” diagram called the Activity Diagram

- We'll look at all of these models in detail this week

Chapter 4

Object Orientation

The UML is primarily geared towards Object Oriented (OO) development - although many aspects of the UML are independent of this. To be able to get a grip on the UML, it is important to understand what OO is, why it is a good thing, and what OO is trying to achieve.

This session won't cover OO in technical detail, but it will establish the basics - we will also use some "pseudocode" to illustrate a few concepts.

As an opening gambit, we will look (in very rough terms) at the "structured" approach…

Procedural Programming

First of all, let's examine (in very rough terms) how software systems are designed using the Structured (sometimes called Functional) approach.

In Structured Programming, the general method was to look at the problem, and then design a collection of ***functions*** that can carry out the required tasks. If these functions are too large, then the functions are broken down until they are small enough to handle and understand. This is a process known as **functional decomposition**.

Most functions will require data of some kind to work on. The data in a functional system was usually held in some kind of database (or possibly held in memory as global variables).

As a simple example, consider a college management system. This system holds the details of every student and tutor in the college. In addition, the system also stores information about the courses available at the college, and tracks which student is following which courses.

A possible functional design would be to write the following functions:

```
add_student8
enter_for_exam
check_exam_marks
issue_certificate
expel_student
```

We would also need a data model to support these functions. We need to hold information about Students, Tutors, Exams and Courses, so we would design a database schema to hold this data.9

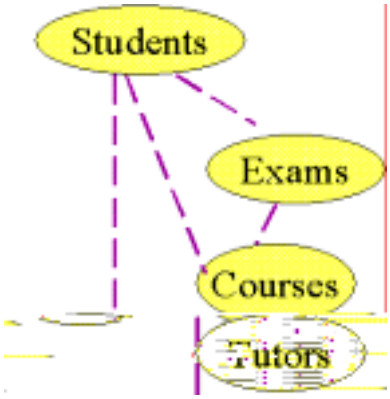


Figure 25 - Simple Database Schema. The dotted lines indicate where one set of data is dependent on another. For example, each student is taught by several tutors.

Now, the functions we defined earlier are clearly going to be dependent on this set of data. For example, the "add_student" function will need to modify the contents of "Students". The "issue_certificate" function will need to access the Student data (to get details of the student requiring the certificate), and the function will also need to access the Exam data.

8 I'm using underscores to highlight the fact that these functions are written in code.

9 Note that throughout this chapter, I am not using a formal notation to describe the concepts

The following diagram is a sketch of all the functions, together with the data, and lines have been drawn where a dependency exists:

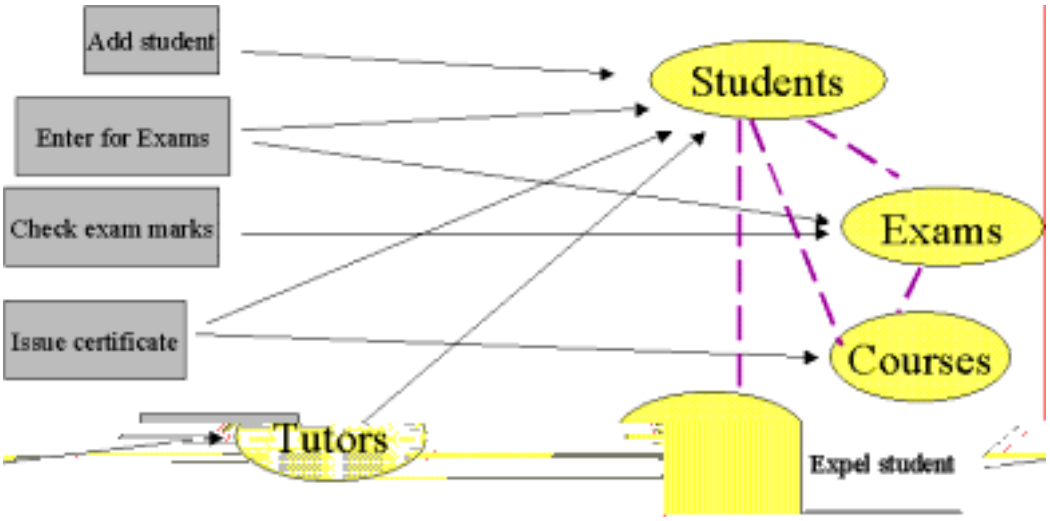


Figure 26 - Plan of the functions, the data, and the dependencies

Procedural Problems

There is nothing inherently wrong with the procedural approach. However, it has been shown to begin to break down once the complexity of the problem reaches a "certain size".

The main problems causing this break down are:

- Separating the Data from the Functions is artificial and leads to stability problems
- As the functions can directly access the data, if the structure of the data needs to change, a major impact can be felt across the code
- The real world doesn't work like this; hence our solutions tend to be overly complex

In other words, it is a "poor abstraction" because it does not model the real world well. This might not have been a problem a decade ago, but as our software systems become more and more complicated we need a better solution.

Object Orientation has been around since the early 1960's, so it is hardly a new idea. But it is only in recent years that much research has gone into OO; these days it is the programming style of choice for most modern languages.

The Object Oriented Approach

In OO, we group together the functions together with the data the functions will access in a single, coherent module

We call this module a "**Class**"…

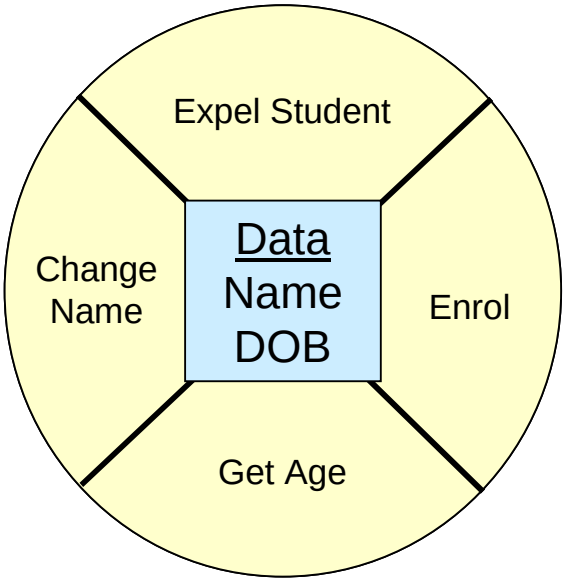


Figure 27 - A schematic of a class

Classes Define Objects

A Class defines the structure (ie the data contents) and the behaviour (the functions) for a type of object

In other words, it **classifies** an object.

To represent actual data when our program is running, we will **create instances** of the class; these instances are called **objects**.

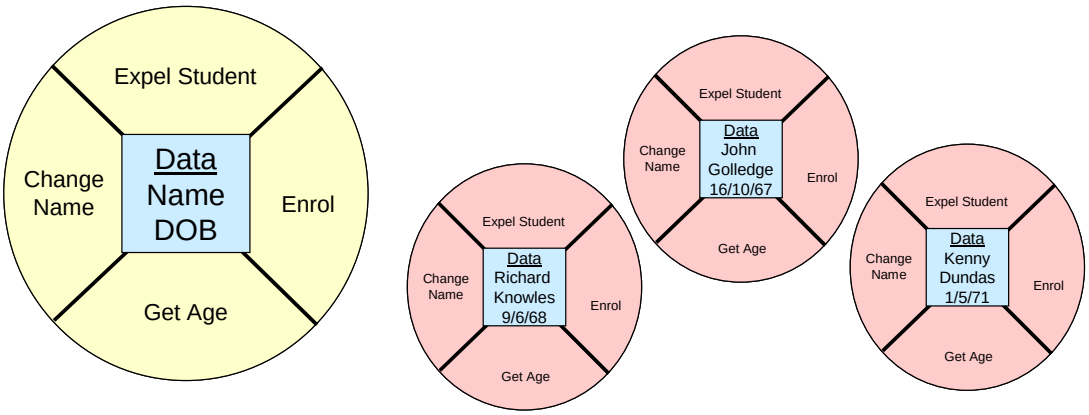


Figure 28 - Classes Define Objects; Objects hold the actual data

So each object holds its own personal collection of data; in theory[10] each object has its own functions.

Encapsulation

This jargon term means that only the functions of an object can access (read, write, update) the data for the object

Our program will work purely through the interaction between methods

In this example, the only way to change the name of the student is through the "Change Name" function.

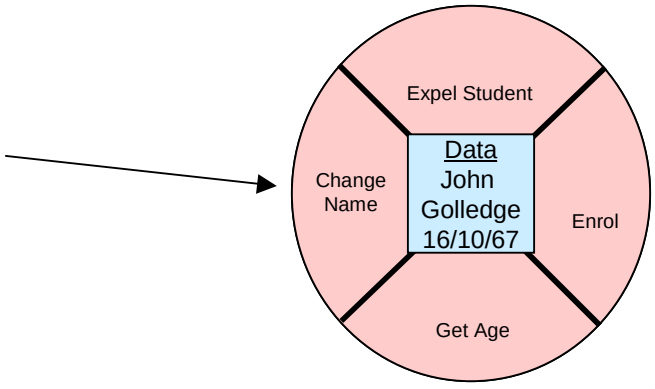


Figure 29 - The data can only be changed through another object calling the object's methods

[10] Java will only really have one copy of a function per class, but it doesn't hurt to think of each object as having its own collection of functions.

The next question is where does this call come from?

Object Collaboration

An Object Oriented system works through chains of objects communicating to fulfil a particular task…

This is called “Object Collaboration”

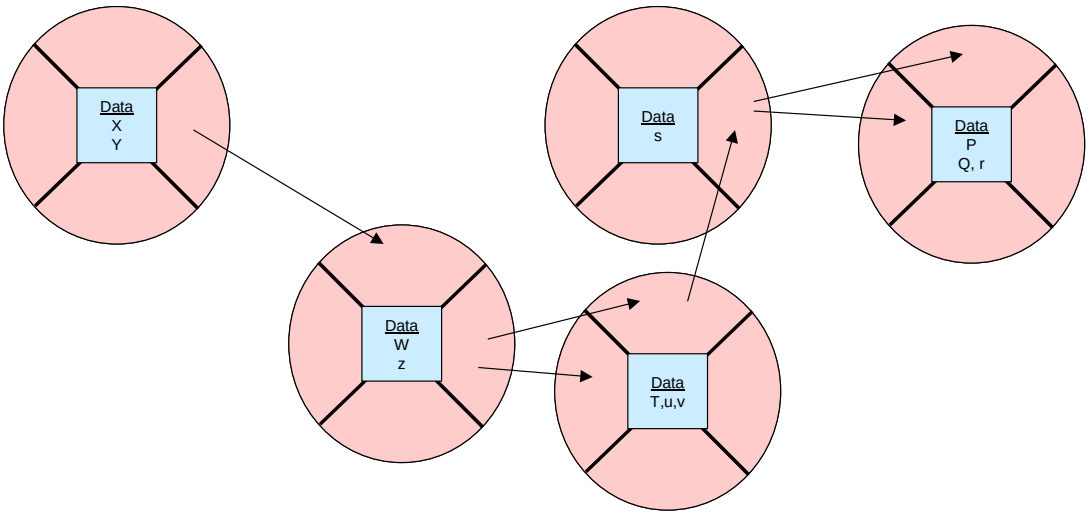


Figure 30 - A Schematic to represent a task being fulfilled through a chain of objects calling each other’s functions in a sequence

How on earth do we work out what these chains should be? Well, this is the art behind object orientation; don’t worry about where all this comes from for now as we’ll be practicing this through the Java case study.

One thing you can think about – where does the *first* function call in an Object Oriented system come from?

OO Jargon

Unfortunately, OO is overburdened with a vast array of jargon. We call the data in a class the **attributes** of the class. We call the functions (or procedures) in a class the methods of the class.

There *is* a technical difference between a function and a method, but it isn’t really worth worrying about the difference. We don’t (much), and if you want to think of a method as being a function or procedure, you won’t go far wrong.

Why Objects?

Throughout this chapter, I have referred to these collections of related data and functions as being "modules". However, if we look at the characteristics of these modules, we can see some real world parallels.

Objects in the real world can be characterised by two things: each real world object has **data** and **behaviour**. For example, a television is an object and posses data in the sense that it is tuned to a particular channel, the scan rate is set to a certain value, the contrast and brightness is a particular value and so on. The television object can also "do" things. The television can switch on and off, the channel can be changed, and so on.

We can represent this information in the same way as our previous software "modules":

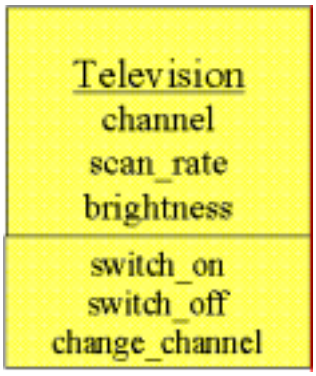


Figure 31 - The data and behaviour of a television

In some sense, then, real world "objects" can be modelled in a similar way to the software modules we discussed earlier.

For this reason, we call the modules **Objects**, and hence we have the term **Object Oriented Design/Programming**.

Since our software systems are solving real world problems (whether you are working on a College booking system, a Warehouse Management System or a Weapons Guidance System), we can identify the objects that exist in the real world problem, and easily convert them into software objects.

In other words, *Object Orientation is a better abstraction of the Real World*. In theory, this means that if the problem changes (ie the requirements change, as they always do), the solution should be easier to modify, as the mapping between the problem and solution is easier.

Our General Strategy

Although this chapter has briefly touched on the benefits of Object Orientation (ie more robust systems, a better abstraction of the real world), we have left many questions unanswered. How do we identify the objects we need when we're designing a system? What should the methods and attributes be? How big should a class be? I could go on! This course will take you through a software development using Object Orientation (and the UML), and will answer all these questions in full.

One significant weakness of Object Orientation in the past has been that while OO is strong at working at the class/object level, OO is poor at expressing the behaviour of an entire system. Looking at classes is all very well, but classes are very "low-level" entities and don't really describe what the system as a **whole** can do. Using classes alone would be rather like trying to understand how a computer works by examining the transistors on a motherboard!

The modern approach, strongly supported by the UML is to **forget** all about objects and classes at the early stages of a project, and instead concentrate on what the system must be able to do. Then, as the project progresses, classes are **gradually** built to realise the required system functionality. Through this course, we will follow these steps from the initial analysis, all the way through to class design.

Inheritance

Inheritance is an important feature of Object Orientation (and it is therefore fully supported in Java). It allows us to “reuse” existing classes by taking their definitions and extending them into new, more specialised classes. Although an attractive idea, inheritance must be used with care - we’ll study inheritance in more detail towards the end of the course.

Database Mapping

Relational Databases were not designed with OO in mind (OO was not a commonly accepted way of thinking when EF Codd invented the concept of relational databases in the late 1960’s/early 1970’s). Many organizations rely on relational database stores, and cannot dispense with their systems lightly. Although there is some similarity between OO and RDBMS, there are sufficient differences to make the “bridging” between the two ways of thinking an expensive pain. Much research is being done in this area, however, and many products exist to help us to bridge the gap (see reference [8] for some thoughts on this serious issue.

The difference in the relational way of thinking to the OO way is often called the "OO/RDBMS impedance mismatch"

Bridging the Gap

The resolution between this "mismatch" is something that only your architecture can address[11]. There are many different solutions:

- Direct calls to the database from the objects (not OO at all)
- Data Access Objects (still not OO but a workable solution)
- A more serious database mapping "layer" – this is often called a "Persistence Framework". This layer could be built by yourselves (but this is unlikely as Persistence Frameworks are very, very difficult to implement from scratch)

If your solution is to use a Persistence framework, many open source options exist…

Persistence Frameworks

For example, in the Java world, there are dozens of open source mapping frameworks (although this is an area that changes rapidly).

The current contenders for leading framework are:

- The Spring Framework (see Rod Johnson's book "J2EE without EJB") or springframework.org
- Hibernate (see Hibernate.org)
- Entity EJBs (but not considered to be much of a solution)

[11] Our aim for this course is remain architecture neutral. However, we do currently offer courses that cover concrete solutions for Java based developments – Hibernate, Spring Framework and EJB.

Keep an eye on developments in this field as there are still many issues to be resolved…

Summary

- Object Orientation is a slightly different way of thinking from the structured approach
- We combine related data and behaviour into classes
- Our program then creates instances of the class, in the form of an object
- Objects can collaborate with each other, by calling each other's methods
- The data in an object is encapsulated - only the object itself can modify the data
- A major consideration in OO is persistence; only your architecture can address this

Digression : UML Stereotypes

A UML Stereotype enables us to take an existing piece of notation and to give it a different meaning; often this different meaning is a stronger or more specific interpretation of the icon's original intention.

As an example, consider a diagram where we wish to express that we need to connect two PC's (a server and a client) together using an RS232 link. The icon for a PC in the UML is a Cube, and the connection is notated using a simple line (the UML calls this an association).

Figure 32 - the UML representing our Client/Server setup

The diagram above is reasonable and is certainly valid UML. But we haven't captured the requirement that the connection must use RS232. Now, the UML as it stands does not contain any elements to represent an RS232 link. But we can take the association graphic (the straight line) and apply a stereotype to give it the stronger, more specific meaning. To apply a stereotype, you simply tag the required icon with the name of the stereotype in Guillemets[13] (French quotation marks). The diagram now looks like this:

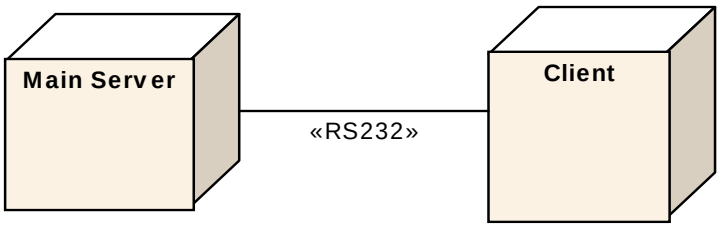


Figure 33 - Using the RS232 Stereotype

There is nothing to stop your project defining their own stereotypes to suit your own requirements. Bear in mind that if you do this, you are producing

[13] Pronounced "gee-may". Quite why the designers of the UML chose such a symbol is a mystery. Jacobson concedes in the UML spec that "typographically challenged" people can use double angle brackets like <<this>>.

proprietary extensions to the UML. More commonly, we use standard stereotypes[14].

Deployment Syntax

The syntax for this model is very simple indeed. Here it is, in pretty much its entirety:

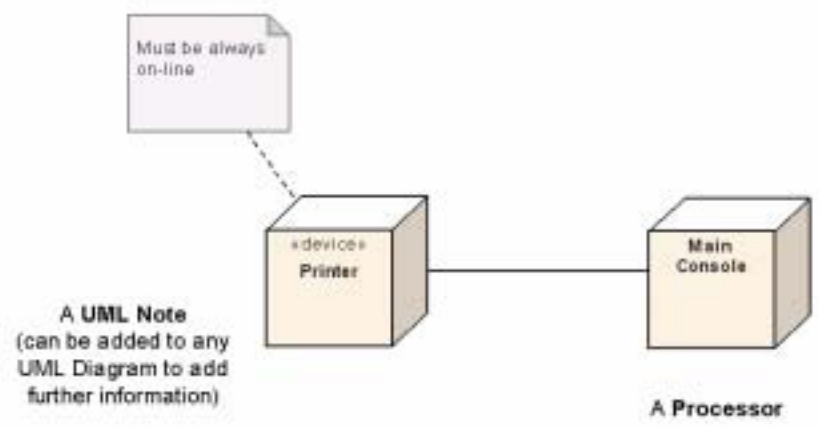


Figure 34 - The UML Deployment Diagram

[14] These standard stereotypes are defined in documents called “UML Profiles”. A profile is a way of using the UML in a different context. A profile exists for Real Time/Embedded systems, and one exists for analysis work.

Chapter 6

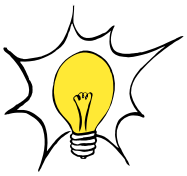
Discovering Use Cases

This session covers:

- The purpose and importance of Use Cases in the UML
- Actors and Use Cases defined
- Uncovering Use Cases from the business processes
- Creating the Use Case Model
- Ranking use cases

Use Cases

Let's be more specific about what a Use Case is. A Use Case describes a set of interactions with the system that supports a particular business goal. We need to be able to capture and describe these Use Cases, so all Use Cases have a name that describes the goal of the Use Case – usually a verb/noun combination.



Use Cases are almost always described using a simple, clear and concise verb/noun combination.

For example, imagine we are working on a submarine control system – a possible name for a Use Case could be "Fire Torpedoes", as this is almost certainly something that one of the users on a submarine may wish to do from time to time.

Using the UML, we can capture these Use Cases on a simple diagram. Don't let the simplicity of the diagram deceive you; Use Cases are incredibly powerful…

Use Case Symbol

The symbol for a Use Case is a simple oval, with the name of the Use Case inside or below the oval (depending on your tool):

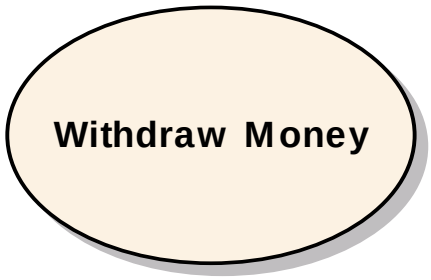


Figure 35 - The Symbol representing a Use Case

Now let's look at Actors…

Bring on the Actors

An Actor is someone (or something) that can **trigger** a use case. In our previous example of Withdraw Money, the actor was probably **Customer.**

Here is the UML notation for an actor:

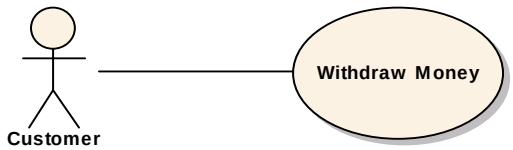


Figure 36 - Adding an Actor to the diagram

Notice that the actor is connected to the Use Case they are able to trigger via an association line. Some tools will add an arrow head to the end of the line; however this isn't really meaningful, as the actor is always the trigger for the Use Case (so the arrow is implicit).

Example Use Case Diagram

Of course, a full Use Case Diagram will contain many Use Cases and Actors. An actor can trigger more than one Use Case, and a single Use Case can be triggered by one or more actors. Here's a quick example:

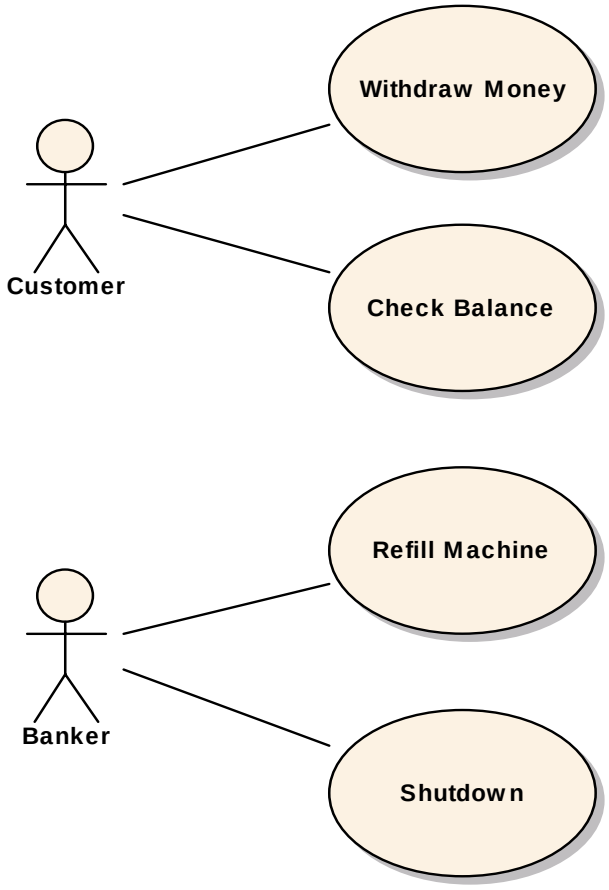


Figure 37 - Example Use Case Diagram

The Purpose of Use Cases

We have established so far that the Use Case diagram is simple and easy to draw. Are Use Cases themselves simple, or perhaps trivial? In fact, the Use Case diagram is considered to be the cornerstone of the UML. Here are a few key uses of the Use Case diagram:

- Use Cases define the scope of the System - the "sum" of the use cases is the whole system. Therefore, Use Cases can take the place of functional requirements – and if we use them correctly, they can do a much better job of representing the requirements.

- They allow for communication between the customer, analyst and developers. Because the diagram is so simple, even non-IT literate people can grasp them – yet they still contain sufficient detail for the developers to work with (albeit as a starting point)

- Guide the development teams through the development process. We discussed in detail in " mind its advantages and disadvantages). The UML " (Page 26) that iterative development depends on us slicing the development into independent "chunks". Well, practical experience suggests that Use Cases are an ideal method of creating these "chunks" – therefore…

- Use Cases are the method for planning and estimation; put simply, a system of 20 Use Cases, with time for 4 Use Cases per one month iteration means a 5 month project.

- Use Cases can be tested; once a Use Case has been coded, it is a large and significant enough block of functionality to be tested in its own right.

- Use Cases help with the creation of user guides; often a single Use Case is big enough to warrant a chapter in the user guide, and the steps inside the use case should be roughly the content of the user guide (because the descriptions of the use cases, which we will see later, are written from the user's point of view).

More on Actors

Actors are often real people – as we saw with the previous example of "Customer". But, don't forget that we defined an actor to be anything that can trigger a Use Case. Therefore, actors could also be:

- another computer system

- a mechanical object

- or some kind of time-based event- for example, "end of the month"

In Real Time Systems, actors are often time or event based systems[15]

[15] Some practitioners define an actor to be "an entity that derives benefit from a Use Case". This definition really prevents the use of time based actors (how can End of the Month derive a benefit from sales figures being run)? In fact, this isn't an issue since the UML specification

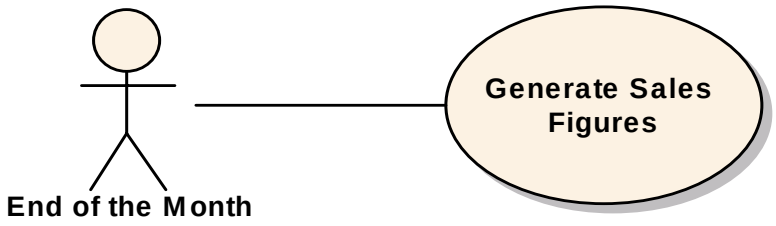


Figure 38 - A time based (aka temporal) Actor

Use Case Granularity

One of the biggest and most open questions when producing Use Cases is how big should they be – and it isn't a question that is really addressed in the UML specification.

We need a rule of thumb; some guidance that will help us get 99% of our Use Cases correct, even if it is a rule that we occasionally break (if we have a good reason).

As an example, let us consider the example of the ATM machine. We need to build the ATM system to allow a user to withdraw money. We might have the following series of common interactions in this scenario:

- Read Card
- Read Pin Number
- Read Amount Required
- Dispense Cash
- Eject Card
- Print Receipt

Should each of these steps be a Use Case?

does not define an actor to require a benefit from the use case, although this is a useful rule of thumb, as we'll see later on.

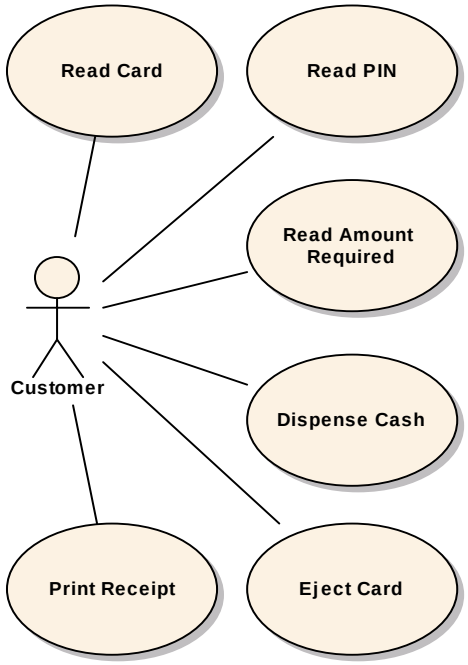
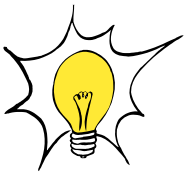


Figure 39 - A Useful Use Case Diagram?

According to the definition of Use Cases, this diagram is perfectly correct, but in practice it probably won't be very useful. We have generated a large number of small, almost inconsequential Use Cases. In any non-trivial system, we would end up with a huge number of Use Cases, and the complexity would become overwhelming.

To handle the complexity of even very large systems, we need to keep the Use Cases at a fairly "high level". The detail can be attacked later in the process (at detailed analysis and design). The best way to approach a Use Case is to keep the following rule-of-thumb in mind:



A Use Case should satisfy a goal for the actor

Apply the rule to these Use Cases, and you'll find that really, none of them describe the goal of the user. The goal of the user is to **withdraw money**, and that should be the use case!

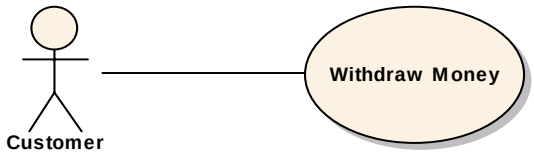


Figure 40 - A more focused Use Case

Just to return back to the example of the time based actor in ity

One of the biggest and most op again, note that our rule of thumb doesn't quite apply, as the "end of the month" doesn't really have a goal as such – but that doesn't matter. The goal of the business is to receive a Sales Figures Report at the end of the month, so the granularity "feels" about right – trust your own judgement.

Finally, be especially careful not to confuse a Use Case with a Business Process. A Use Case is IT based, and will be designed and coded and appears in the final system. A Use Case is usually achieved in a single "sitting" by the actor, whereas a Business Process is usually "end-to-end" and can last for days or weeks – and could involve non IT activities as well.

Cockburn's Use Case Levels

Alistair Cockburn, a leading thinker on the effective use of Use Cases, has proposed that Use Cases can be categorised into three levels:

- Summary Goals
- User Goals
- Subfunctions

This is a useful categorisation, and can help clarify the granularity we are aiming for…

Use Case Levels

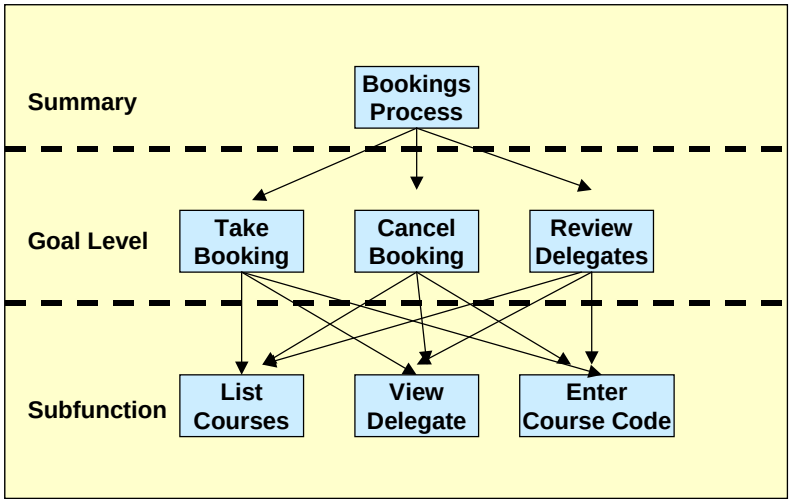


Figure 41 - Cockburn's Use Case Levels

Effectively, Cockburn is saying that Summary Level Use Cases (Use Cases describing high level business functions) are useful when you are aiming to capture your high level detail. We tend to handle this level of detail with Business Process Modelling (see Ariadne's Business Analysis course for more details).

The Goal Level Use Cases are the level we have been discussing in this chapter.

Subfunction Use Cases are smaller, step-oriented Use Cases that an actor would expect to execute as a step in a Goal Level Use Case.

Sometimes, it **is** useful to identify Subfunction Use Cases, but the danger is that the Use Case diagram bloats into a complex mess – and the Use Case Diagram simply isn't good at functional decomposition.

It may be the case that at the next level of design detail, you do wish to organise your diagram and pull out "common" Use Cases – this is fine, but identify your Goal Level Use Cases first.

Note – some projects have misinterpreted Cockburn's classifications as permission to perform functional decomposition using Use Cases. They usually regret.

Example

Here's a simple set of requirements; can you identify the major Use Cases?

- The Computer Bookmaking System shall allow users to place bets on a particular horse race. The system invites users to select a race, and once the race is selected, they are provided with a list of the runners in the race together with their odds.
- Once their selection has been made, the user enters their stake, and the money is deducted from their betting account.
- At the end of each race, the bets on the race are settled. Winning bets are credited to the account.

Example Use Case Model

Space is provided here to note down your answer:

Finding Use Cases[16]

One approach to finding Use Cases is via interviews with the potential users of the system. This is a difficult task, given that two people are likely to give two completely different views on what the system should do (even if they work for the same company)!

Certainly, most developments will involve some degree of direct one-to-one user communication. However, given the difficulty of gaining a consistent view of what the system will need to do, another approach is becoming more popular – the *workshop*.

The workshop approach pulls together a group of people interested in the system being developed (the *stakeholders*). Everyone in the group is invited to give their view of what the system needs to do.

Key to the success of these workshops is the *facilitator*. They lead the group by ensuring that the discussion sticks to the point, and that all the

[16] Business Process Modelling is perhaps the best way to identify Use Cases if you are building a system for a business you are not familiar with – see Ariadne's detailed Analysis with UML course for details.

stakeholders are encouraged to put their views across, and that those views are captured. Good facilitators are priceless!

A scribe will also be present, who will ensure that everything is documented. The scribe might work from paper, but a better method is to connect a CASE tool or drawing tool to a projector and capture the diagrams “live”.

The simplicity of the use case diagram is critical here – all stakeholders, even non-computer literate stakeholders, should be able to grasp the concept of the diagram with ease.

A simple method of attacking the workshop is:

1) Brainstorm all of the possible actors first

2) Next, brainstorm all of the possible Use Cases

3) Once brainstorming is complete, as a group, justify each Use Case through by producing a simple, one line/paragraph description

4) Capture them on the model

Steps 1) and 2) can be reversed if desired.

Some good advice on the workshop:

- *don’t work too hard trying to find every single Use Case and Actor!* It is natural that some more use cases will emerge later on in the process.
- *If you can’t justify the Use Case at step 3), it might not be a use case*. Feel free to remove any Use Cases you feel are incorrect or redundant (they’ll come back later if they’re needed!)

The above advice is not a license to be sloppy, but remember the benefit of iterative processes is that everything doesn’t have to be 100% correct at every step!

Brainstorming Advice

Brainstorming is not as easy as it sounds; in fact I have rarely seen a well-executed brainstorm. The key things to remember when participating in a brainstorming session are:

- Document ALL ideas, no matter how outrageous or nonsensical they seem. Stupid ideas *might* turn out to be very sensible ideas after all
- Also, silly ideas may trigger off more sensible ideas in other people’s mind – this is called *leapfrogging ideas*
- Never evaluate or criticise ideas. This is a very hard rule to observe – we are trying to break human nature here!

"mmm. No, that won't work. We won't bother to document that!"

The facilitator should keep on their toes and ensure that all ideas are captured, and that all of the group participate.

On the course, a Use Case Workshop will be carried out alongside our client.

Final Notes on Use Cases

It is possible to "relate" use cases and to break use cases up into smaller use cases. However, there is a danger that this idea can be abused or used as an excuse for functional decomposition - It is arguably too early in our project to do this "detailed" work just now anyway, so we will leave this concept to one side (it is interesting to note that many projects ban the use of Use Case relationships anyway).

A commonly used concept is of "primary" and "secondary" actors – this is **not** formally part of the UML, but many projects find this exercise beneficial…

Primary and Secondary Actors

A **primary actor** is the actor that receives the benefit of the Use Case. A **secondary actor** is an actor that plays a part in, or supports in some way, the Use Case. A secondary actor (like a primary one) will be outside the system, and often an external computer system.

Identifying Secondary actors helps to clarify external interfaces and the protocols to communicate with them.

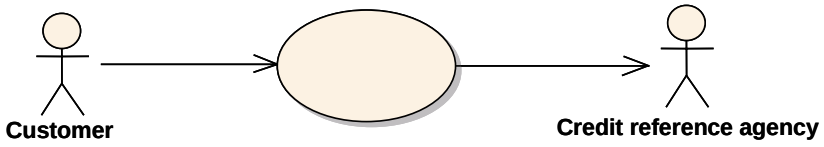


Figure 42 - Using Primary and Secondary Actors

A common practice (again, NOT a UML rule) is to place primary actors on the left, and secondary actors on the right.

On our project, they are not mandated, but use them if you wish, or if your real life project employs them.

Summary

In this session we covered:

- The purpose and importance of Use Cases in the UML
- Actors and Use Cases defined
- Brainstorming to discover Use Cases
- Creating the Use Case Model
- Primary and Secondary Actors

In this session, we have produced first cut (Short) Use Cases; formal detail will follow later

Chapter 7

The Domain Model

In this session, we will:

- Introduce the idea behind Domain Modelling
- Introduce a technique for starting off the model
- Define the necessary jargon terms
- Build a very simple Domain Model

The Domain Model

The Domain Model is a UML Model that aims to establish a common vocabulary for the project. It is simply not possible to create detailed use case descriptions, business rules or state-charts unless we first agree a common terminology.

The model essentially provides a definition of the classes of objects that exist in the business and how they are related. It will also help us to define some of the business rules.

The overall aim is to obtain a deeper insight into the business, thereby ensuring that other analysis and design activities are more complete.

The Domain Class Model can be thought of as an Analysis activity; however it is also the starting point for the design.

What is a UML Domain Model?

In UML it is a diagram showing:

- The **classes of objects** from the problem domain
- The **attributes** of those classes (or the data that is pertinent to each class of object)
- The **associations** between the classes; in other words, how the classes of objects are related

It is a first cut attempt to identify the major business classes - it will evolve into design class model(s) later (and in fact the model we build here should form the basis of the eventual code).

It looks very much like an Entity Relationship Diagram (from other methods) or a Data Model.

Beware – the biggest danger with this model is to start to lapse into design! The key here is to Capture The Requirement!

What is a Domain Object?

We have mentioned the term Domain several times so far without formally defining it.

A Domain Class (we'll call it simply "class" from now on) is a type of object or entity which is fundamental to, and recognised by the business.

We are looking for objects (or things, or ideas) that the user understands or recognises, so some good examples of domain classes from various different businesses/domains:

- Lift
- Cloud Formation
- Footballer
- Postal Order
- TV Channel

And here are some bad examples:

- OrderPurgeDaemon

- EventTrigger
- CustomerDetailsForm
- DbArchiveTable

All of these bad examples are pure IT “fabrications” – leave these until we understand the problem.

The art of building a Domain Model is in finding the classes in the first place…

Finding Classes of Objects

Use singular, noun-phrase names from the Business domain. The best way of finding out what the classes should be is probably through a facilitated workshop with representatives from the business present.

Following the brainstorming session, test the validity of each class by asking : “Is it easily described?” (and can you come up with a simple description?)

As a start point, consider the following:

- Physical or tangible objects in the problem
- Places
- Containers for other Objects
- Other Systems external to the system (eg Remote Database)

A Class in UML Notation

A class in UML is denoted as follows. The Class is divided into three boxes. In the top box, we write the name of the class, in the second we list the attributes and in the third, we list the operations:

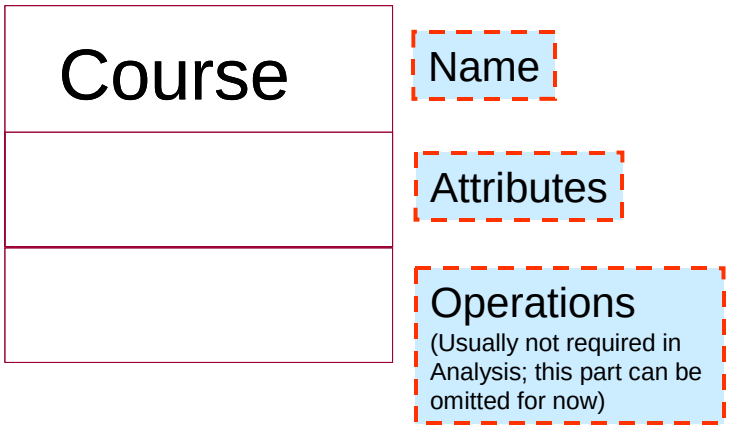


Figure 43 - A Class in UML Notation

We'll look at what the attributes box should contain in a moment. The Operations box is usually not required in analysis; we'll return to this at design.

Documenting Domain Classes

Domain classes should be documented in the class specification. As a minimum the class should have a description, a list of synonyms (aliases, if any) and an indication of volume (current number of instances) and projected growth rate (this will help in database or memory sizing).

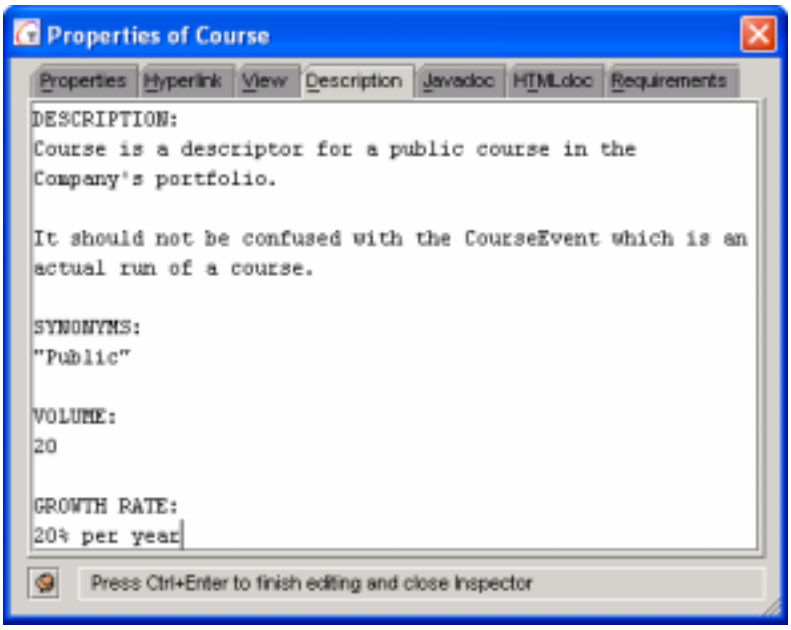


Figure 44 - The description for a course class

Adding Attributes

An attribute describes a piece of information associated with a class:

- A text string (Name, Address)
- A numerical value (Quantity, Weight)
- A boolean (IsCancelled)
- A date/time (DateOrdered, TimeReceived)

An attribute is usually (but not always – this is a rule of thumb) a single valued primitive.

UML Notation

Some projects capture the data types at this stage. The UML notation for datatypes is shown below.

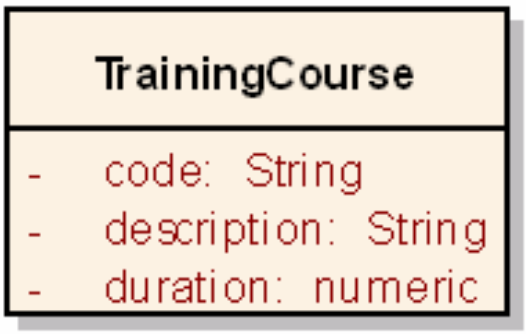


Figure 45 - Listing attributes

If this activity is carried out as an analysis task, then the datatypes can be omitted, or a project wide list of “analysis types” are maintained (eg, date, alphabetic, constrained, positive, sequence, etc…). The mapping to true code types is done later.

Notice the style of the attribute definition – it is formally stated as follows:

```
attributeName : AttributeType = DefaultValue
```

Attribute Guidelines

The most important question to ask is “does the user recognise them?”. Other questions to ask are:

- If the attribute is composite (eg Address), should it be a class instead? Not necessarily – use your own judgement.
- Don’t include association (or “foreign key”) attributes. Associations will be implemented using association attributes when designing/coding but NOT NOW

The question of whether a possible attribute should be a class instead is a thorny one – your own judgement should be sufficient to make the decision but *don’t agonise over it!* It is probably an attribute if you think of it as a ‘primitive’ - eg a number or a textual string associated with a class

It is probably a separate class if you think of it as a collection of information or other classes.

Example Attributes

Here are some attributes added to the domain model for a college management system (we are sure you could do a lot better!)

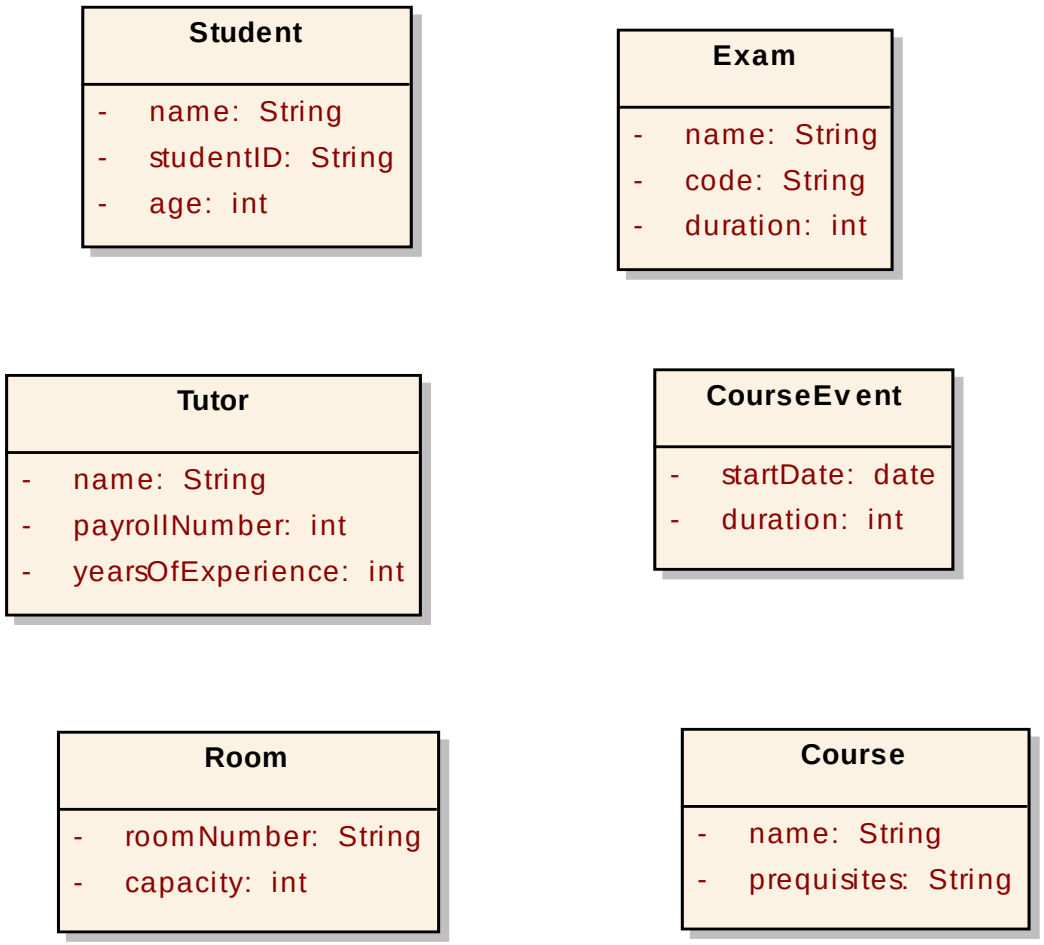


Figure 46 - Example Attributes for a College Management System

Here's an example of the relationship between Courses, Course Events and Rooms…

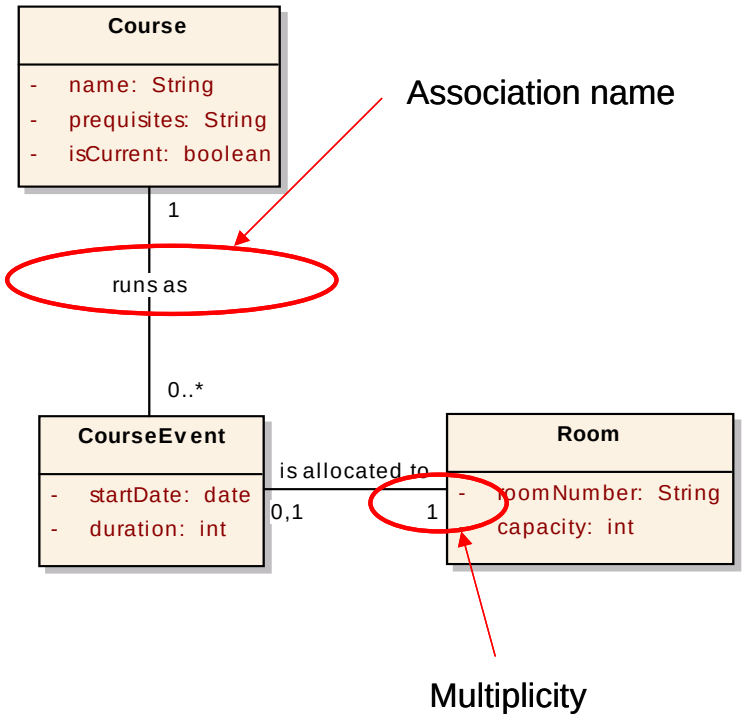


Figure 47 - Example Relationships

Classes that have an association are joined together via a line (called, not surprisingly, an *association*). The association is named - more on this is a moment - and we also add numbers to each end of the line to denote how many instances of each side take part in the association – again, more in a moment.

This diagram is now getting a little technical, but it should still be meaningful and (relatively) understandable to the business – even if we have to guide the client through the diagram.

If we write the diagram carefully, it can be read back as an English statement; the association name and the multiplicity forming part of the sentence.

Starting with the Course class, we foll

"Each course runs as zero or more course events". This makes good sense, and captures a business rule. If it reads back incorrectly, change the diagram accordingly.

Note that the multiplicity alongside "Course" (in this case, 1) plays no role at all here – this comes into play when reading back the association in the other direction.

Let's have a look at reading the association back in the reverse direction. We have a problem here, because the association name is written to make sense in one direction only. When reading back in the reverse direction, we "mentally invert" the association name. It is awkward, but this is done to avoid clogging up the diagram with twice as many association names.

Each. CourseEvent. "is for". One (and only one). Course.

So "each course event is for one and only one course". Again, that seems to make sense. A course event might be "needlework for beginners", but it cannot be more than one course.

The convention for the "normal" way to read the diagram is to move from left to right or top to bottom, and the "mental inversions" are done when reading from right to left or bottom to top.

Reading Direction

By convention, the associations are read "top down" and "left right" - the need to only write for "one direction" cuts down diagram clutter.

The "reading direction arrow" can be used to override the default reading direction. It has NO other meaning. Note however that many tools don't support it; you can use < or >.

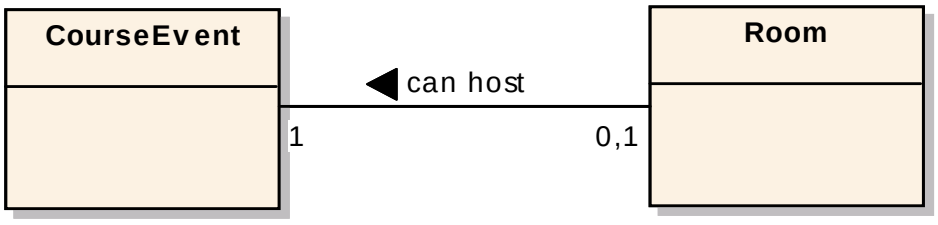


Figure 48 - A reading direction arrow, used to override the default reading direction. Here, each room can horse a single course event

Do NOT use arrowheads on the association link; this has a very specific design meaning which we cannot make a decision on yet.

Multiplicities

The UML uses a very simple notation for capturing the multiplicities. Here is a list of examples which should cover most requirements:

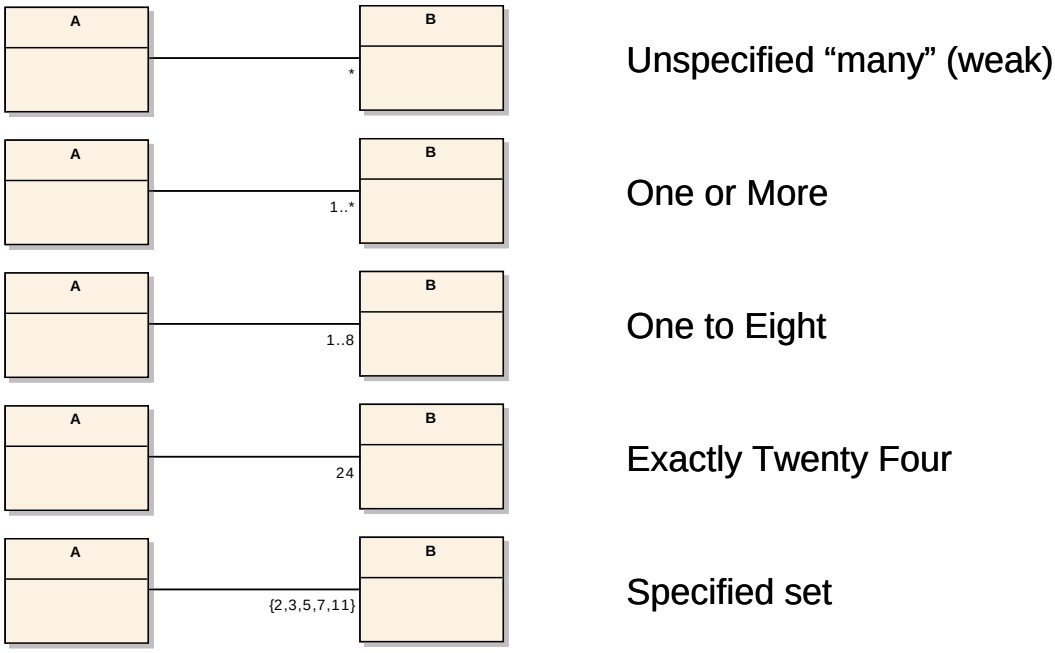


Figure 49 - Example Multiplicities

The Unspecified “Many” is the only odd one; it is often used when the precise figure is not known; it is weak in its meaning and shouldn’t be used when you mean “one or more” or “zero or more”.

Multiple Associations

Sometimes, multiple associations are required. Here is an example of the relationship between tutor and course. We have added the extra requirement that each course is owned by one (and *only one*) tutor, even though a course could be taught by a variety of tutors…

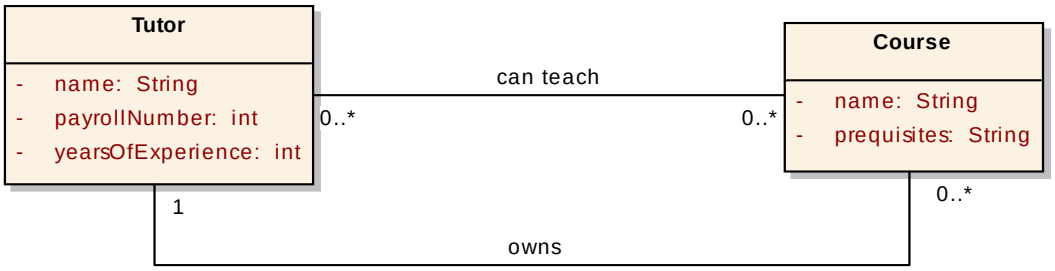


Figure 50 - A Multiple Association

Read back in English, the above diagram states:

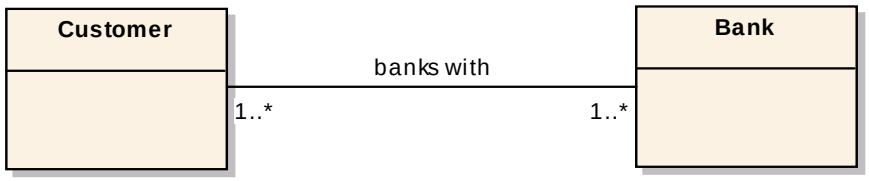
- "Each Tutor can teach zero or more Course(s)"
- "Each Course can be taught by zero or more Tutor(s)"

and…

- "Each Tutor owns zero or more Course(s)"
- "Each Course is owned by one, and only one, Tutor"

Many-to-Many Associations

This is especially for those with extensive data modelling experience. It is surprising to learn that "Many-to-many's" are *perfectly valid* in Domain Modelling – they may cause headaches at implementation time, but that is not our concern in analysis. Consider the following example – is it valid?



"Each Customer banks with one or more banks"
"Each Bank is patronised by one or more customers"

Figure 51 - A many to many relationship – is it valid?

When asking if the diagram is valid or not, it is sufficient to read the diagram back and to ask if it makes sense (from a business point of view). The

phrases “Each Customer banks with one or more banks” and “Each Bank is patronised by one or more customers” are fine and make perfect sense to the business.

It is not implementable in a relational database, but remember *we are not building a relational database!* We are building a model of the business.

However, often there is something missing between two classes that are related in a many-many fashion. In this case, it seems to be Account, as follows:

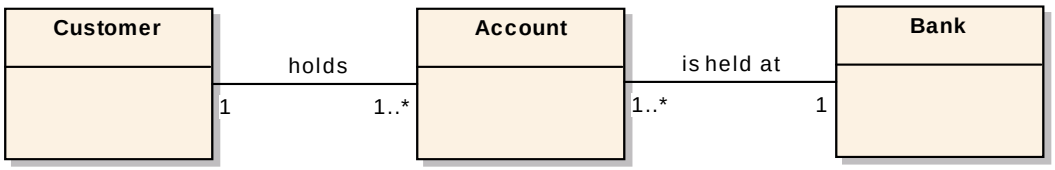


Figure 52 - Adding an Account to the Domain Model

… again, reading the diagram back, it makes perfect sense to the business.

Don’t worry about many-to-many’s, and don’t agonise about getting rid of them, but do check that you haven’t overlooked an “intermediate” class.

Association Classes

Just for the sake of completeness, we’ll mention Association Classes. This is a special symbol used to denote a class that sits in the middle of a many-to-many association, as in our Customer-Account-Bank example.

The following diagram is **equivalent** to the diagram in erfect sense to the business.

Don’t worry about many-to-many’s, and don’t agonise about getting rid of them, but do check that you haven’t overlooked an “intermediate” class.

:

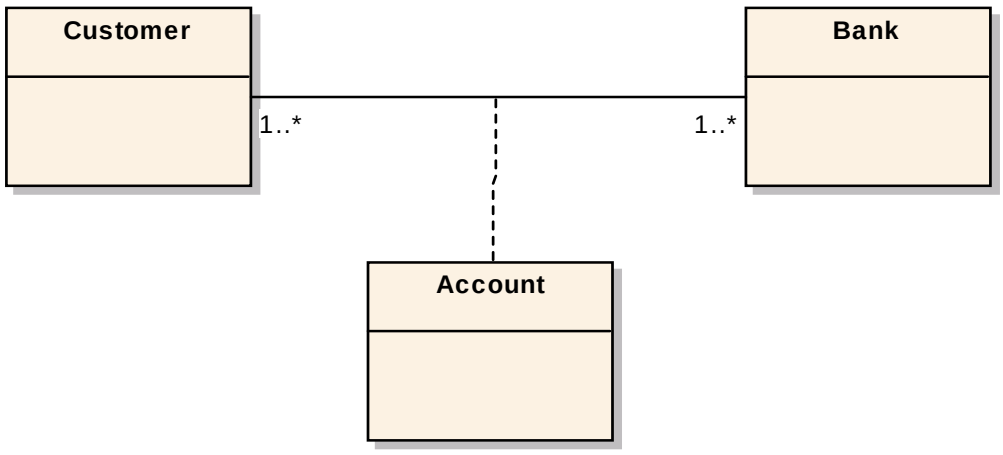
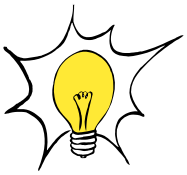


Figure 53 - An Association Class

This is sometimes called the “hanging basket” notation. It may have uses in design, but at the business level it really only serves to obscure the message of the model. We recommend that this notation is avoided, as the diagram in erfect sense to the business.



Don’t worry about many-to-many’s, and don’t agonise about getting rid of them, but do check that you haven’t overlooked an “intermediate” class.

says exactly the same thing, but in much clearer terms.

Building the Model – Approach

Following this 7 step plan to build the model:

1. Read the existing documentation
2. Convene a facilitated workshop with users and developers
3. Brainstorm classes without reservation

4. Qualify and describe classes
5. Determine associations
6. Determine and place attributes
7. Capture and fully document all model components

CRUD Associations

A technique which has been around for decades (especially in Data Modelling) is the construction of CRUD Matrices to check the completeness of our Use Case and Domain Modelling; the general question we are asking here is “have we identified all of the use cases, and do we have enough classes to support them?”

In general, each Use Case is able to Create, Read, Update and/or Delete an instance of the classes we have identified in the data model. For example, the “Create Provisional Booking” Use Case will create a new instance of the “Booking” class.

It might look like a tedious exercise, but it is an illuminating exercise…

Example Matrix

Here we have constructed (using a simple spreadsheet package) a matrix for the Use Cases and Domain Model we have built as the example through this course so far…

	Booking	Delegate	Company	Course Event	Tutor	Course	Location	Room
Create Provisional Booking	C	U						
Create New Customer		C	C					
View Course Schedule	R			R		R		
View Customer Details		R						
Cancel Provisional Booking	D			R				
Confirm Provisional Booking	U							
Confirm Course Event				U				
View Course Event Details	R	R	R	R	R	R	R	R

Figure 54 - CRUD Matrix for the College Management System

Chapter 8

The State Model

In this session, we will:

- Look at the UML State Model and see how it can benefit our Analysis effort
- Define ‘States’ and ‘Events’
- Learn the structure, usage and notation of the state diagram

Capturing More Business Rules

Many classes (and Use Cases) are governed by complex business rules which dictate what changes are permissible and when. For instance, it may not be permissible to allow a ‘withdrawal’ event on an account which is in the state ‘overdrawn’.

It is vitally important that such business rules are agreed with the user, designed into the system and implemented correctly and consistently.

Class State Diagrams

When we create a state diagram for a class we are showing all of the events that can impinge on a class including its creation and deletion. All changes to the state of a class are caused by Use Cases and we can therefore use State Diagrams as cross checks on Use Case completeness.

Events and States

The best way to understand the concepts of states and events is by example. We will use several examples and build up the notation as we go. Let us start with the Registry of Births, Deaths and Marriages…

Births, Deaths, Marriages

Our customer, the Registry Office, wishes to record certain events in people's lives. The Events of interest might be:

- Birth
- Coming of Age
- Marriage
- Divorce
- Widowhood
- Death

States of Interest

We are also interested in the resulting 'state' of a person's record after undergoing each event:

	Event		State
after	Birth	Person becomes	Child
after	Coming of Age	Person becomes	Adult
after	Marriage	Person becomes	Married
after	Divorce	Person becomes	Divorced
after	Death of Spouse	Person becomes	Widowed
after	Death	Person becomes	Deceased

Figure 55 - The events that can happen to a person, and the resulting state they will be in after the event

Capturing State Diagrammatically

The preceding table is not too bad, but adding a few more states and events would make the table impossible to follow

Instead, we can capture these states and events on a picture…

A Life

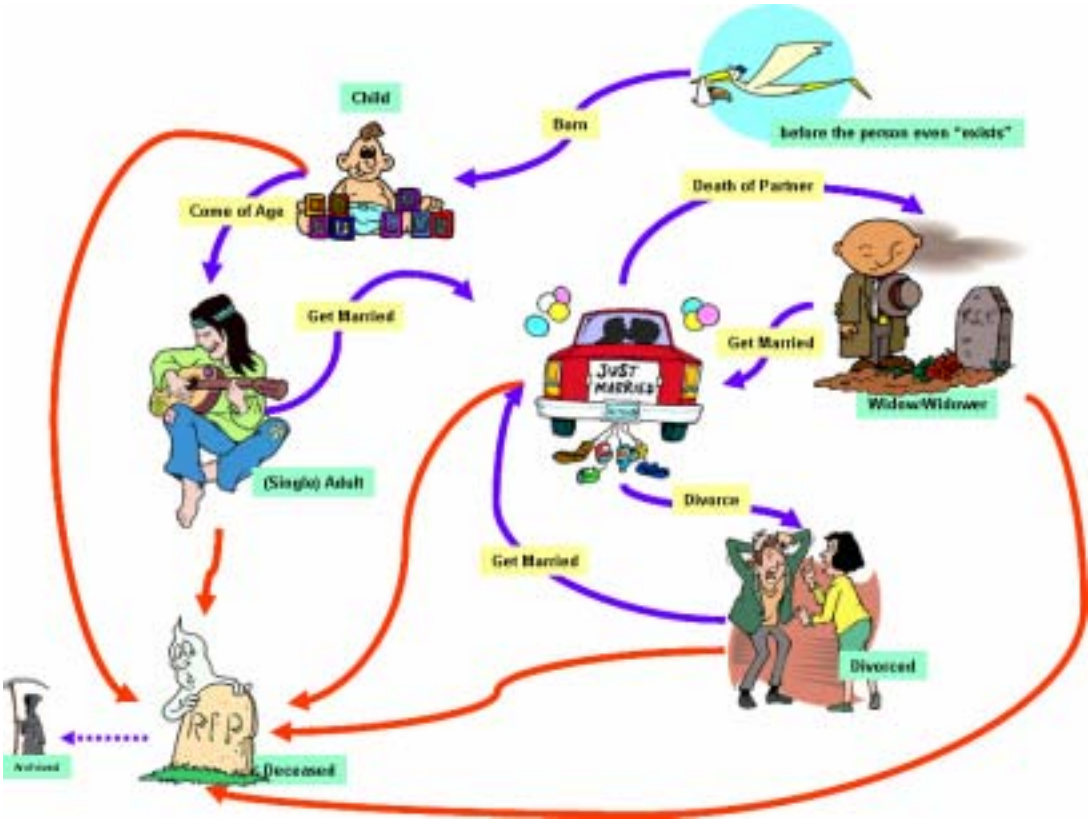


Figure 56 - Drawing the States and Events as a picture

This diagram is fine – it is certainly more readable and understandable than the previous table. The UML has a mechanism for representing states and events – although it is (of course) more formal…

UML State Diagrams

The UML has a very rich notation for modelling state[17]. We will now present the main elements of the state diagram and then use the Registry to show how they fit together…

[17] The UML didn't invent the notation; rather it is an existing notation by David Harel

State Model

Events, States and their transitions are captured in the UML on a State Diagram (sometimes called a State-Transition Diagram or Statechart). The diagram captures the permissible states of an 'entity' during its lifetime, and what events cause the state to change over time.

These entities could be any software artefact or process that has important state (class, database entity, use case, operation...). Our interest is in classes (and this is usually the most common use of a statechart in OO systems) so we concentrate on the states and events pertinent to the single class 'Person'…

A State

A State is shown as a rounded box with the state name enclosed[18].

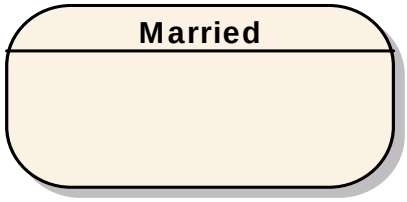


Figure 57 - The "Married" state

A state is normally important to the business being modelled (eg: the state of marriage is important to the Registry). This is important to remember – there is little to be gained from clogging the diagram with lots of states that are meaningless or unimportant to the business. In the example of the registry, a person passing their driving test is totally unimportant – although you could add it to the diagram!

Start State

Almost all statecharts begin with a "Start State". This is the state before the object of the class is created (the state a person is before birth - unknown to the Registry).

[18] Notice that in the following diagram, the CASE tool has added a line across the box; we'll see that extra information can be added to the lower "window" later, but in the absence of extra information this line is optional and your tool might not add it

Figure 58 - The Start State

A start state can have a label or name attached, but often this is omitted as it is implicitly called “start”.

End State

An end state is shown as a ‘target’:

Figure 59 - An end state

It is the state after the object is destroyed - not ‘death’ in the Registry because the person object will still be known to system. Normally there is a period of time which elapses before the person is removed or archived and thus becomes unknown. There isn’t always an end state and the label is once again optional.

Transition

A transition is shown as an arrow between one event and another:

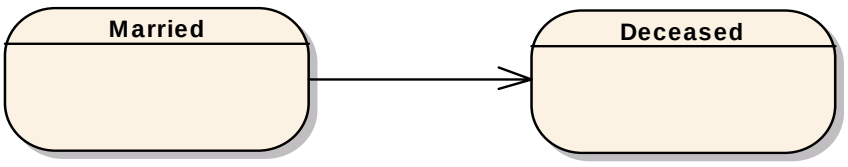


Figure 60 - The Transition from Married to Deceased

The presence of a transition means it is **possible** to move from married to deceased – it might not happen, but it is possible.

There is no transition present moving from Deceased to Married, so this denotes that this transition is **not** allowed in this system.

Event

For a state transition to occur, an event must happen. We capture the name of the event on the transition line. On the statechart, you **must** include the name of the event (otherwise we cannot see why or when the transition would happen).

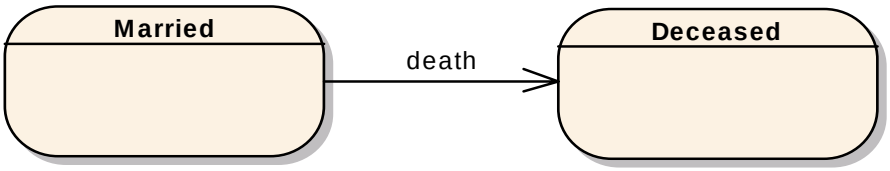


Figure 61 - The Transition from Married to Deceased happens as a result of the "death" event

Events originate via:

- Use Cases - an attempt to create, update or delete an object (Log Birth, Notify Death etc)
- Time Passing (A Person becomes adult) – this is called a **temporal** event

Summary of the Basic Notation

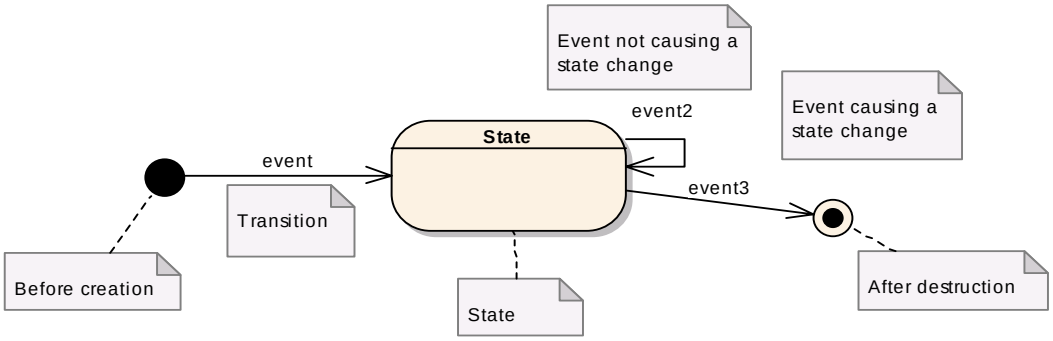


Figure 62 - The Basic Notation Summarised

The only concept we haven't mentioned so far is the event not causing a state change. This is an event that is permissible but doesn't affect the object's state. For example, an event called "change of address" is permissible in the "Married" state, but it doesn't cause the state to change.

Be careful not to overdo this type of event; remember to only capture events that are interesting to the business. It may well be the case that the Registry is not interested in changes of address – in which case it would merely clutter the diagram.

The 'Person' State Diagram

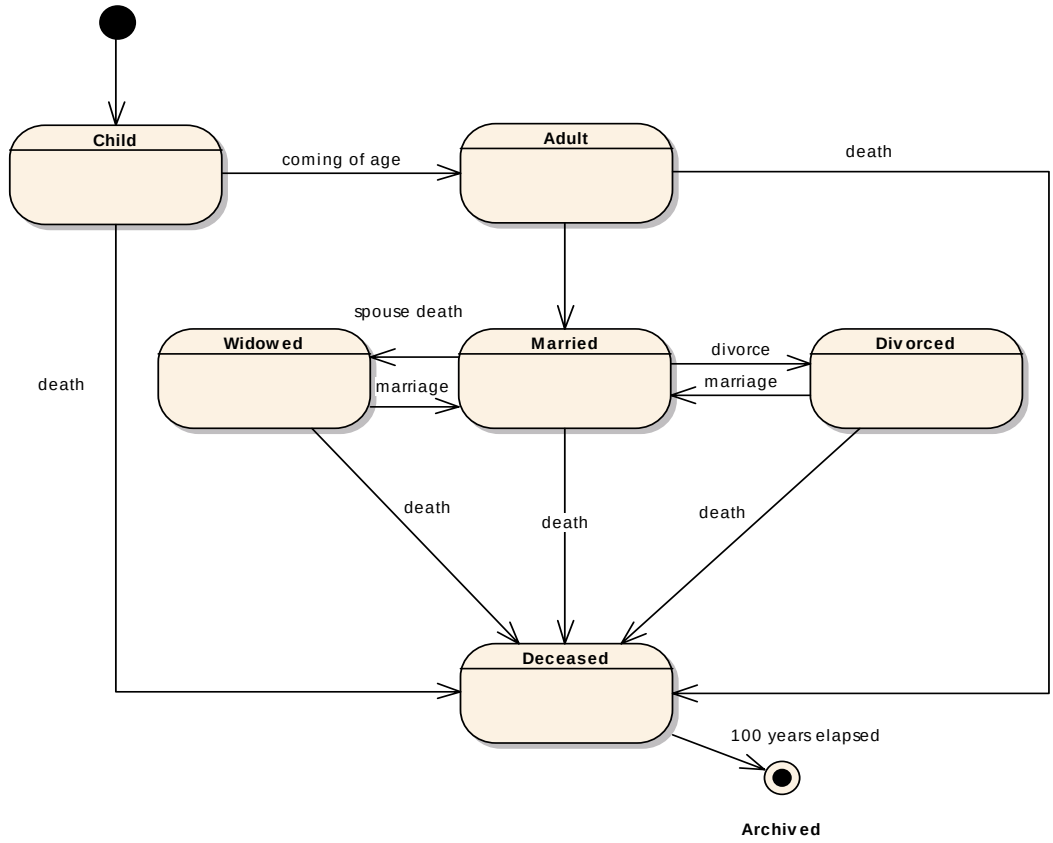


Figure 63 - The Person's statechart, redrawn using the UML

Business Rules and State

The state diagram captures **Business Rules.** It determines the constraints on data modification and (as we shall see) actions that must be carried out as a result of attempts to modify data.

Using the registry example, a person:

- Can't do anything until born
- Can't become an adult unless a child
- Becomes an adult at 18
- Can't get married unless an adult, divorced or widowed
- Can't divorce or become widowed unless married

These rules must be implemented in the eventual design and code.

More Notation

Let's use a very simple and familiar scenario in order to explore more state notation. The scenario we will consider is as follows:

- A **TrafficLight** class is designed to record the state of real traffic lights in a complex city centre traffic light control system

- A TrafficLight object should correctly record the traffic light being created (becoming part of the system), turned on, turned off, cycled and reset…

TrafficLight State Diagram

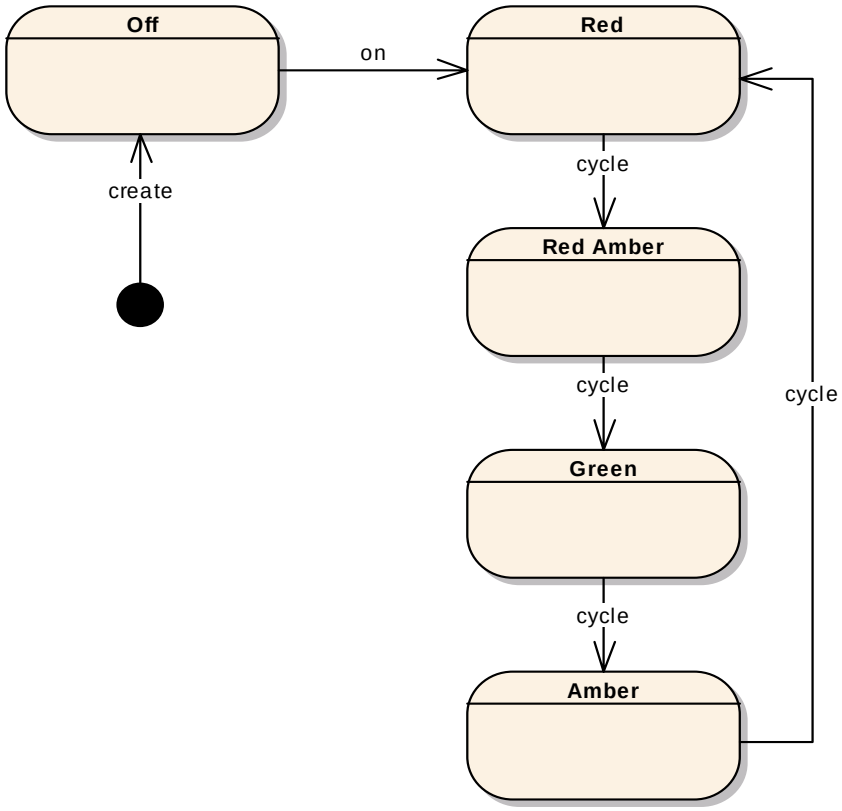


Figure 64 - First attempt at the Traffic Light State Diagram

Extending the Traffic Light

So far, so good but we still have some way to go:

- We have no ‘Turn Off’ Event
- We have no ‘Reset’ event to set the lights immediately to Red in an emergency
- We have no ‘Destroy’ Event

There is a simple way to achieve the Turn Off event but it’s not very elegant…

Turn Off Event - Version 1

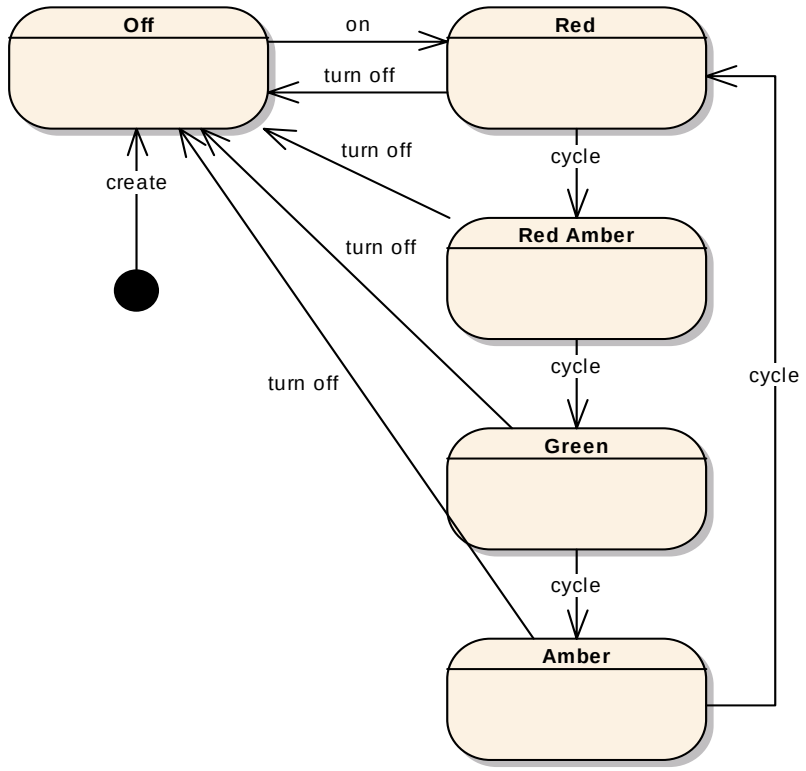


Figure 65 - First attempt at adding the Turn Off event

This is very inelegant and messy (it would be worse if we had more than four states to switch off).

There must be a better way? Well, if you look at the diagram, we really have two classes of state in play here. We have the overall state of the light – this can be “On” or “Off”. But if the state of the light is “On”, it can be in four further **sub**states – Red, RedAmber, Green and Amber…

Turn Off Event with Sub-States

The notation for super states and sub states is as follows:

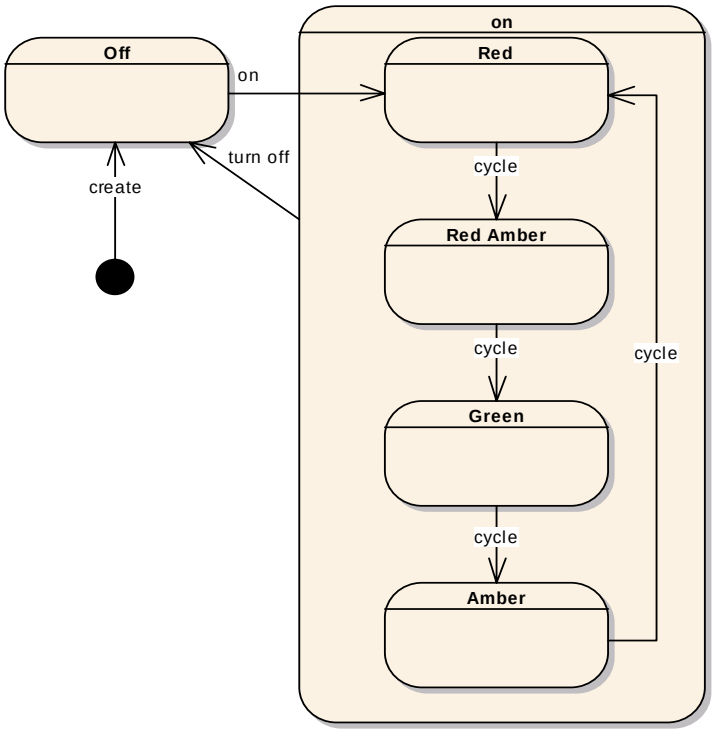


Figure 66 - Using Super and Sub states to tidy the diagram

Here, when the light is turned on, it moves to the “On” superstate, and at the same time it assumes the “Red” substate.

The light can be turned off at *any time* while it is in the “on” state, regardless of the colour of the light.

Reset Event

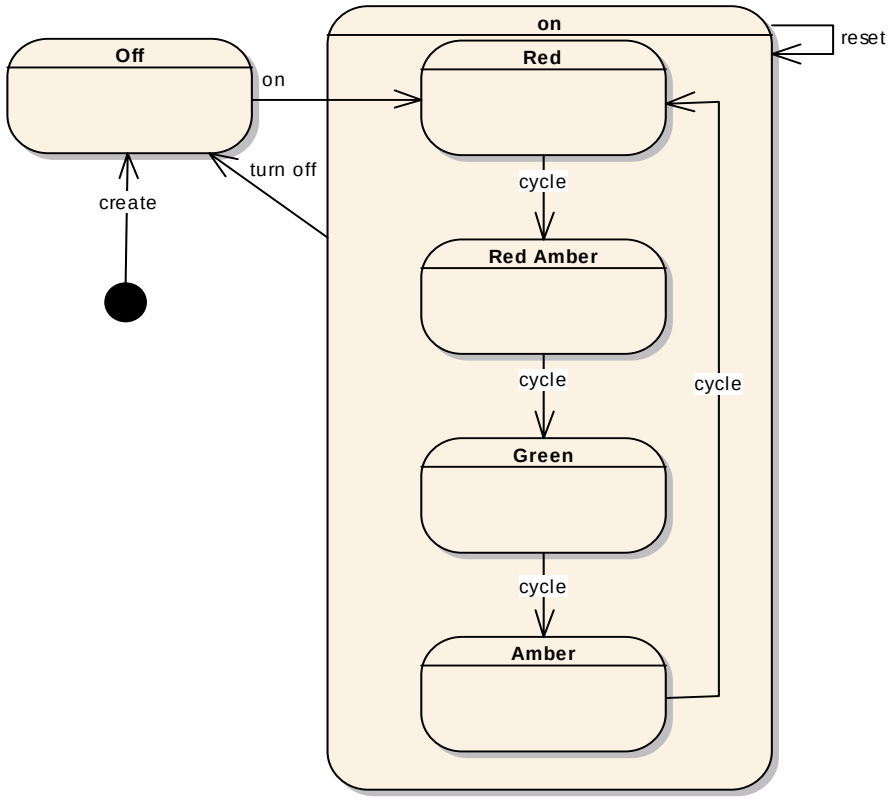


Figure 67 - A reset event added

Here we have added the “reset” event, which can happen at any time; the sub state will resume from its default state, Red.

Revised Person Statechart

Now we can return to our person Statechart (which we last saw in and State

The state diagram capt) and tidy up the diagram using sub states. The problem with the chart was the proliferation of “death” events…

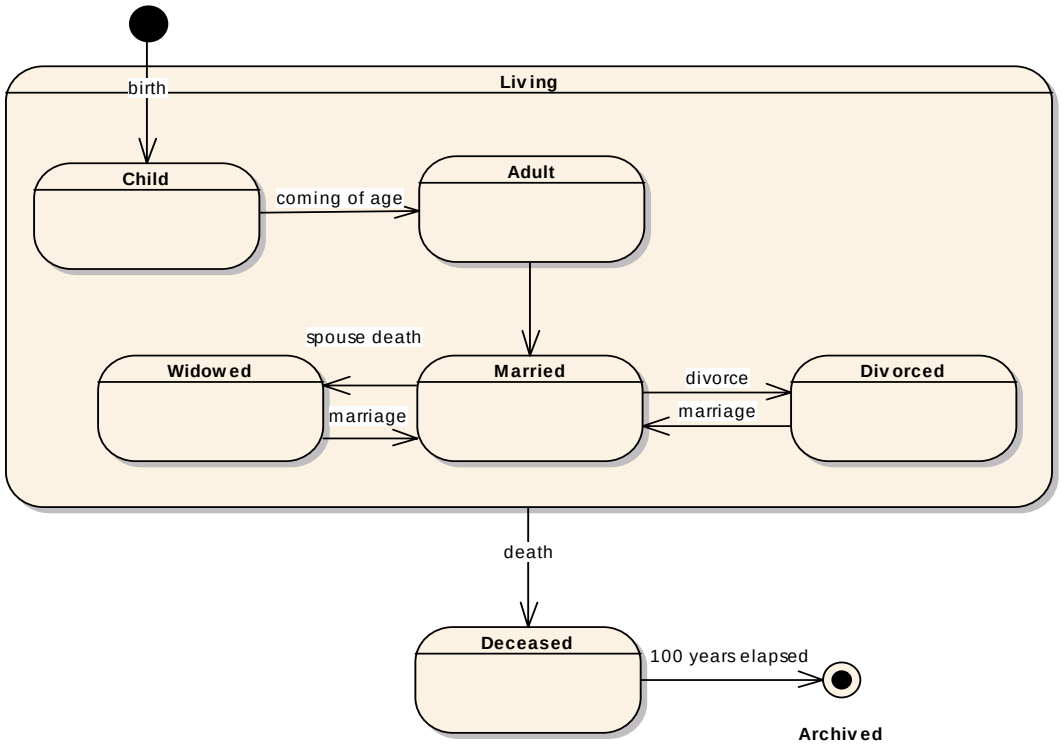


Figure 68 – Person Statechart redrawn using a "Living" superstate

Conditional Transition

Sometimes transitions are only permissible under certain circumstances. To notate this, we can use a *guard condition.* For example, imagine that in a certain country it is not permissible to get married until a year after the divorce is finalized:

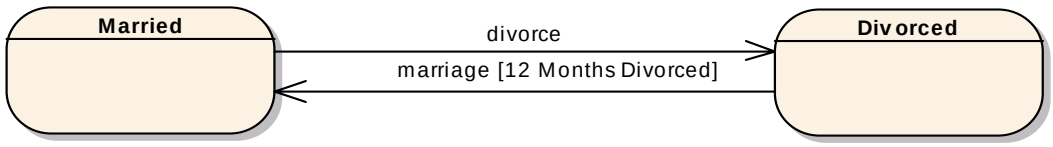


Figure 69 - Adding a Guard Condition

This diagram can be read as "*if the marriage event occurs AND the guard condition '12 months divorced' is also true*". Notice that the guard condition is written inside square brackets.

Actions

When a event occurs, it is often useful to initiate some action on entry or exit from a state or on the transition

Here the birth event triggers an action to issue a birth certificate…

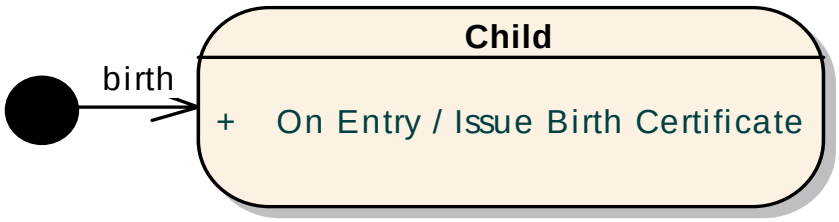


Figure 70 - Entry Action

Event Sources

It should be obvious from this session that objects respond to events and that some objects have rich and complex rules governing their response. Every event on the state diagram originates from a Use Case so check every event and ensure you can trace it to a Use Case. If you can't you have probably missed some Use Cases.

Which Classes Have State Diagrams?

Not every class will need a state diagram, but every class should be treated suspiciously. Justify why you are not going to create a state diagram for the entity – you should always have a good reason.

Summary

In this session we covered:

- Definition of states and events
- The structure, usage and notation of the state diagram
- How the state model captures business rules

Chapter 9

Ranking Use Cases

Ranking and Estimation

Recall that Use Cases are the unit of development (and the unit of estimation)

All iterations (both elaboration and construction phase) are planned according to the 'rank' of the uses cases. The higher the rank, the earlier the use cases are scheduled.

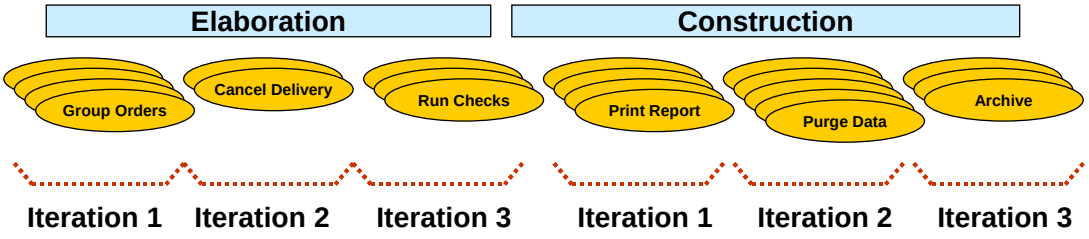


Figure 71 - Iterations are planned using Use Cases

High Ranking Use Cases

We cannot give you hard and fast rules on how to rank your use cases; this will be a decision your project will have to make using judgement and experience. However, high ranking Use Cases are likely to be those that:

- Exercise the Architecture – ie those are going to cross a lot of the software architecture (using many of the classes identified), or Use Cases that will make heavy demands on your hardware architecture (eg a Use Case that updates thousands of records a second is an excellent use case to derisk your database implementation)

- Exercise many and/or complex business rules

- Involve new technology, need research, are time-critical or very complex
- Are very significant to the business

In short - use cases that tackle RISK - although it is also fine for “Quick Wins” to appear early to build confidence.

Chapter 10

Specifying Use Cases

In this session, we will:

- Take our basic, bare bones Use Cases and build them into more formalised, detailed Use Cases
- We will determine the pre-conditions, post-conditions, main, alternate and exception flows

Note that this work is very detailed and we would usually only formally develop the Use Cases scheduled for the particular iteration we are in.

A real problem in writing a UML course is that the UML does not define in any way how to produce detailed and formal descriptions of Use Cases. However, a handful of approaches are common in the industry, and we'll try to provide a flavour of each one in this session. Remember though, that none of this is set in stone, and it is up to your project to use the approach that suits you best.

Why Specify Use Cases?

Most use cases, and especially those that will be built in the early iterations, are complex/important enough to require a formal description. The description gives us the detail behind the requirements to build the use case.

This specification of the use case isn't actually design; it is providing more detailed requirements in advance of the design and build stage.

So how does the UML define how to perform this specification?

UML Definition of Specification

There is none! The heavily tool-influenced UML Spec ignores any issues that need to be addressed with text. Conventions have, however, sprung up throughout the industry – but as is often the case with ad-hoc solutions, some organisations write Use Case specifications very poorly (some don't write them at all).

We will follow an approach distilled from our own experience; we also lean heavily on Alistair Cockburn's work. If you need to set project standards, his book (reference [9]) is an excellent start.

Use Cases vs Requirements

It should be remembered that **Use Cases are Requirements** - they are, in fact, **all** of your **functional** requirements.

Use Cases aren't all of your requirements though (Cockburn suggests about a third of them):

- Performance
- Availability, Reliability and Maintainability
- I/O Protocols
- Data Formats
- Business Rules
- UI Requirements

Where Use Cases Fit In

Use Cases can be thought of as being at the heart of the requirements; they hold the whole system together - but the Use Case Descriptions themselves cannot specify everything. Alistair Cockburns "Hub and Spoke" model of requirements reflects this:

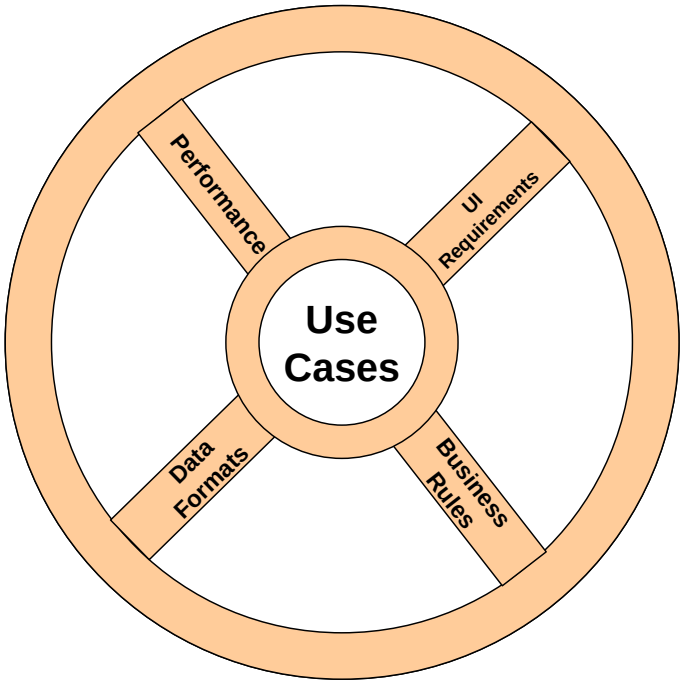


Figure 72 - Cockburn's Hub and Spoke Model

Key Information Required

The short description contained Name, Intent, Description, Requirements Cross Reference and Outstanding Issues.

We will need to add at least the following information in our formalised use case:

- Pre and post conditions
- Main and Extension Flows
- If necessary, descriptions (sketches or prototypes) of the GUI

We'll look first at pre and post conditions and provide a working definition of them – but heed the warning : *your real life project may use a different working definition that changes the emphasis of the conditions*

Pre Conditions

Preconditions state what must be true before the use case can even begin. These preconditions are not tested within the use case; they are assumed to be true before the use case starts. In other words, it is a statement about what has already happened in other use cases. Try to avoid trivial pre conditions, such as “the system is on”!

Good example:

The user has logged in and authenticated as a system administrator

Post Conditions

Post conditions are declarations about the state of the system on completion of the Use Case. They detail how the system was changed:

- What entities have been created and/or deleted
- What attributes have been modified
- What associations have been created and/or deleted

Note: They describe WHAT changed not how the change was achieved.

1. An attendee was removed from a Course Event

Figure 73 - Example Post Condition

Use Case Main Flow

The main flow in the text document describes the normal sequence of events that takes place between the actor and the system (the “80%” flow). It is concerned only with the user actions and the system’s response to those actions.

Main Flow

Cancel Booking Use Case Description

Short Description – *as before*

Requirements covered : R71.2; R72 (all); R12 (part covered)

Other Use Cases Impacted : Cancel Course

Actor : Training Administrator

Pre-Conditions :

1. The training administrator has successfully logged in

Post-Conditions:

1. An attendee was removed from the Course Event

Success Criteria:

1. The delegate has a booking against the course event they have quoted
2. The booking is in the status confirmed
3. The corresponding course is in the status “scheduled”

Main Flow:

1. The Actor selects to cancel a course booking

2. The System displays a list of available Course Codes

3. The Actor selects the Course Code required

4. ….etc

Figure 74 - An example textual main flow

An alternative style is to use two columns, with the actors actions on the left and the system response on the right (see Larman [2]).

Extension Flows

Extension flows are any other scenarios that form the use case that vary from the main flow (you may see the terms “Alternate Flow” and “Exception Flow”; these refer to roughly the same idea).

Main Flow:

1 The Actor selects to cancel a course booking

2 The System displays a list of available Course Codes

3 The Actor selects the Course Code required

….etc

13 The booking has its status set to “cancelled”

14 A message is displayed indicating the booking has been cancelled

Extension Flows

8a No bookings are found for the selected course event. [use case ends]

13a The course is no longer viable; Trigger the “Cancel Course” Use Case

13b A Message is displayed to the user warning that the course has now been cancelled [return to main flow]

Figure 75 - Example Extension Flows

The style suggested by Cockburn is to number the extensions to match the Main Flow step they follow on from. This, as Cockburn admits, is difficult to maintain – but then alternatives such as paragraph numbering (as suggested by the RUP) is just as bad. Perhaps soon the CASE tools will catch on to this?

Style Guidelines

- Use Simple Grammar – “Subject, Verb, Object, Prepositional Phrase”.

  “The System removes the Delegate from the nominated course”

- Always say if the action is from the system or the actor

- Write from a third party point of view

  Ie “get the account balance and deduct from the total” is not clear style

- Keep each step fairly coarse grained

  “Enter name, address, town, county and postcode” would be better written as “the actor enters their name and address”

- Show the intent of the actor, and not how they should achieve it on the interface - the job of designing an interface to support the Use Case should follow on next. Embedding GUI steps into the storyboard makes for clutter, and poor GUI design.

- There is nothing to stop you referencing GUI prototypes from your Use Case document, but keep your scenario clear of GUI

CRUD Use Cases

You will nearly always encounter Use Cases that manage a particular Object – such as for a Booking object, you may expect to see Create Booking, Read Booking, Update Booking and Delete Booking Use Cases.

Should we roll these four Use Cases into a single Use Case (Manage Booking)?

The answer is that there is no firm industry opinion on this.

Our suggestion is to perhaps use “Manage Booking” if the Booking has no interesting state chart, and no complex business rules are present. In this example, we feel that Delete Booking (really : Cancel Booking) is a fairly serious business case and warrants a Use Case of its own.

Graphical Form

Arguably, the textual form of the Use Case Description is the most common in the real world. However, as the UML is supposed to be graphical, we can capture the flow information in the form of a UML diagram. To do this, we’ll pull in the “general purpose/catch all” diagram, the Activity Diagram…

Example Activity Diagram

This is a simple example, but it covers all of the syntax of the Activity Diagram…

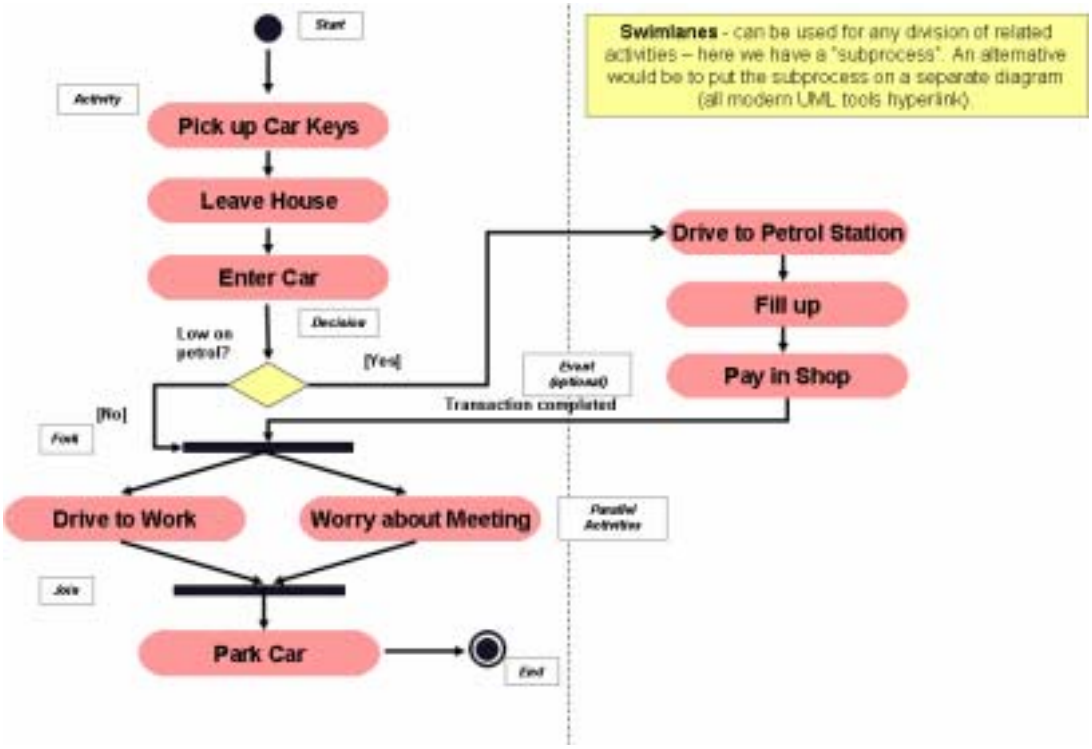
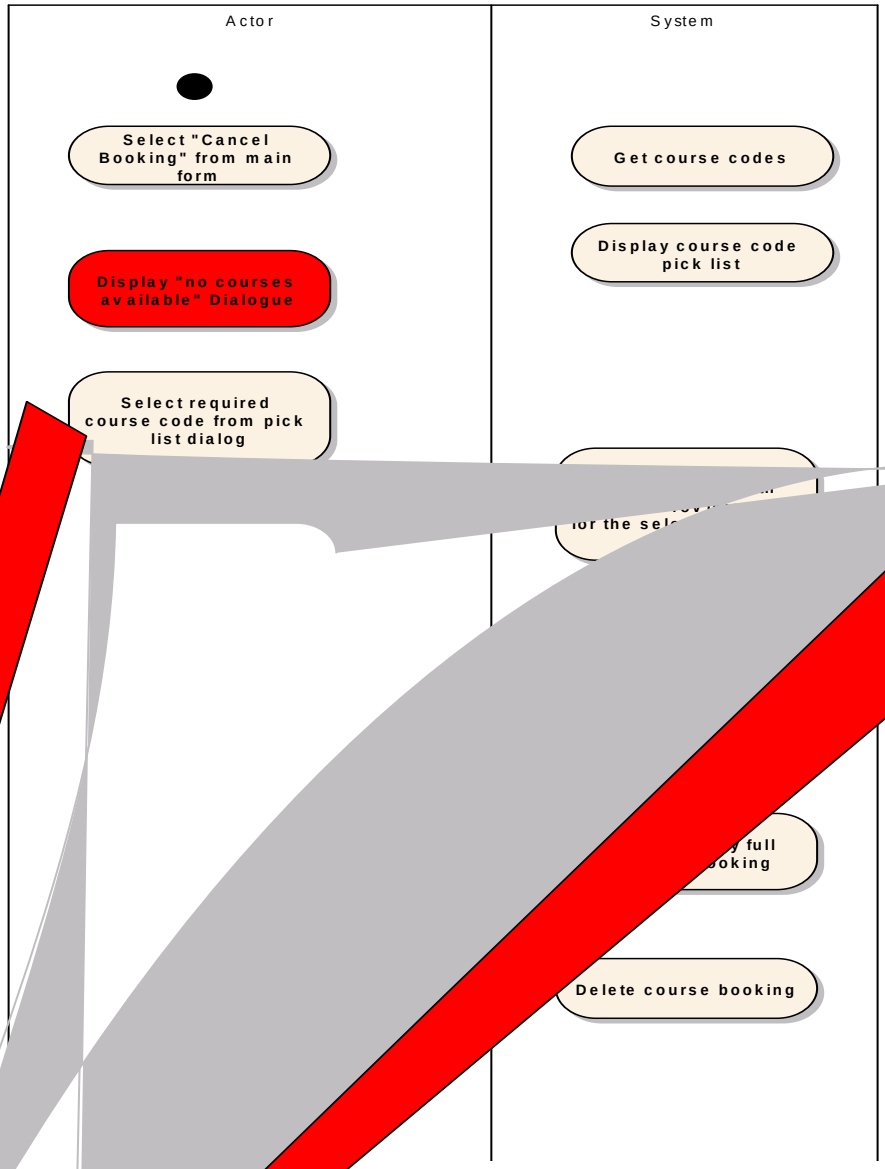


Figure 76 - Activities Performed by a boring businessman on his way to work

Now let's build a Use Case storyboard using the activity diagram…

Use Case “Storyboard”



- Determining pre-conditions, post-conditions, triggers, main, alternate and exception flows
- Drawing Standard Storyboards
- Producing Textual Descriptions

Chapter 11

Interaction Modelling

At last, we can now begin the Object Oriented Design. This process is supported in the UML through two closely related diagrams:

- The Collaboration Diagram
- The Sequence Diagram

Both do the same job; it is really up to you which one you work with. We'll start with the Collaboration Diagram and consider the Sequence Diagram later on.

Transition to Detailed Design

During inception we concentrated on WHAT had to be done and there was little use of OO techniques. During elaboration we have extended the analysis and have tried to lay a solid architectural foundation.

We often talk about "realizing" Use Cases. This essentially means that we have to look at each Use Case in turn and decide how the objects we have identified are going to make the Use Case happen. So for the first time, we will be adding methods to our class diagram.

So, we now need to determine how the architectural elements interact to realise the use cases. Here, there is great emphasis on three concepts: **Objects** (which we've already studied), **Responsibility** and **Collaboration**.

Responsibility and Collaboration

We *could* implement the behaviour for a single Use Case inside a single method in a single class. This would be easy, but it would lead to huge, bloated methods inside classes. It would mean the data required for the Use Case would sit inside "dumb" objects that only hold data and don't actually do anything useful.

In other words, we would be designing a system in the structured/functional way.

Instead, in Object Orientation, we have to look at the objects we have identified and decide:

- What each object should be able to *do* – this is called **Responsibility**
- How the objects should "work together" to make the Use Case happen – this is called **Collaboration**

Many real world situations need a variety of objects and people to work together to get something done. Consider the following simple, real world example…

Simple Real Life Example

We are going to forget about IT systems for a short while and think about how a real-world Use Case would run. Let's look at a library, and consider one of their Use Cases (or business processes); in this case "Borrow Named Book".

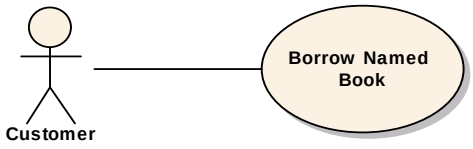


Figure 78 - Real World Use Case

By this Use Case, we mean the non-trivial case where a customer approaches the librarian's desk and asks for a particular book but doesn't know where it is in the library.

We first consider the "Domain Classes", or the objects that are available to take part in this Use Case. In this particular library, there is a librarian in charge, and working for the librarian is a group of Library Assistants.

Capturing this on a class diagram, we have:

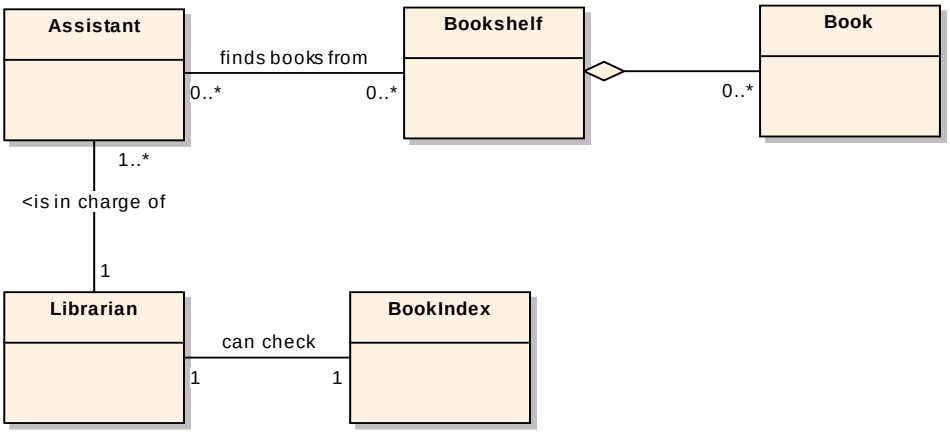


Figure 79 - Domain Diagram for the Library

We are now going to work out how we can get the objects to work together to make this Use Case happen. We'll capture our thoughts using clip art rather than real UML. We'll do the UML later, but bear in mind that the thought process we are following here is identical to the thought process we carry out in real OO design…

The Collaboration Sequence (step 1)

First of all, the customer walks into the library. He **collaborates** with the librarian, and asks for a particular book. We can capture this collaboration as follows:

The Collaboration Sequence (step 2)

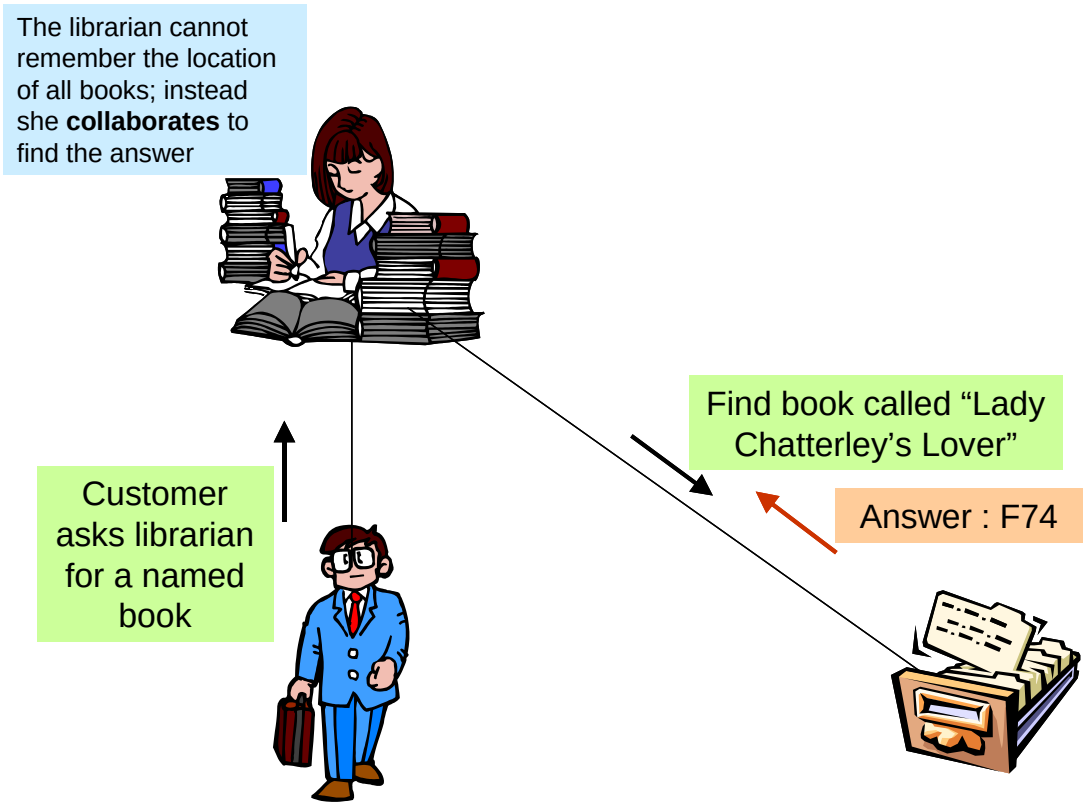


Figure 81 – Second collaboration.

The librarian asks the book index where the required book is. Note that although the book index is an inanimate object, we can still think of the index “doing” things and responding to requests from other objects. We do this often in OO design, where objects like Purchase Orders are able to do things.

Anyway, the book index has the required information to do this job, so it can easily return the answer: in this case, shelf F74.

So, the librarian now knows where the shelf is. But as we said, we don’t want her scurrying around the library, so she can delegate the work and collaborate with another object. The Library Assistant looks like a good choice here (part of their job description is to retrieve books for customers)…

The Collaboration Sequence (step 3)

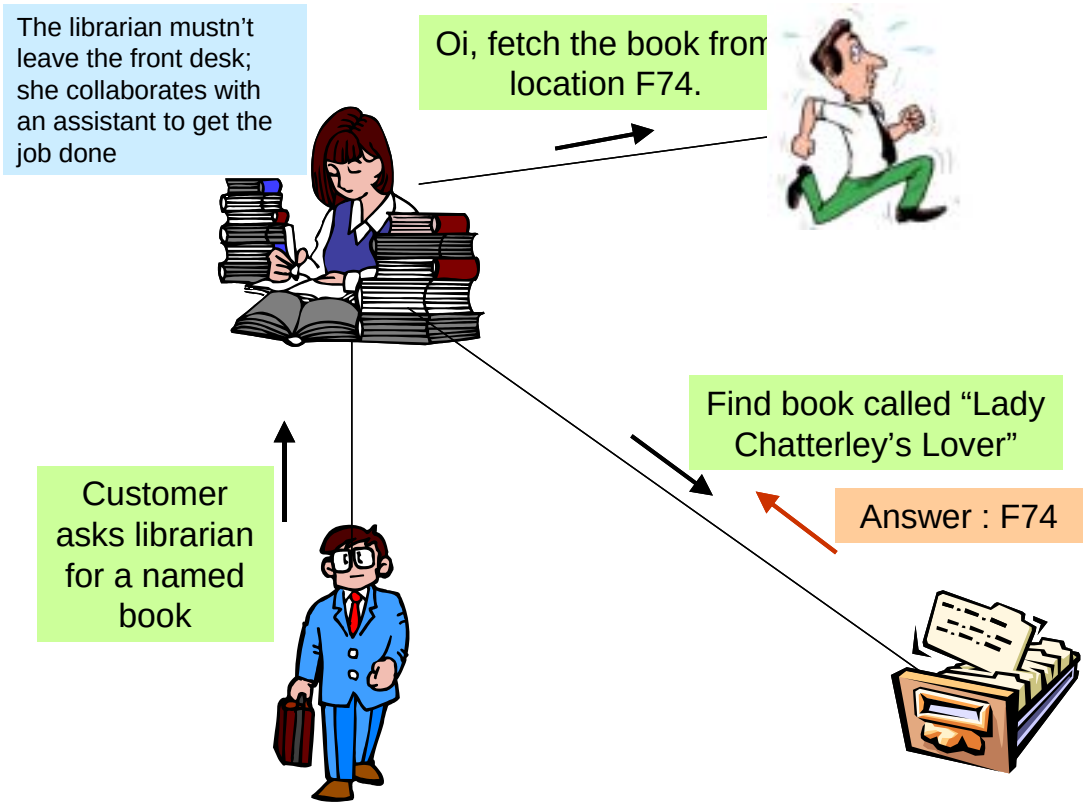


Figure 82 - Step 3 of the collaboration; the library assistant now joins in

The library assistant is told to get the required book, and they are told which shelf to get the book from. They then run off and fulfil the responsibility.

The Collaboration Sequence (step 4)

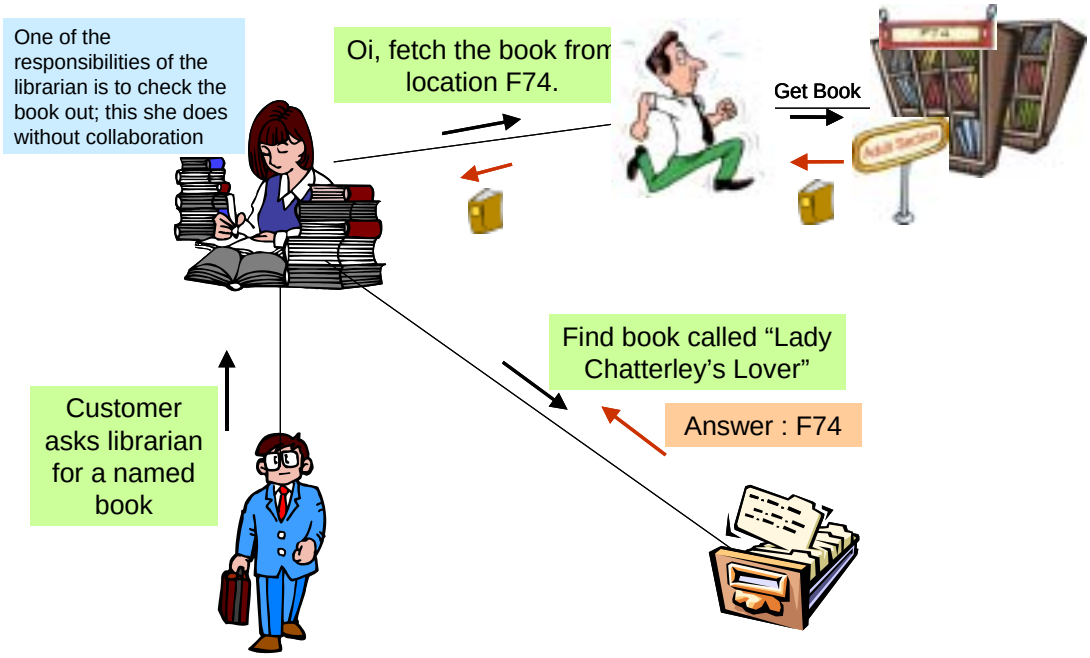


Figure 83 - The book is found and returned

Once the book has been found, it can be returned back to the librarian (note that the diagram shows the book being “handed back” in a relay fashion back from the bookshelves).

According to the Use Case Storyboard, once the book has been retrieved, it must be checked out. This is one of the jobs of the librarian, so she can do this without further collaboration.

The Collaboration Sequence (step 5)

Customer asks librarian for a named book

Objects

Objects are denoted by a box with a name of the object inside. The colon and underlining means “Object” rather than “Class”.

Figure 85 - Two Objects on the Collaboration Diagram

Method Calls

When one object needs to call a method in a second object, we connect the two objects with a line. Then add the method you need on top.

Figure 86 - Adding a message. The "Assistant" class is calling the method called “getBook” in the Bookshelf class

Note : if you add more messages later, no need to draw an extra association line; just add the messages.

Parameters

Parameters are indicated in brackets. The type of the parameter can be shown (here String, but it could be any type of data). Simply comma separate multiple parameters.

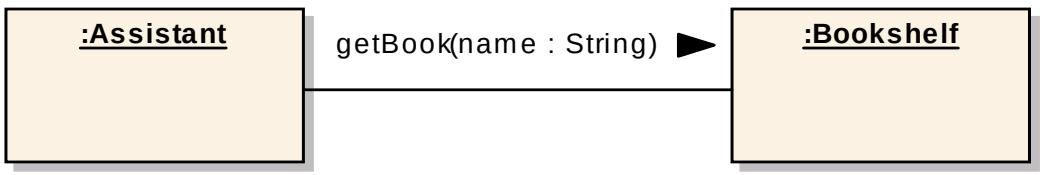


Figure 87 - Passing parameters with a message

Return Values

A message can return a value, in the style of a traditional function call. Any syntax can be used, but this is standard UML; use this style if you wish to remain language independent:

```
return := message (parameter : parameterType) : returnType
```

However, how the return type is denoted tends to vary from Case tool to Case tool; in ours it is denoted as follows:

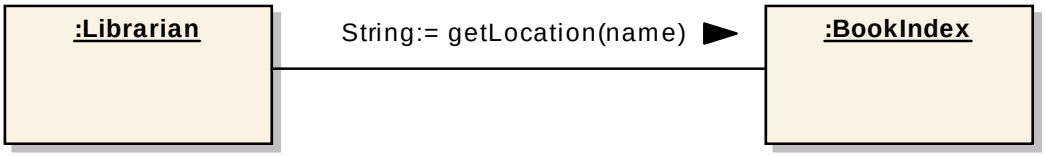


Figure 88 - Showing a return type

Note – we don't show an arrow "pointing back" to denote a message that can return. This annoys some people, but we assume it was done to cut down on clutter.

Looping Messages

Note : this slide has been removed from the course as it is not part of UML2.0; however you will see it used in existing designs, so we have left it in the book just in case…

If you wish to send the same message to a collection of objects from the same class, use the "Multiple Instance" notation (we call this the "deck of cards"):

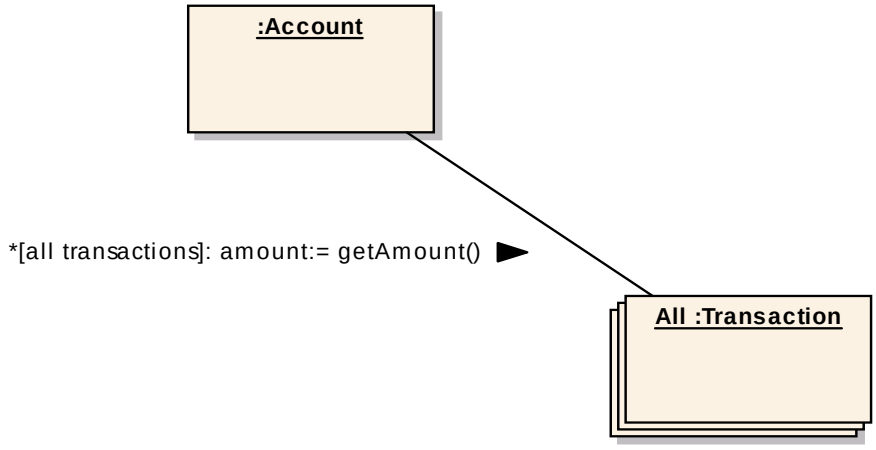


Figure 89 - Calling multiple objects from the same class

Notice our use of the word “all” in the figure above. This isn’t required, we just do this to give the diagram a little more emphasis.

Creating Objects

Objects can be created using the following syntax:

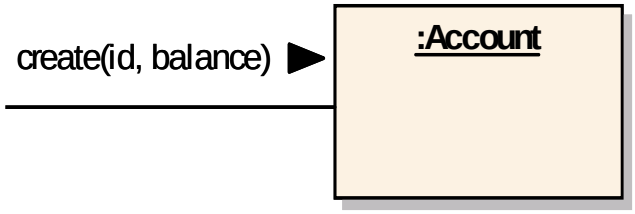


Figure 90 - Creating a new instance of an "account" object

Conditions

Read square brackets in UML as an “if”. In the following example, we are saying “If new account, then send this message”:

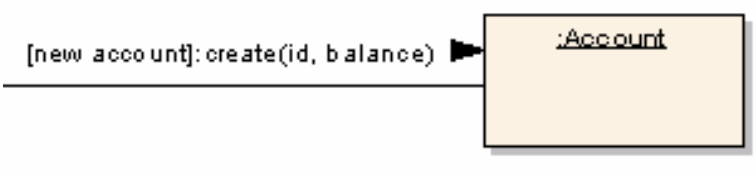


Figure 91 - A conditional message

Full Worked Example

We'll work through the Collaboration for a simple Use Case from a Share Tracking Application. As designer, you have the following tools at your disposal:

- The full Use Case Description, giving you full details of the flow of the Use Case
- The Class Diagram, giving you the classes and their relationships

Keep these two artefacts "in view" and the collaboration diagram is much easier to build.

Please note that this work is very hard to do and this example may scare you at first; try not to panic. Often it takes several iterations to get a collaboration diagram right; you'll find it much easier once you've done a practical exercise…

The GUI?

We haven't referenced GUI's at all so far - we have concentrated on the business' Domain Classes. As we move to design – the question has to be answered: where *does* the user interface fit in?

We could, if we wanted, simply create some GUI 'classes' (eg Visual Basic Forms, or a Java Swing GUI) which could do "the work". In the next figure, we're showing a real life example of this. We have written, in Java (using a GUI builder) an interface with a button that allows the user to cancel a Purchase Order (based on an ID number they have typed into a text field). The code takes the number and issues a database command to delete the Purchase Order from the database…

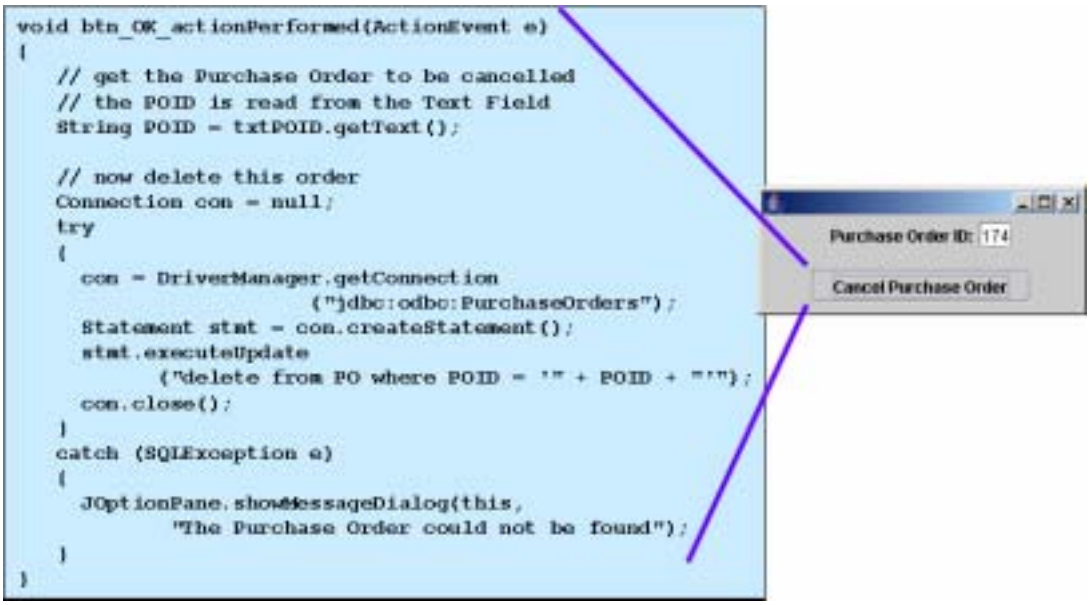


Figure 92 - Embedding code "behind" GUI buttons

This approach is incredibly common, largely because rapid GUI development tools (such as Visual Basic, Delphi, or any of the modern Java environments) encourage you to do so – you usually just double click on the button in the designer, and then you're dropped straight into a skeleton method, and away you go!

A much better approach is to separate the GUI classes from the Domain Classes – then either can be varied independently. In fact this is a general principal in design:

Identify aspects of your system that might vary independently, and isolate them.

In most systems, the GUI and the Domain classes are two major areas that will vary independently, so a wise architecture is to separate the two: in other words, have classes dedicated to the GUI and classes dedicated to the Domain.

This architectural approach to building systems is called "Model/View Separation".[19]

[19] Note: in earlier versions of this book, we referred to this as the "MVC". Unfortunately there is a fair bit of confusion about the true nature of MVC, and the classic Smalltalk MVC is quite different from simple Model/View Separation. We have decided to swerve the debate and be more neutral in our terminology.

The Class Diagram and Use Case Description

Here is the class diagram and Use Case Description; remember we need to keep these on hand to see what objects are available for us to use and for what needs to be done.

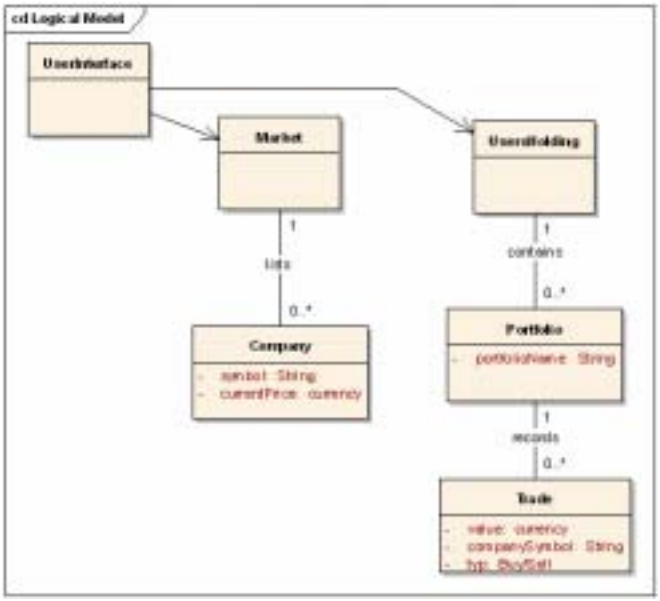


Figure 93 - The Class Diagram Fragment

Buy Shares Use Case Description

1. The actor wishes to buy shares : they select "Buy Shares" from the menu
2. The system responds by asking them:
   - The symbol of the share they want to buy (eg BAE)
   - The number of shares they want to buy
   - The name of the portfolio they want to store the details of the Trade in
3. The user enters their requirements
4. The system responds by informing them the trade was successful, and tells them the price that the trade was struck at

Postcondition: A new Trade was created in the appropriate portfolio

Extension Flows:

3a [the symbol entered was invalid] An error message is displayed

Figure 94 - Buy Shares Use Case Description

Building the Diagram – Step 1

We start by looking carefully at the storyboard for the Use Case. Clearly the GUI needs to gather the symbol, the number of shares required and the name of the portfolio they want to hold the trade in.

This means that steps 1, 2 and 3 of the Use Case Description can all be implemented by the User Interface class. This means no collaborations, and we therefore do not need to denote this on our design.

We find though, that denoting the user's interactions with the GUI is an excellent way to kick the diagram off. The actor isn't formally part of the collaboration diagram.

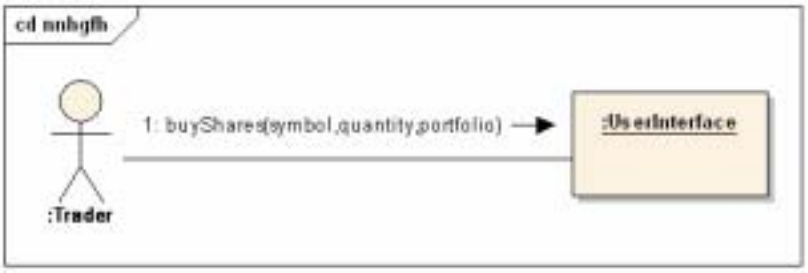


Figure 95 - Starting the Diagram

What next? This isn't easy. Time to make a design decision. First of all, we have to ask the Market class if we can buy the share requested by the user (we need to know the price of the share, and if there are shares available in the selected company…)

Continuing the Diagram

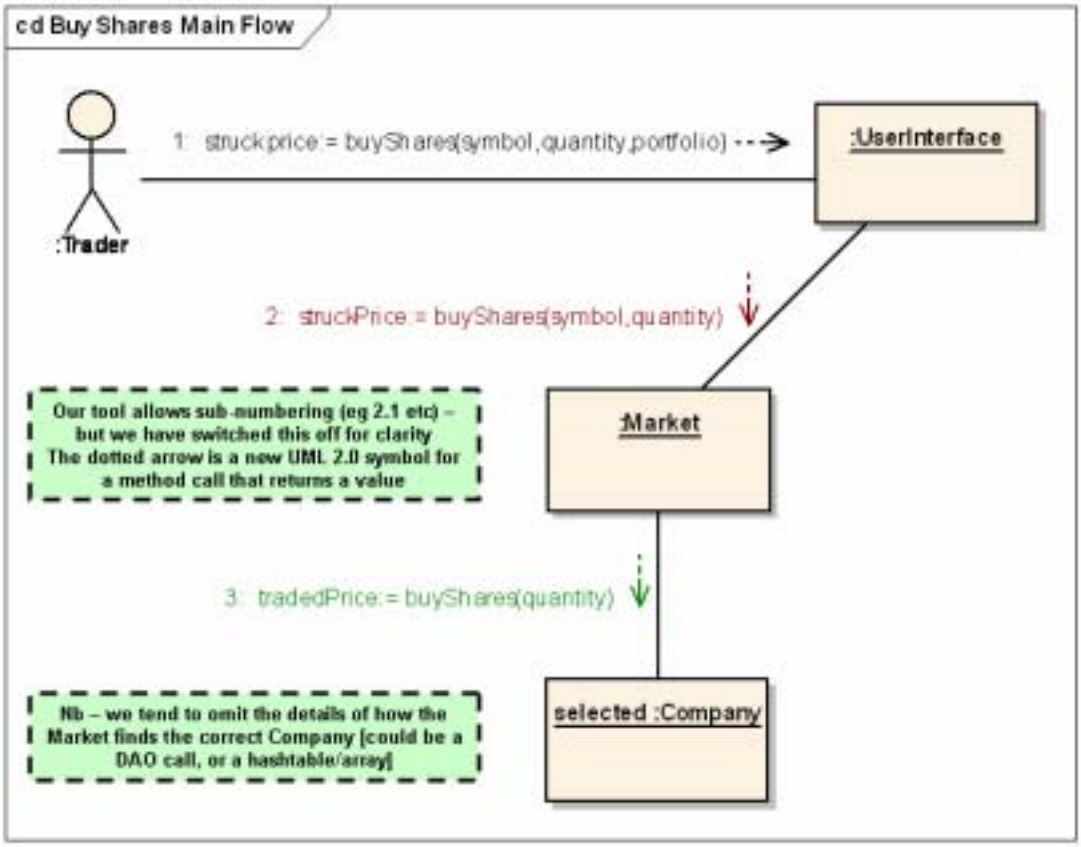


Figure 96 - The next steps on the diagram

Finishing the Diagram

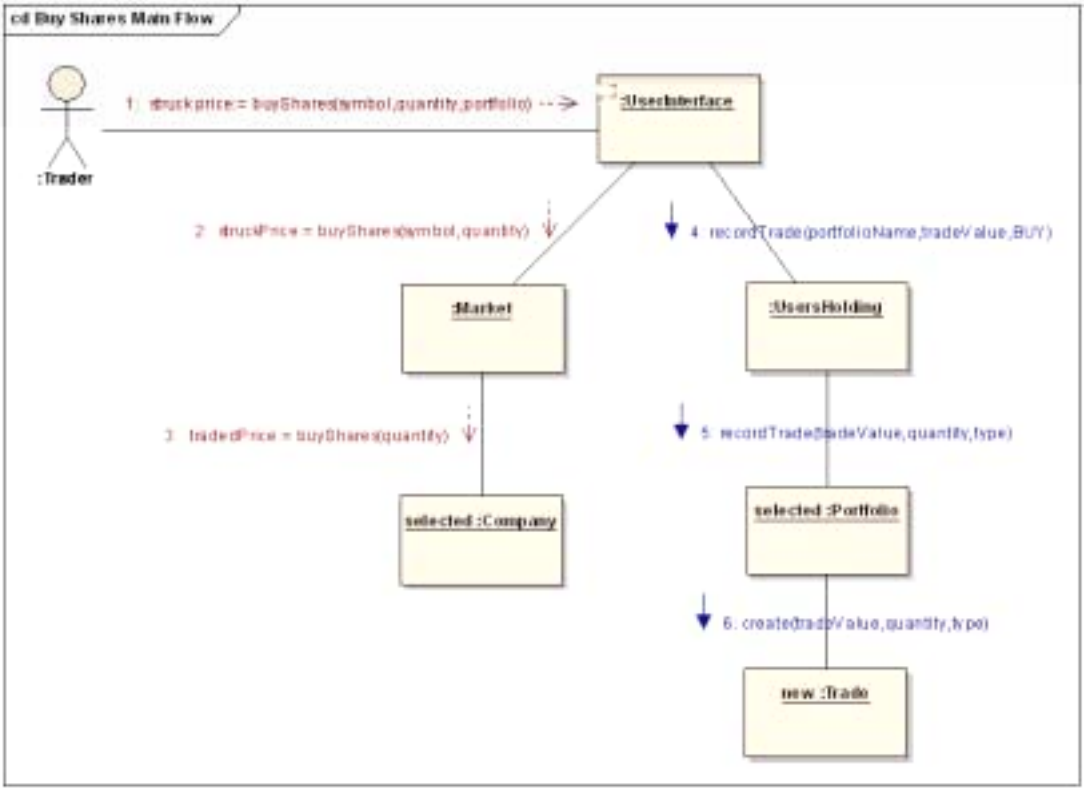


Figure 97 - the finished Collaboration Diagram

Uncovered Methods

All this is hard work, but it is worth remembering that we have just broken the back of the object oriented design. The methods we have uncovered can now be added to the class diagram. For example, due to message 5, we now know that we need a method called “recordTrade()” in the portfolio class.

If you are lucky, your Case tool will add the methods for you automatically as you build the collaboration diagram. Ours has, and this is the result…

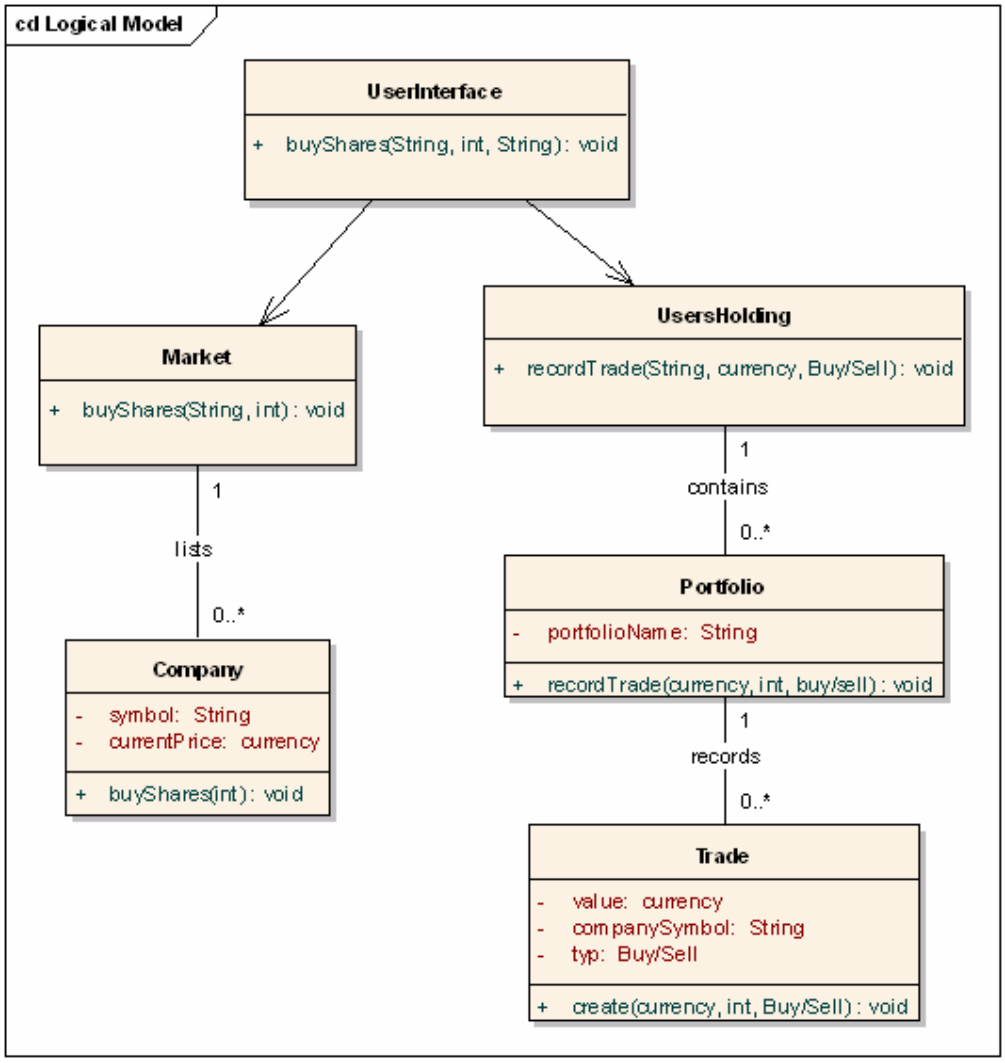


Figure 98 - The completed class diagram

Association Direction

You may also add arrow heads to your class diagram to indicate the direction in which messages are sent (if the messages go in both directions [see later], then for some reason in the UML you leave the arrowheads off)

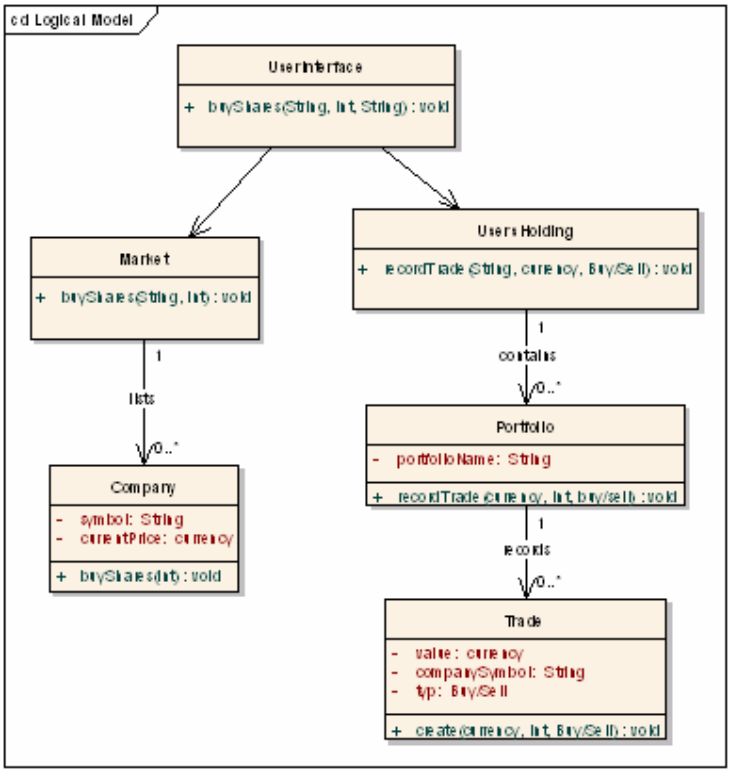


Figure 99 - Adding arrowheads to the Class Diagram; this is denoting that the Portfolio needs visibility of the Trade class in order to send messages to it

The Alternate Flows

After method 3, we had an exception flow – the share symbol was not found. Do we need to include this on the Collaboration Diagram?

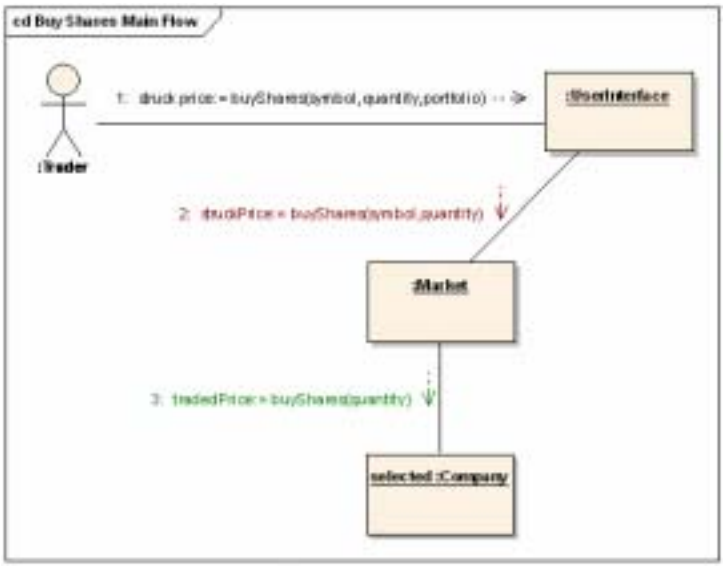


Figure 100 - Should we include the logic for the symbol not being found?

The answer is “probably not”. A lot of practitioners criticise the collaboration for not being particularly good at capturing flow logic like this – but this is missing the point of the diagram…

Only Model “Interesting” Flows

The collaboration diagram is for modelling responsibilities and collaborations. In this case, the collaboration would simply terminate (with an error message) after method 3, so no extra responsibilities (methods) are added.

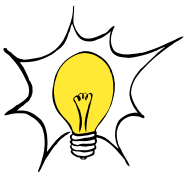
It **would** have been interesting if we had to perform some special operations, such as “unwinding” any changes or triggering a different Use Case. In this case, it is usually better to build a separate diagram.

[In this specific case, the condition would be handled by exception handling, which once again doesn’t fit with the style of this diagram]. A definite criticism of the diagram is that it isn’t complete and accurate, so it doesn’t always form a good “handover” document for a coder. It does, however, help the OO thought process.

Collaboration : Guidelines

The key rule is to keep them as simple as possible. If the diagram gets complicated, break it down into separate diagrams.

In particular, avoid classes that do too much work, or don't communicate with other classes, or have too many association links.



A spaghetti of links suggests high coupling this means that even small changes to classes can have dramatic impact on other classes in your design. Some coupling is inevitable in any system; try to minimize it as much as possible.

Only use the association links identified on the Domain Model – think VERY carefully about adding new ones.

Summary

Interaction Modelling is the core of Object Oriented Design

Remember, we are not designing algorithms or flows through Use Cases; we are assigning the correct methods to the correct classes

Two alternative diagrams are available

- Collaboration Diagram
- Sequence Diagram (coming later…)

Building them is hard work; but it is the core activity in OO Design

All non-trivial Use Cases require an interaction diagram

Chapter 12

Polymorphism, Inheritance and Composition

Inheritance in UML

Often, several classes that you design may share several characteristics. It can be beneficial to factor out the common attributes and operations. We can arrange these classes into what is called an “Inheritance Hierarchy”.

Consider the following classes (which presumably come from a personnel staff management system):

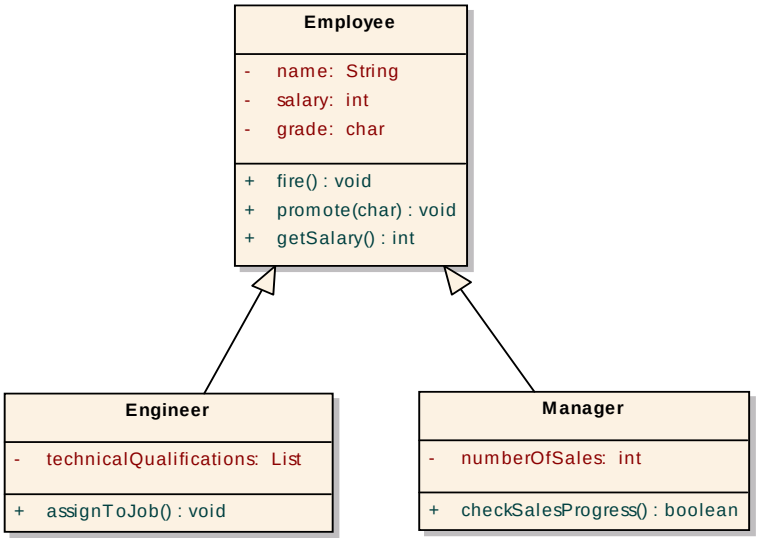


Figure 101 - UML Inheritance otation

Notice that we have taken out the common methods and attributes from Engineer and Manager. In an OO language, the coder of Engineer and Manager only needs to add in the extras required.

In OO, the more general class (here called “Employee”) is often called a *Base Class*, whilst the more specific classes (“Engineer” and “Manager”) are often

called *Derived Classes.* However the terms “Parent/Child” or “Superclass/Subclass” are often used for Base and Derived Classes respectively.

Protected Methods

We have seen that a minus sign denotes that a method or attribute is *private*, whilst a plus sign denotes *public*. Relating to inheritance, there is a third level of visibility, called *protected:*

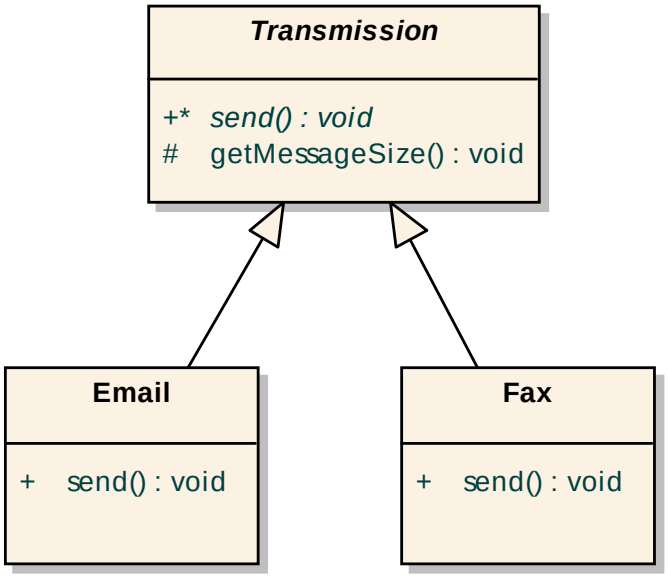


Figure 102 - A Protected Method

Private methods are accessible to derived classes. In the figure above, we want the getMessageSize() method to be private, because it is used as a “helper method” from within the Transmission class. However, we also want the method to be accessed from both the Email and Fax classes, but *from no other classes.*

Protected methods allow us to achieve this, and the symbol for a protected method in the UML is a hash symbol[20] (#).

[20] Pound symbol in the US

Summary of Visibility Levels

- Private
 # Protected
+ Public

To summarise the three levels of visibility:

- denotes ***private*** and means that the method or attribute can only be accessed from the class in which it is defined. This should always be used for attributes, and should be considered the "default" option for methods, although you will usually need to relax the visibility level for methods. Private methods are only used within the class they are definied and are often called "helper" methods.

# denotes ***protected*** and means that the method or attribute can only be accessed from the class in which it is defined *plus all subclasses of the class*[21]. Once again, it shouldn't be used for attributes (a method should be used to access the attribute).

+ denotes ***public*** and means that the method or attribute can be accessed by any other class.

In .NET, the concept of a *property* is just a private attribute with accessor and/or setter methods that read or modify the value of the attribute. The only difference is that the method is called automatically when a programmer tries to directly read or modify the concepts of the attribute.

There is no special notation in UML to denote read/write properties, although some projects use stereotypes as follows:

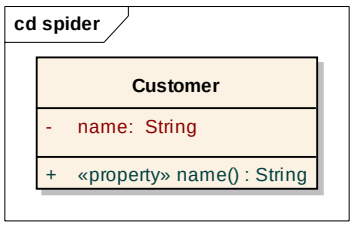


Figure 103 - Using the <<property>> stereotype to denote a .NET property

[21] Java uses a slightly different definition : a protected method can also be accessed from other classes in the same package!

Inheritance => Coupling

By the very nature of inheritance, a base class and its derived classes are quite tightly “coupled”; this means that a change in one class could easily affect the other classes.

When looking at simple hierarchies, it is easy to underestimate the severity of this problem. In the real world, you might be working with trees of classes dozens of levels deep, and the “roots” of the tree may well have hundreds of classes dependent on them.

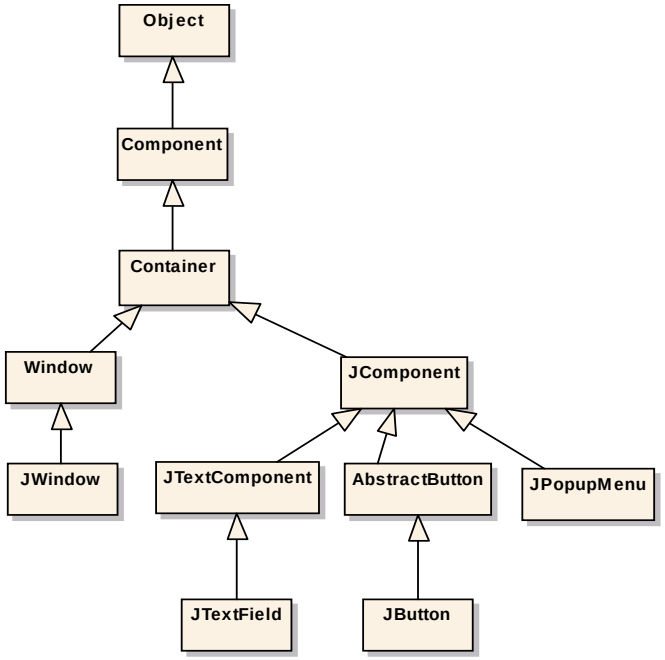


Figure 104 - A *tiny* fragment of a real world inheritance hierarchy

In this heavily abridged example (taken from the Java GUI library), the Component Class acts as the root of the entire graphical library. A change to any method in this class will potentially have a shattering impact on the rest of the structure.

The Java GUI library is well designed, and the classes at the top of the tree are fairly stable – but remember that if you go wild with inheritance in your design, you face a serious maintenance problem.

It is also worth remembering that humans aren’t particularly good at dealing with more than 7 or 8 things at the same time – so if you’re working with a class at the foot of a deep tree, you could find the complexity overwhelming.

Having said all this, a couple of simple rules – when applied carefully – help us to avoid invalid inheritance…

The 100% Rule

The 100% Rule says that all of the base class' definition must also apply to the derived class - if this rule does not apply, you are breaking the definition of inheritance and you are setting yourself up for some serious maintenance problems.

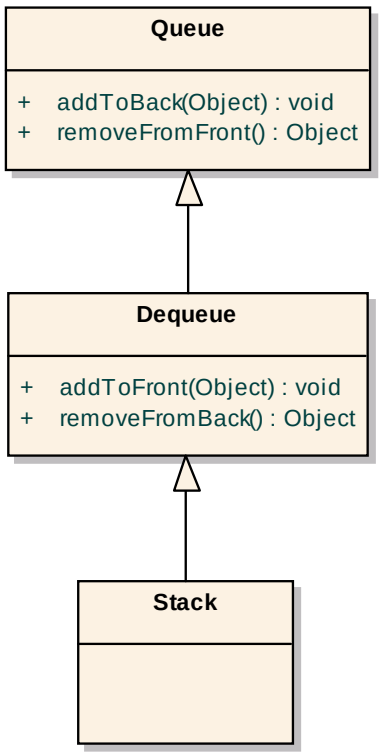


Figure 105 - Lazily Building a Stack from a Dequeue

Here a Dequeue (a "Double Ended Queue", pronounced "Deck" – a Queue where items can be added and removed from both the front and back) has been built from an existing Queue class. This isn't too bad, as all of the methods in Queue are also appropriate to a Dequeue.

The blunder is in our construction of the "Stack" class. Stacks are collections where items can only be added or removed to the front (they are usually used where efficiency is needed). The developer here has had to do no further work – but they have a Stack class that is crippled with two major problems:

1. It has too many methods (both addToBack and removeFromBack are redundant)

2. The methods that are required are badly named – addToFront should be called Push and removeFromFront should be called Pop (or Pull – these names are traditional for Stacks)

Some people may claim that both problems can be solved:

1. Simply override the two redundant methods with “do nothing” stubs

2. Add two new methods in the Stack class called Pop and Push that go on to call the existing addToFront and removeFromFront methods.

These two “solutions” are not solutions – they are bodges. If we did this, our stack classes would now have two methods that don’t do what they say they’re going to do, and two sets of “duplicated” methods (both sets of which are publicly exposed).

Classes that don’t do what they say they are going to do, and badly named or duplicated methods don’t always seem like a problem when they are created – but they soon become a problem when the project becomes hundreds of classes that are confusing and difficult to understand![22]

The “Is A Kind Of” Rule

Correct inheritance should also pass this simple rule. Insert the phrase “is a kind of” in the middle of the names for your derived class and base class. Does it make sense? If so, the test has passed. If the phrase is nonsense, the test has failed.

We have been dealing with “Engineer” and “Employee” throughout this chapter. Let’s try the test:

“An Engineer is a kind of Employee”.

That sounds ok. In our system, an engineer is indeed a special kind of employee (and so is a manager).

Let’s take a different example. We are designing a hierarchy to model a PC computer system. We are looking at how to add **Laptop Computer** into the model. We cleverly spot that we already have a class called “Operating System” with the methods bootUp(), openWindow() and so on.

[22] It is interesting to note that in the original Java Collections (pre 1.2), the Stack class was indeed inherited from Vector. It was in recognition of design blunders like this that motivated the construction of a whole new framework for 1.2 onwards.

Clearly, a Laptop Computer needs to do these things too. So let's use inheritance to save some work…

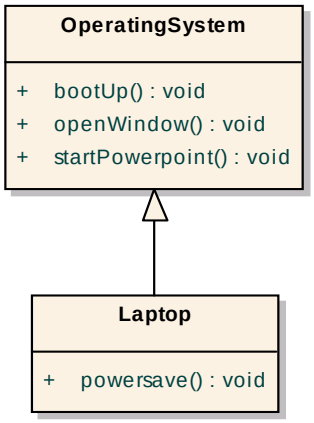


Figure 106 - Is this a valid use of inheritance?

Notice that we've also added the extra method "powersave()", which we need for laptops only.

Is this valid? We would argue not:

"A Laptop is a kind of Operating System".

This sounds like nonsense. A Laptop isn't a kind of operating system! The two are related, but a laptop certainly isn't a special type of operating system.

We'll do an exercise at the end of this session that deals with this problem again – but in the meantime perhaps you could think about what the *real* relationship is here.

Now we've established *when* to do inheritance, lets move on and look at some more advanced concepts…

Overriding Methods

Now imagine we have an extra requirement for our staff management system - a Manager must now receive a £100 bonus for every sale they have made, as part of their salary.

In OO, we can use a technique called "overriding" – this is the replacement of the *implementation* of an inherited method, even though the *signature* of the method remains the same (the signature is the parameter list for method takes, and the type of value the method returns).

So, in our example, we still need a method called getSalary() in all classes, but we need to implement it differently in the Manager class. Let's see this in UML:

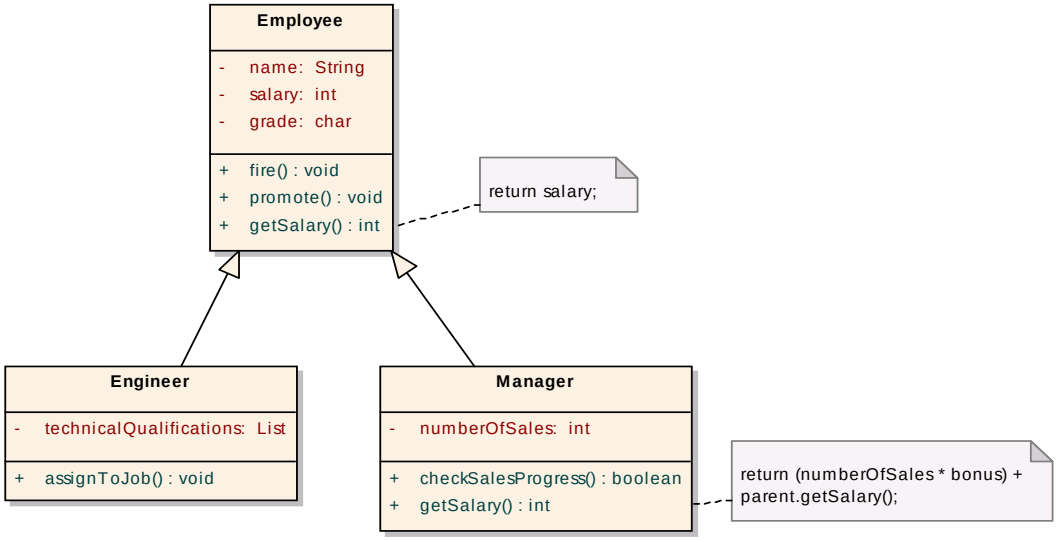


Figure 107 - Overriding getSalary() in Manager

We've used a UML *note* to show an example implementation of the methods. Note that the getSalary() method is inherited, without change, in the Engineer class.

Abstract Methods and Classes

In any OO language, we can leave some methods *unimplemented*, and defer their implementation to the derived classes. We call these unimplemented methods *Abstract*.

Unfortunately, the notation for an abstract method in the UML is rather clumsy (in our opinion); the method is written in *italics*.

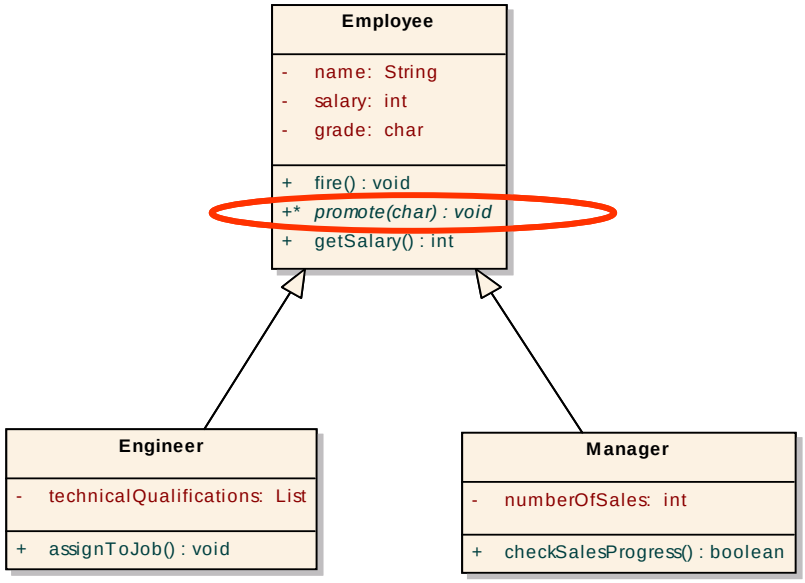


Figure 108 - Making promote() abstract

Any class that has at least one abstract method is itself called an *Abstract Class*. A programmer cannot create an instance of an Abstract Class; therefore we have avoided the problem of a programmer calling the unimplemented promote() method.

By the way, a class which is **not** abstract is called **Concrete**. So, in our example, the programmers will implement promote() in both Engineer and Manager, and therefore those classes are concrete.

Polymorphism

All modern object oriented languages apply the following rule:

Any method that requires a parameter of a base class can also accept ANY derived class (even if the derived class is many levels down).

This is often referred to as the “Liskov Substitutability Principle” or LSP, named after Barbara Liskov, Professor of Computer Science at Massachusetts Institute of Technology. It can be difficult to grasp at first – let’s look at a simple example:

Example

A trading system needs to automatically confirm trades with both the trader and the counterparty.

This can be done either via **SWIFT messages** (an automated form of Telex used by the banking industry[23]), or if for some reason SWIFT is not available or desirable, a fallback of a **Fax Message** is used.

These two forms of message are called “Documents”. Let’s have a look at a first cut design…

First Cut Design

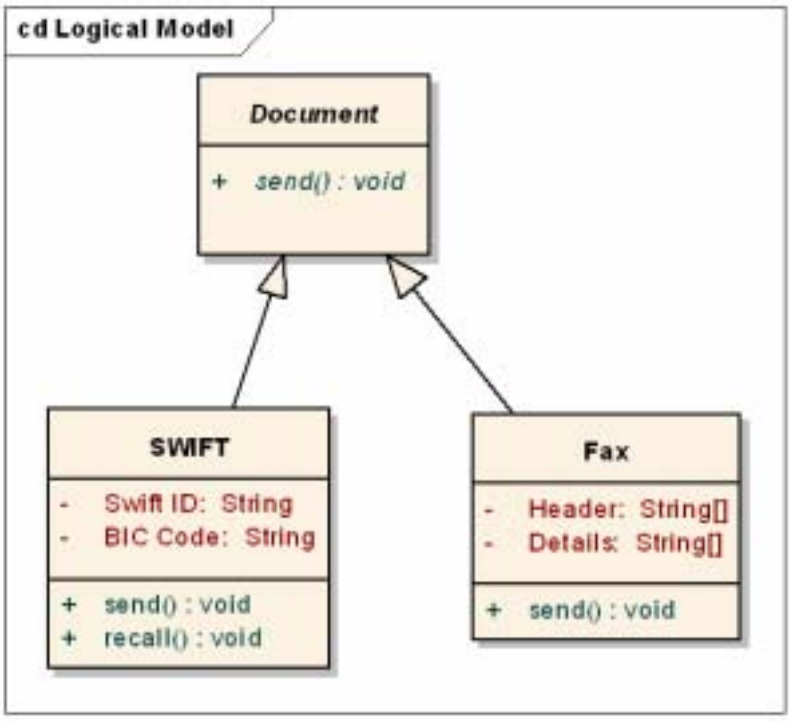


Figure 109 - First cut class hierarchy

From the design, we can see that Faxes and SWIFTs have very different attributes. Also, a SWIFT can be recalled[24], whilst a Fax can not.

[23] Society for Worldwide Interbank Financial Telecommunications - See www.swift.com

[24] For anyone from banking, this might not be true – we made this up for the exercise! Tell us if this “fact” is false…

But – they can both be “sent”. The implementations of both methods are very different, but the signatures of them are the same. This is why an abstract class looks to be appropriate here. But apart from the nicety of the design, have we really achieved much?

Don’t forget the LSP principle – we’ll illustrate with an example…

Example Use Case

A possible Use Case for this system is that we have to send a large collection of Documents “on batch”. Throughout the day, Documents are added to the queue and are sent once the day’s trading is complete.

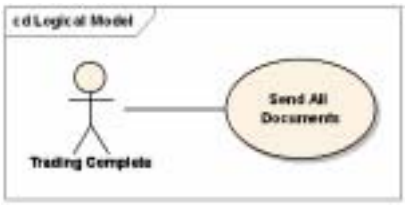


Figure 110 - Example Use Case; “Send All Documents”

But, we have a headache - different types of document have to be sent. Let’s see how LSP/Polymorphism helps…

Solving Using Polymorphism

Our aim is to write a client class that deals with *general documents only* – with care we can avoid being specific about which type is required…

In the UML this aim would look like this…

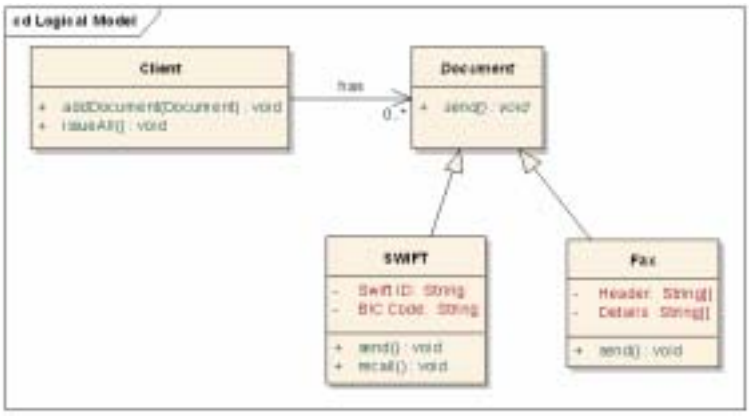


Figure 111 - The client class “holds” documents

But isn't this nonsense? Document is an abstract class – therefore we cannot create any objects from the class!

So how can we pass in document objects into the "addDocument" method of the Client class?

The LSP is the answer – recall that this means that *any subclass* can be passed in to the method instead. And furthermore, it means that wherever we refer to a Document in the code, any subclass can be switched in *at runtime*. Let's see what this would look like in the code.l..

Example Pseudocode – Client

```
-- Pseudocode (Ariadne 07)
Client
Begin
   -- the client holds an array of documents
   -- which type? Who cares?

   documents : Array of Document

   method Add_Document (new_doc : Document)
   begin
     -- this method adds a new document

     documents(documents.length) := new_doc;
   end method

   method Issue_All
   begin
      -- send out all the documents
      -- who cares about their types?!

       for count : all in documents
         documents(count).send;
      end loop
   end method
End class
```

Figure 112 - the client can be written with no knowledge about the specific type of document

Notice that all the methods here accept a parameter of type "Document" - but we can NEVER create a Document!

However, the LSP means that an object from any subclass can be passed in instead. Presumably, once the system is running, actual documents (Faxes, SWIFT or whatever) will be created outside of this class and then passed in to the add_document method.

The Big Payoff

For illustration, imagine that 20 years later, we find our creaky old system is still running but has been modified and must now support many more Document types…

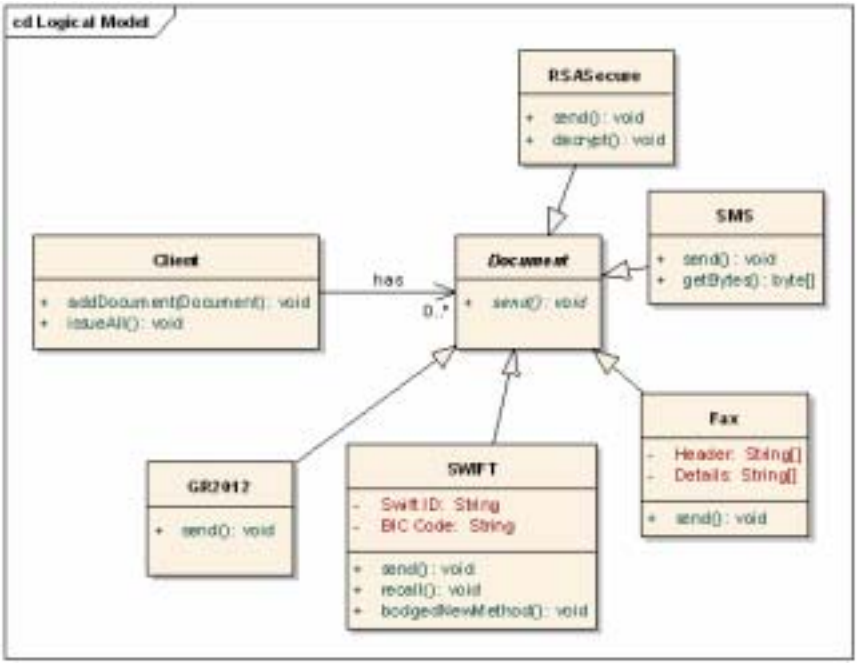


Figure 113 - The Design 20 years later

As an exercise, look back at the client code in tice that all the methods here acce – how much of the code is affected and how much rework is there to do?

Interfaces

Some languages (Java, VB and C# for example) recognise the concept of an **interface.**

This is defined as being a *purely abstract class*. The “document” class on the previous example could have been designed as an interface instead…

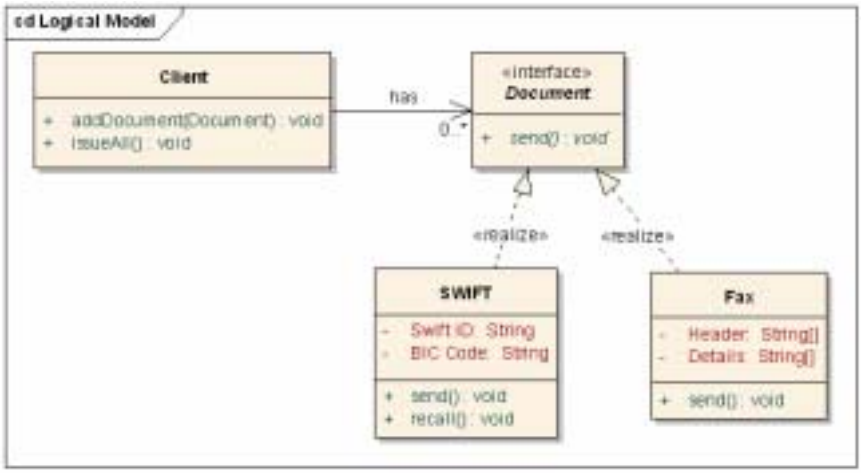


Figure 114 - Making the Document class into an interface instead

The UML notation for an interface is the <<interface>> stereotype. Also, note that the "generalises" relationship is replaced by "realize" to emphasize that no concrete behaviour is being inherited.

Interfaces are generally considered to be more flexible in modern programming languages that normal abstract classes – but the theory is the same and they really are just purely abstract classes.[25]

Composition /Aggregation

We'll leave inheritance behind now and look at Composition and Aggregation, a concept that is frequently confused with Inheritance. The UML has a diamond symbol to denote "Composition" or "Aggregation" (we'll be more specific about the difference between the two in a moment).

[25] Most languages have now banned Multiple Inheritance, in recognition of the huge problems that can be caused by them. However, those languages do allow multiple realization of interfaces, as abstract methods cannot clash. This is one purpose of interfaces, but generally interfaces are because building pure abstract classes is a very strong design principle and modern languages feel the need to support them as "first class concepts".

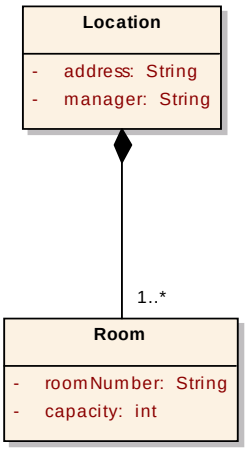


Figure 115 – A "Has A" relationship

This kind of relationship is often called "Has-A". So – Each Location Has one or more rooms.

Let's look at the difference between Composition and Aggregation:

Composition

Composition tends to mean that the parent object is **built** from the child objects. Without the parent object, the child objects have no meaning, and this means in programming terms that deleting the parent means deleting the child as well:

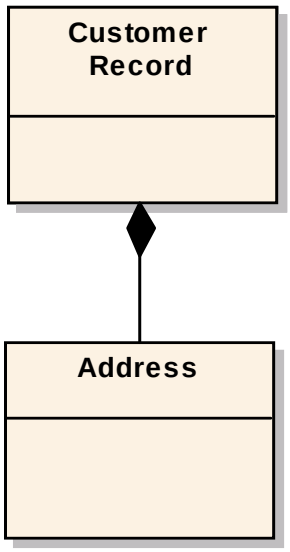


Figure 116 - A Composition Relationship

In this example, an address is owned entirely by a customer record, and if the customer is deleted, then the address must be deleted as well.

Note that the diamond is solid to denote composition.

Aggregation

In aggregation, the parent still “has” an instance of the child class, but the lifetime of the two objects are not bound together. In addition, the child object might also be shared across many parents.

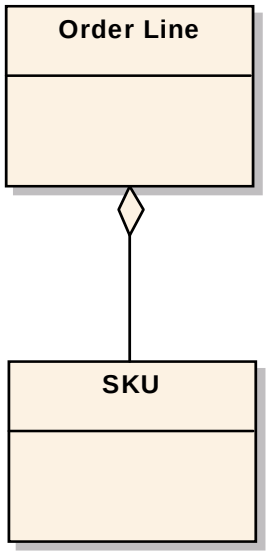


Figure 117 - Aggregation (note the "open" diamond)

Here, the order line contains an SKU, but deletion of the line does not mean deletion
of the SKU. In addition, an SKU could appear on many different lines.

Programming Language Note

This is only for those interested in moving towards a programming language.

- Composition will become a **by value** variable in the code
- Aggregation will become a **by reference** variable in the code

Often, the difference between composition and aggregation is ignored, especially if you are programming in a language where “by value” is not allowed (like Java).

Composition vs Inheritance

There is a large school of thought that suggests that Composition should be favoured over Inheritance[26].

Inheritance is not inferior to composition, but it turns out that many relations which may appear to be inheritance at first turn out to be composition - in general terms, composition is much easier to manage.

Take the following example – here we wish to “reuse” the functionality of a engine and a brake light, and make them part of a car.

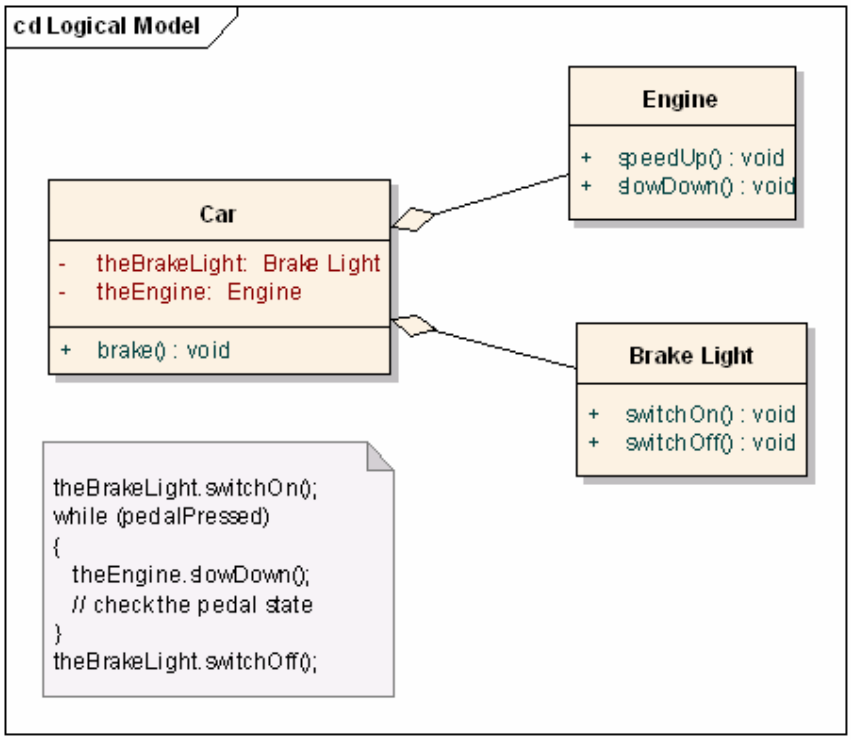


Figure 118 - A Car Has-An Engine and Has-A Brake light – no inheritance needed for this form of “reuse”

[26] See reference [5]; this classic work argues this very strongly and even sets it as the overriding goal of the book.

Users of the Car class only need to know about the “brake” method. The low level methods dealing with the engine and the brakes are “hidden” from the user of the car (just as in real life, thankfully). This is Encapsulation again, just in a slightly higher form.

Note that an attempt to model this design using inheritance would expose ALL of the methods through the Car class, even though the brake light and engine would be “reused”…

Summary

In this section we looked at Inheritance and Polymorphism

- Classes can be arranged into an “inheritance hierarchy”
- A sub-class must inherit all of the parent class’s public behaviour
- Polymorphism is an incredibly powerful tool to achieve code reuse

Ensure you are inheriting in a valid way, and make sure you consider the composition/aggregation alternative first

Chapter 13

Design Heuristics

In this session we will present some guidance on how to produce “good” OO designs.

We’ll look at Loose Coupling and High Cohesion. Also, we’ll present some simple rules (heuristics) to keep your designs on course.

“Talk to the Expert”

The first heuristic is easy to state:

Ensure that behaviour is allocated to the correct classes.

What do we mean by “the correct class”? Well, the correct class is often the class with the information necessary to do “the job”.

It is easy to state “talk to the expert”, because we have used the collaboration diagram, which is basically forcing you to apply this heuristic.

However easy the heuristic is to state, it is **the** fundamental OO design heuristic. A sloppy and casual approach to allocating responsibilities leads to poor quality designs.

“Talk to the Expert” should be at the forefront of your mind at all times when designing.

“Talk to the Expert” Example

Assume that in this example, the job at hand is to calculate the cost of a purchase order.

Here is an example (rotten) solution…

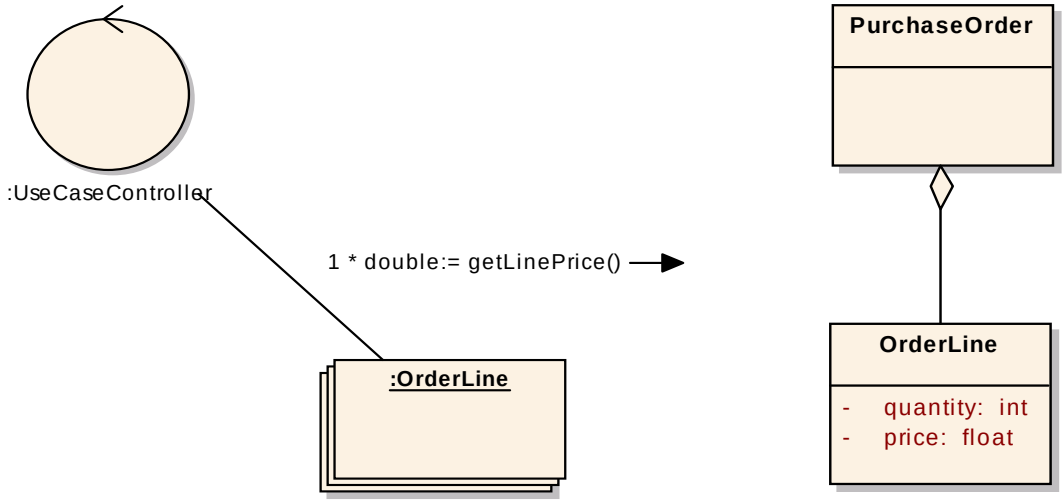


Figure 119 - A first cut solution. Not very impressive.

The solution is poor because how does the POTotalCalculator know which lines to send its message to? There could be thousands of order lines in the system, only 5 of which appear on our required PO!

The **expert** here is the Purchase Order – it knows about the contents of purchase orders, and so it should be asked to return the total of the purchase order, and it should make the calls to its lines…

"Talk to the Expert" Solution

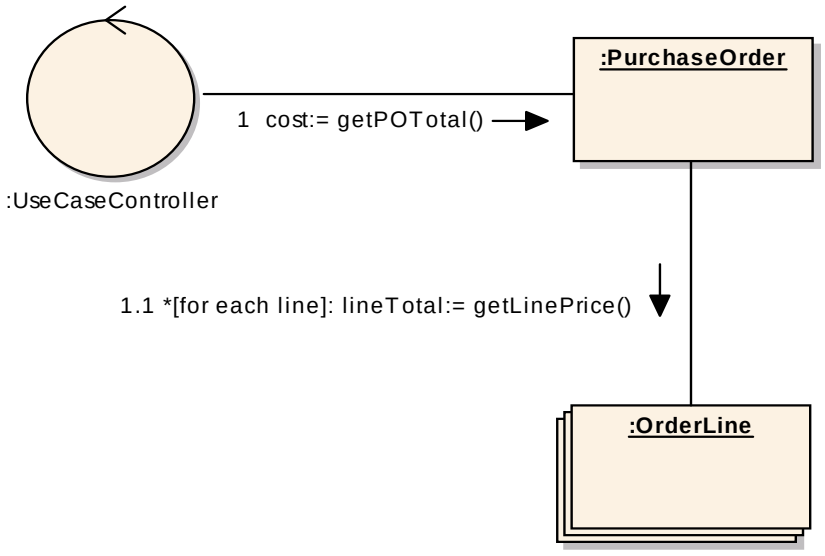


Figure 120 - Refactored solution, with "talk to the expert" in mind

Not only is this a better solution, it is far easier to code.

Heated debate: note that more messages are required here that in the previous solution. Is this going to perform more poorly?

In fact, you will often find that many good OO designs involve long chains of objects passing messages to each other; many OO methods are very “thin” methods that simply “pass the parcel” on receipt of a message.

From a performance point of view, however, this is unlikely to cause a serious problem. Assuming that the messages are not passing across a network or invoking database calls, the overhead involved will be negligible.

Tight Cohesion

Tight (or high) cohesion means that the responsibilities of a class (in this case) are strongly related. A class that does too much work (ie has lots of unrelated methods) is said to exhibit *Low Cohesion.*

Classes that are in-cohesive are harder to understand (and therefore harder to maintain). The design is also more brittle; a single change to an incohesvie class is more likely to impact other classes.

To give a real world analogy : a person in a company who does too many different jobs can be a liability!

Tight Cohesion Heuristics

Some simple rules of thumb to avoid loose cohesion:

- Keep the responsibilities of each class focussed
- A single class should not do too much work
- Strive for classes with the minimum number of methods
- Strive for classes which represent specific objects (aka “One Key Abstraction”)

As an example of the “One Key Abstraction” Heuristic, consider the following class. It is trying to model a lift control system:

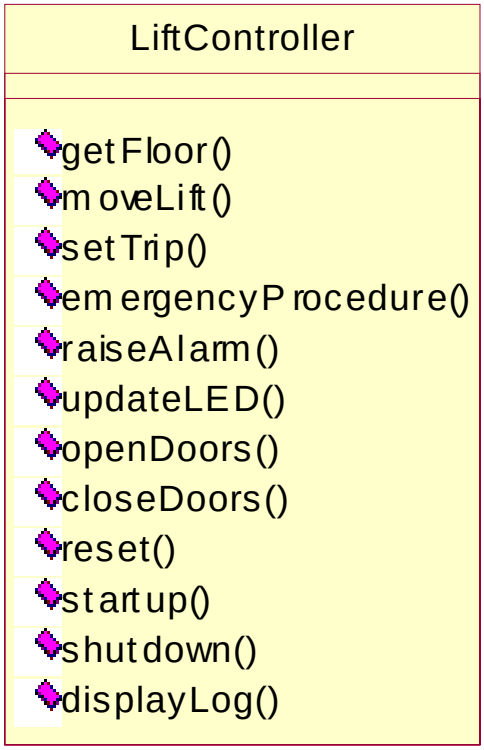


Figure 121 - a bloated (incohesive) class

Let's look at how to improve the quality of this design…

Refactoring the LiftController

The problem with this class is that it is representing more than one real world object. If we look carefully, we can see that there is a Lift, Alarm, Door and a Fault Log being modelled by this single class.

The class diagram might look something like this:

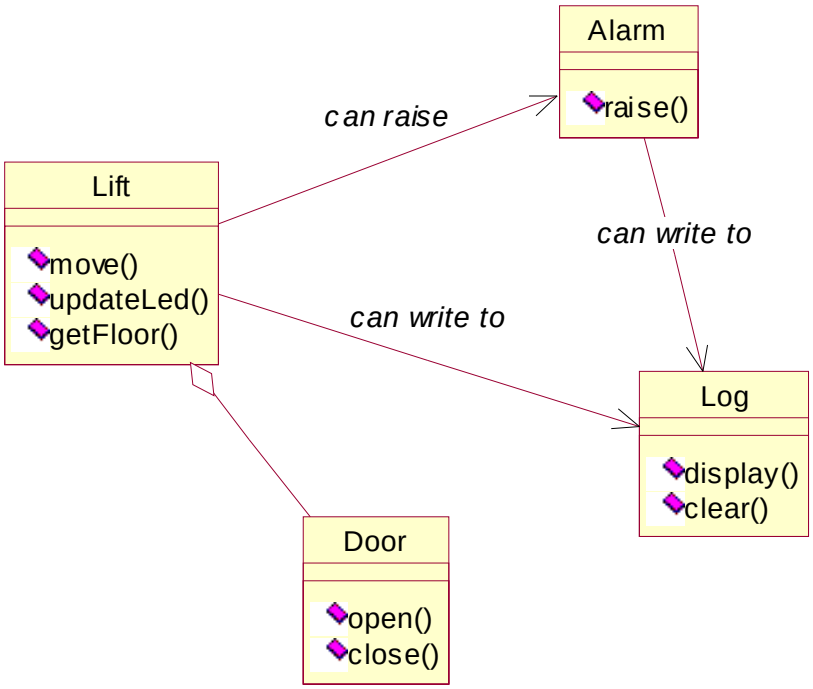


Figure 122 - The Refactored solution

The concept of High Cohesion always goes hand in hand with the concept of Loose (or Low) Coupling…

Loose Coupling

Coupling is a measure of the level of dependency that exists between your classes. High coupling leads to hard to maintain code, because a change to one class leads to changes in many other classes - a single change can in other words “ripple” throughout the system.

Always think of Coupling as a “Bad Thing” - but some coupling is absolutely required (a design with lots of independent classes doing their own thing is just as bad as a tightly coupled system).

The art is to *reduce* coupling to a *minimum.*

Once again, we have a set of heuristics to help us to avoid unnecessary coupling…

Spotting Coupling

Perhaps the simplest coupling heuristic is an intuitive one: “complex class or collaboration diagrams are a *bad smell*”.

The term *bad smell* is a semi-formal OO term. We use it when we are suspicious that something may be wrong. It may well turn out that we are wrong and all is well, but most designers develop a sense for when something is going wrong.

The following diagram (admittedly : it is made up) is giving off a bad smell. It just looks complex – why so many relationships – why relationships cutting across the diagram – why SEVEN associations coming from the “Tank Controller” class?

Certainly, the “Tank Controller” seems to be playing too much of an important role in this design, and you can bet that simple changes in there are likely to impact many of the other classes.

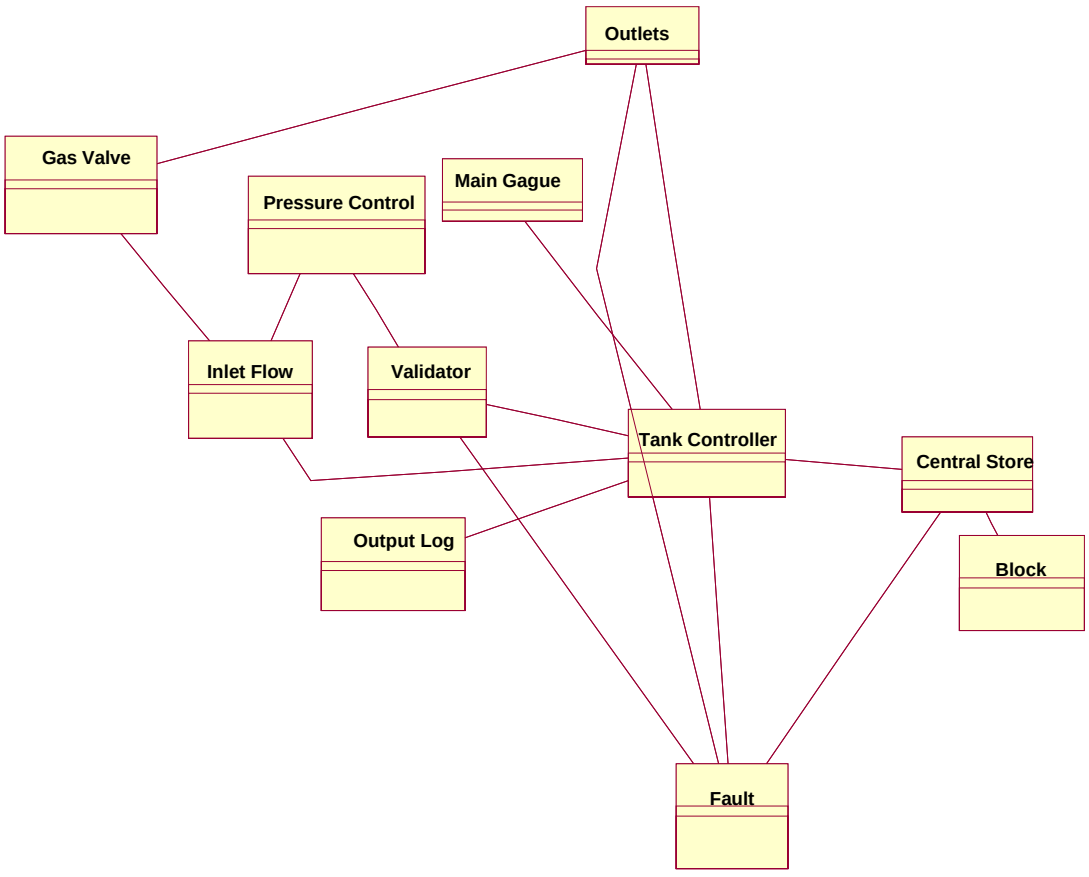


Figure 123 - A worrying class design

Heuristic : Don't Talk to Strangers

Another coupling heuristic is known informally as "Don't Talk to Strangers". You may see it formally called "The Law of Demeter"[27].

In "Don't Talk to Strangers", we are simply saying : make your objects "talk" to as few other objects as possible.

Although this is another intuitive heuristic, there is one mechanical rule you can apply here: use your Domain Model as a guide to the *minimum level of coupling* acceptable in your system.

When you decide (when building a collaboration diagram) to allow an object talk to another object, and your have not already associated them on the Domain model, you are **raising coupling.** This may be perfectly valid – perhaps you forgot the association originally, or its need wasn't obvious. But, stop and ask yourself the questions:

- Is this really the best way to do it?
- Is there another way that avoids the coupling?

If you ask yourself the question and you are satisfied that you aren't raising coupling, then go ahead and add the new association to your class diagram.

More Coupling Heuristics

- Avoid "Two way Traffic"

  Two way traffic is where you have two classes who are sending messages to each other (rather than just one class sending messages to the other, in one direction). There is nothing inherently wrong in having two way traffic (and if you need it, go right ahead), but coupling is effectively doubled when Class A depends on Class B and vice versa.

- Don't use public attributes[28]

[27] The Law of Demeter has a more formal definition than our looser heuristic. See ref [2] for the full statement

[28] For .NET users : a read/write attribute is still private. It is just that special get/set methods are provided and these methods are called implicitly when a value is read or set by a programmer.

There are *very few* situations where public attributes are necessary. A public attribute opens up the possibility of another class in the system getting direct and uncontrolled access to the implementation of your class. That means coupling.

- Only provide get/set methods when strictly necessary

  A contentious heuristic this one, but there is no need for it to be. We are not saying that get and set methods (ie methods that are designed to return or modify the value of an attribute) are invalid. You will need them, and you will need them often. But don't just blindly put them in (some CASE tool code generators do this!) – only put them in when they are needed – in other words, when your collaboration diagram highlights the need for them.

  If you need lots of gets and sets, have you violated "Talk to the Expert"?

- Minimise data flow around the system

  Once again, methods with long parameter lists are a bad smell – the best methods are the methods that can be executed by a class without being given further information. If you have applied "expert" diligently, you will find this is often the case. As with the other heuristics, don't follow this slavishly, but keep it in mind.

- Don't consider coupling in isolation - remember High Cohesion and Expert! You could remove coupling (well, theoretically) by making the whole system one big class!

Heuristics Summary

Hopefully you'll find this list of heuristics useful when designing:

- Talk to the Expert
- Keep the responsibilities of each class focussed
- A single class should not do too much work
- Strive for classes with the minimum number of methods
- Strive for classes which represent specific objects
- Complex class or collaboration diagrams are a bad "smell"

- Use your domain model as a guide to the minimum degree of acceptable coupling
- Avoid raising coupling beyond the minimum
- Avoid “Two Way Traffic”
- Don’t use public attributes
- Only provide get/set methods when strictly necessary
- Minimise data flow around the system
- Don’t consider coupling in isolation - remember High Cohesion and Expert

Chapter 14

The Sequence Diagram

The Sequence Diagram is the second of the two **Interaction Diagrams**. It is very, very similar to the Collaboration Diagram – in fact until recently, the UML spec defined them as being interchangeable.

Many tools can generate one from the other – although most tools do not provide the full suite of syntax as defined in the spec.

We'll stick to the "implemented" features, but we will briefly mention the other syntax as well.

Collaboration Diagram

Returning briefly to the collaboration diagram, consider the following diagram:

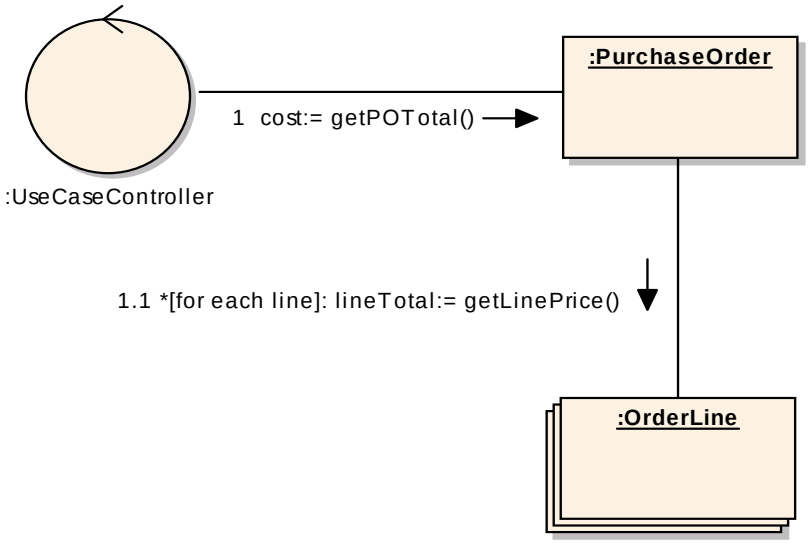


Figure 124 - A Simple Collaboration

As you're an expert in reading collaboration diagrams now, we won't detail what is going on here, but we will "convert" the diagram into a sequence diagram and look at the differences…

Equivalent Sequence Diagram

This diagram is **directly equivalent** to the previous collaboration diagram; it says no more and no less…

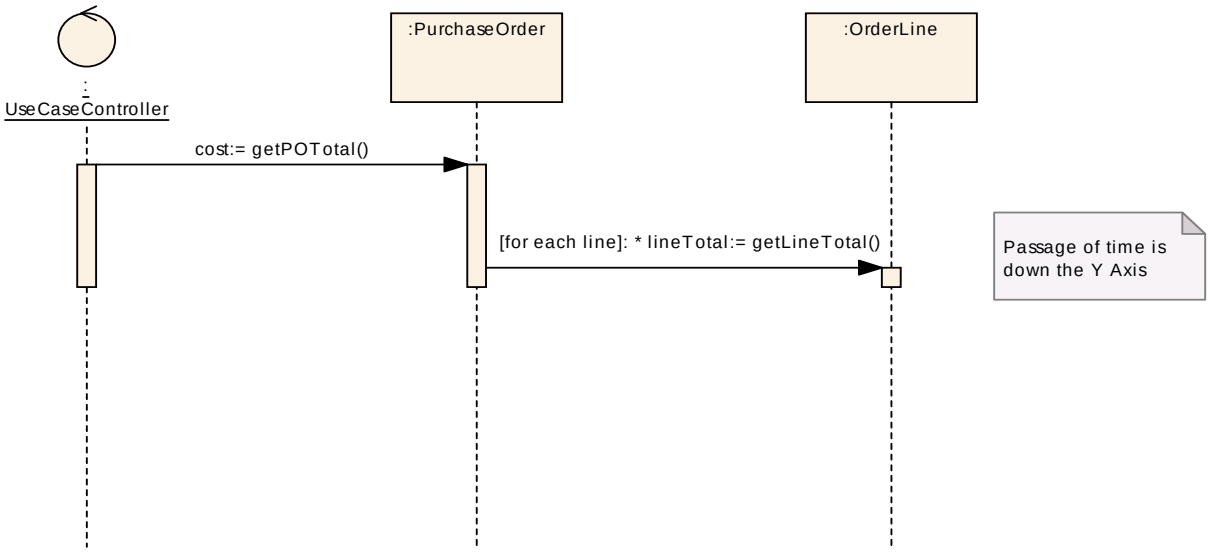


Figure 125 - The Equivalent Sequence Diagram

As you can see, the only real differences are that the objects are arranged in a straight line at the top of the diagram, and the numbering of messages is no longer required, as the Y-Axis in the downwards direction represents the passage of time.

The sending of messages is denoted by an arrow crossing from one object's "swimlane" to another. The ordering of the objects alone the top of the diagram is irrelevant.

You may also notice that there are "blocks" across the dotted lines on the swimlanes – these are called "Focus of Control".

“Focus of Control”

These blocks on the dotted lines (“Focus of Control”) are defined in the spec as: “The period during which an instance is performing a procedure either directly or through a subordinate procedure”.

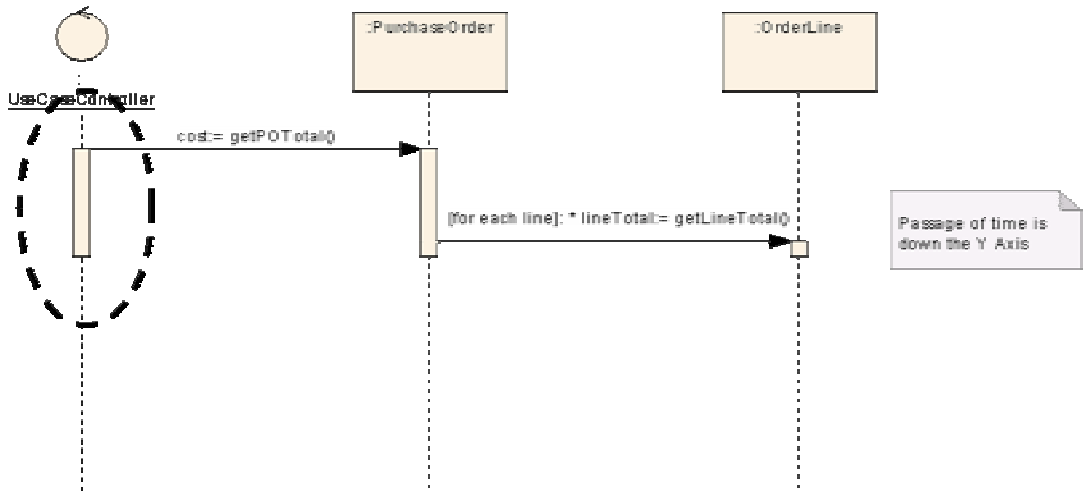


Figure 126 - The "focus of control"

Often it is not used or ignored; many people don’t understand what it means and some tools don’t implement it very well anyway

Iteration

Iteration can be denoted in a more graphical style than the collaboration diagrams could achieve. Its implementation on tools is very patchy; we certainly don’t rely on it at all.

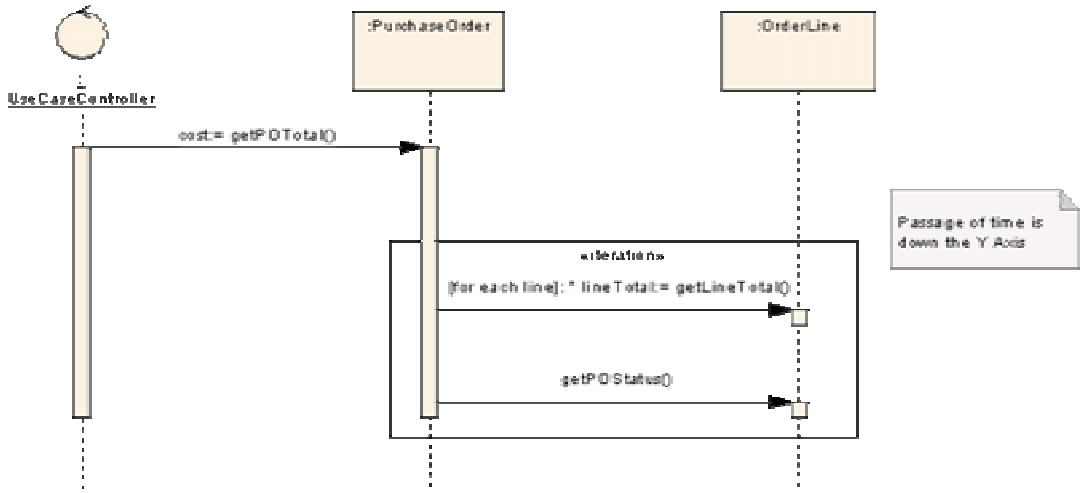


Figure 127 - Showing iteration with a box

Sequence Commentary

A benefit of the sequence diagram is that you can annotate it easily with the Use Case Description (as long as you don't mind copying and pasting into notes:

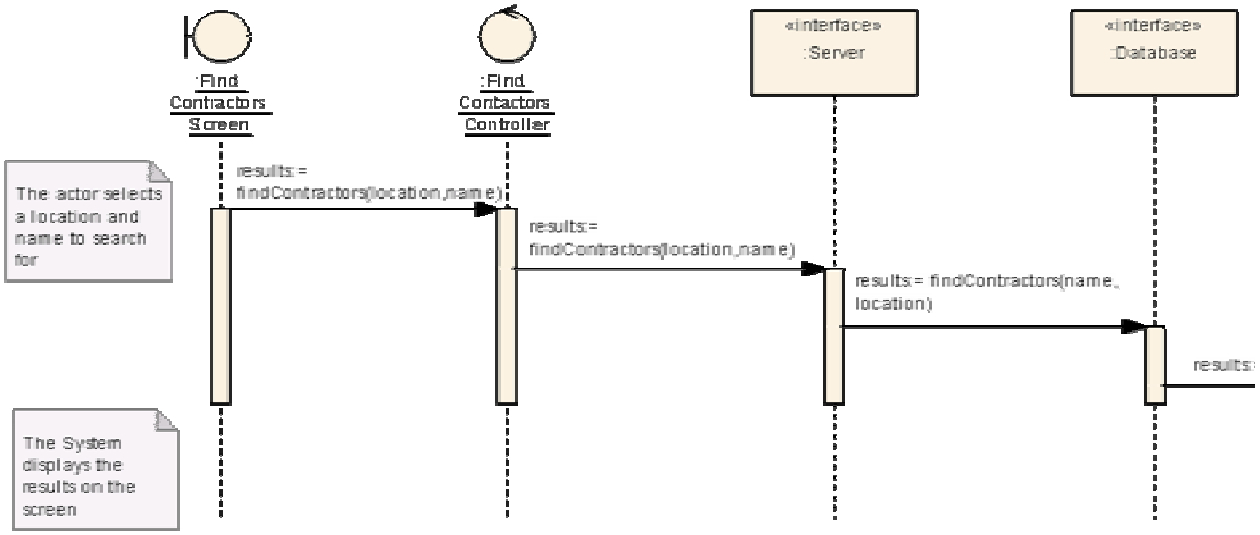


Figure 128 - Adding notes to provide a Use Case "commentary"

Deleting Objects

An extra item of notation (again this has a patchy implementation in tools) is for when an object is deleted…

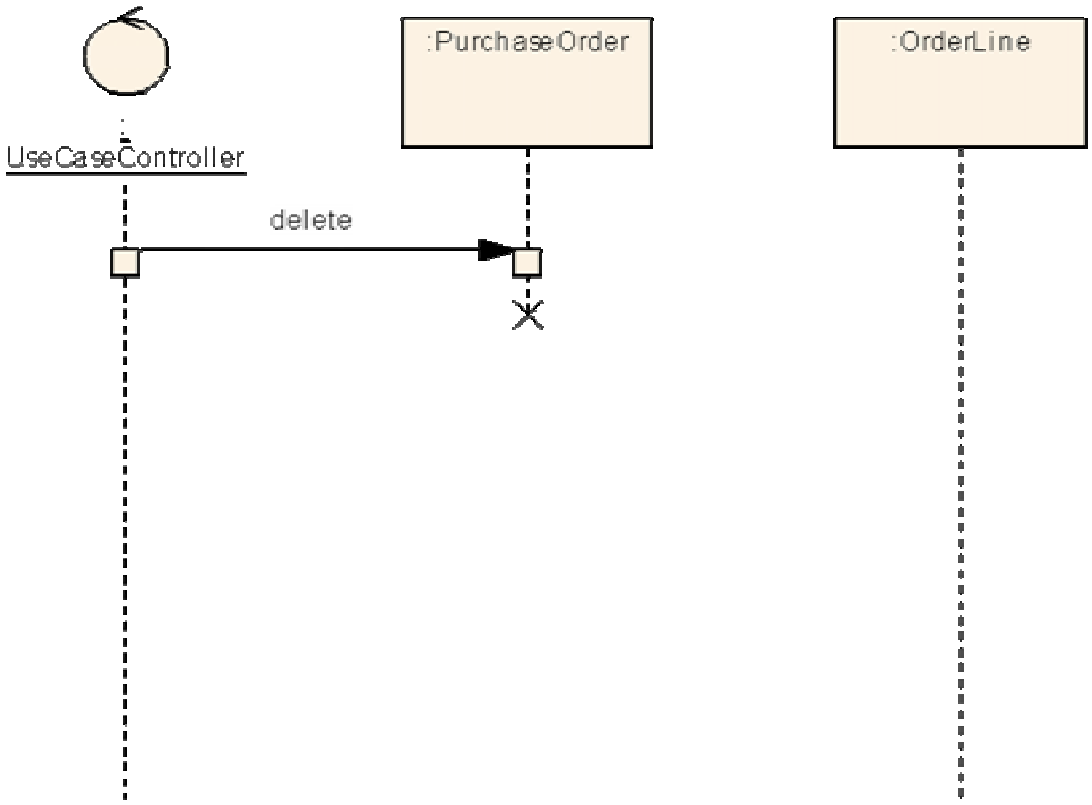


Figure 129 - An object is deleted

Notice that the swimlane for the object ends abruptly, and an “X” marks the point of deletion.

Which are Best?

In our very humble opinion, we think that Sequence Diagrams usually look more complex than Collaboration Diagrams – largely because the Sequence Diagram spreads from left to right – and looks horrid with more than 4 or 5

objects. It soon becomes very difficult to lay out[29]. The collaboration diagram is much more “free” in terms of layout.

One perfect use for the Sequence Diagram is where you are working with two or three objects, but you have a lot of message passing between them. This wouldn’t look so good on a collaboration diagram (you’d have a lot of messages all stacked up), but the sequence diagram would look quite neat.

So, our advice:

- Object Oriented model, with more than four objects collaborating => **Collaboration Diagram**
- Two or three objects with a lot of “chattiness”, say a diagram of how components interact => **Sequence Diagram**

Collaboration diagrams are much better for spotting erroneous dependencies (and therefore coupling) between the objects – it is easy to see if the diagram has turned into spaghetti, whereas the Sequence Diagram (due to its more rigorous structure) always looks the same regardless of the quality of your design.

However, our advice is very humble indeed, because we find that most developers prefer the Sequence Diagram. We think they’re missing out…

Summary

- A Sequence Diagram expresses roughly the same information as the Collaboration Diagram
- It has strengths and weaknesses over the Collaboration Diagram
- It is due to be renovated in UML 2.0 (to make it more “engineering strength”)[30]

[29] Our experience of struggling to fit the sequence diagrams on to the A4 width of this book proves the point beautifully

[30] Note also that the Collaboration is due to be renamed the “Communication Diagram” (even the UML falls foul of the rebranding curse sweeping the globe), and we are not aware of the potential impact of any changes at present

Chapter 15

Design Patterns

In this session we will introduce the concept of “Design Patterns”. Patterns are a very interesting and vibrant topic in Object Oriented design - it is, however, a very large topic and we can only scratch the surface in one session.

We’ll look at just a few of the design patterns that you will need to study if you want to go further with your design patterns

The Origin of Design Patterns

The architect (of buildings, not software) Christopher Alexander founded the concept of patterns in his classic book “The Timeless Way of Building”

A pattern defines a commonly occurring problem (conflict) in a particular context; he describes a generic resolution of the conflict.

Different Chairs

May be part of Sequence of Sitting Spaces (142), Sitting Circle (185), Built-In Seats (202).

Conflict

People are different sizes; they sit in different ways. And yet there is a tendency in modern times to make all chairs alike.

Resolution

Never furnish any place with chairs that are identically the same. Choose a variety of different chairs, some big, some small, some softer than others, some with rockers, some very old, some with arms, some wicker, some wood, some cloth.

May contain Pools of Light (252).

Pools of Light

May be part of Alcoves (179), Workspace Enclosures (183), Entrance Room (130), Sitting Circle (185), Eating Atmosphere (182).

Conflict

Uniform illumination- the sweetheart of the lighting engineers- serves no useful purpose whatsoever. in fact, it destroys the social nature of space, and makes people feel disoriented and unbounded.

Resolution

Place the lights low, and apart, to form individual pools of light which encompass chairs and tables like bubbles to reinforce the social character of the spaces which they form. Remember that you can't have pools of light without the darker places in between.

May contain Warm Colours (250).

Figure 130 - Two examples of the style of Alexander's Patterns

Design Patterns / GoF

In the early nineties, a group of OO Gurus began developing a range of flexible solutions to commonly occurring design problems. Their work was heavily influenced by the work of an Architect (of the building variety) called Christopher Alexender.

The classic “Design Patterns” book by Gamma, Helm, Johnson and Vlissides (1995) catalogued 23 of them. The four authors have become known as the “Gang of Four”; you will often see reference to this, or “GoF”.

This book is considered as being the definitive object oriented work; every OO team should have one. However...

Design Patterns Book

The book is not an easy read. For one thing, it uses OMT (it was written in 1995 and for some reason is still at the first edition), and C++ or Smalltalk. These apparently minor inconveniences do make the material harder to digest, and the style of the book is quite academic and formal.

We find the best approach when starting with design patterns is to:

- learn a few common patterns

- learn new ones as often as you can
- know what most of them refer to, even if you don't know the detail

Helpfully, the front of the book lists all the patterns with a one sentence-ish summary of each - learn all the summaries if you can. Other books exist that may be easier to understand/digest (see reference [3] for a much more approachable and no less insightful read).

Learning the design patterns is not an exercise in identifying copy-and-paste techniques for solving design problems; the patterns distil insight.

Intent and Problem in Context

Design Patterns are not telling you how to Design Software. They are not best practices, rules of thumb or design guidance. Many people assume that all the patterns must be applied at all times, and that implementing a particular pattern makes their software "good".

To fully understand a pattern, you must understand that all Design Patterns have an *intent* (describing the motivation for the pattern), and a *problem* (in context) - this describes in what situations the pattern is appropriate.

The 23 GoF Design Patterns

The design patterns have defined and recognized names. If you are talking to an object oriented designer, and you ask her to add a Decorator to the design, they will probably know what you're talking about – all 23 of these terms are now standard IT lingo.

Here's the list of 23:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy
- Chain of Responsibility

- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

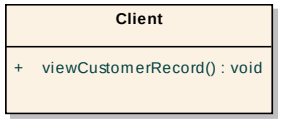
Adapter (1)

Imagine the following scenario…

- You are developing the client side of an application that will eventually connect to a database
- The database will hold customer records; your client application needs to manipulate the records
- The database is going to be developed by a third party developer, and the “interface” to their database is not available yet

Adapter (2)

You develop your client application, and make assumptions about the methods that might be available in the database component when it arrives…




```
procedure viewCustomerRecords
begin

   customer
      = theDatabase.getCustomerRecord(1)
   displayRecord(customer)

end procedure
```

Figure 131 - Our client side application accessible the “to be delivered” database

The code in the figure above is just some made up pseudocode – the important thing to note is that we have made a “stab in the dark” and decided

that the database will probably support a method called "getCustomerRecord".

Adapter (3)

Now assume that the third party database is delivered and… disaster… the interface to it is completely different!

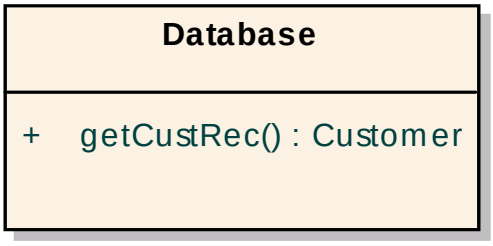


Figure 132 - The third party database

Their method is called "getCustRec" (they are clearly a lazy bunch of developers who like saving keystrokes). Our client code cannot work with this class - should we rework the client class?

Adapter (4)

A design pattern helps here – the intent of the Adapter is to (extract from GoF):

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

This is a simple pattern, where we simply introduce an "intermediate" class to bridge the gap between the two incompatible classes…

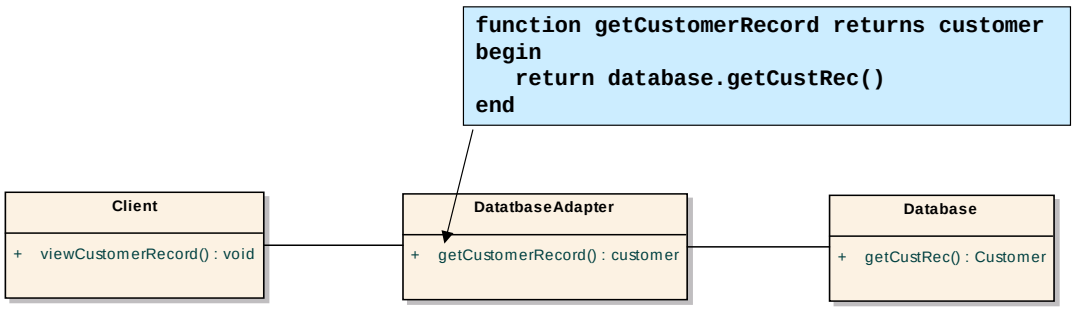


Figure 133 - Adding an Adapter

The adapter class simply is compatible with our own “client” class, and it simply delegates the method calls to the Database class. The adapter class will have almost no logic inside it – it will only have short methods that delegate to the incompatible class.

Simple, but elegant! Too simple perhaps?

Design Patterns Complexity

There is nothing new, earth shattering or complex about the Adapter pattern – but this is generally true of all design patterns. Some of them are a little complex, but they are all merely concise statements of good practice that you will often recognize.

However, with design patterns, we have:

- The ability to catalogue a wide variety of design issues in a consistent form
- A common language: when faced with the problem above, we could have said to the designer “I think we can solve the database problem with an Adapter”; and they would know what we mean (as long as they’ve studied design patterns of course!)

Another Design Scenario

We are going to introduce another design pattern, and along the way learn some more UML notation. To get there, consider the following scenario: imagine that, without proper consideration for architecture, your Design Class Diagram began to look like this after a few iterations (we have omitted methods and attributes for “clarity”)…

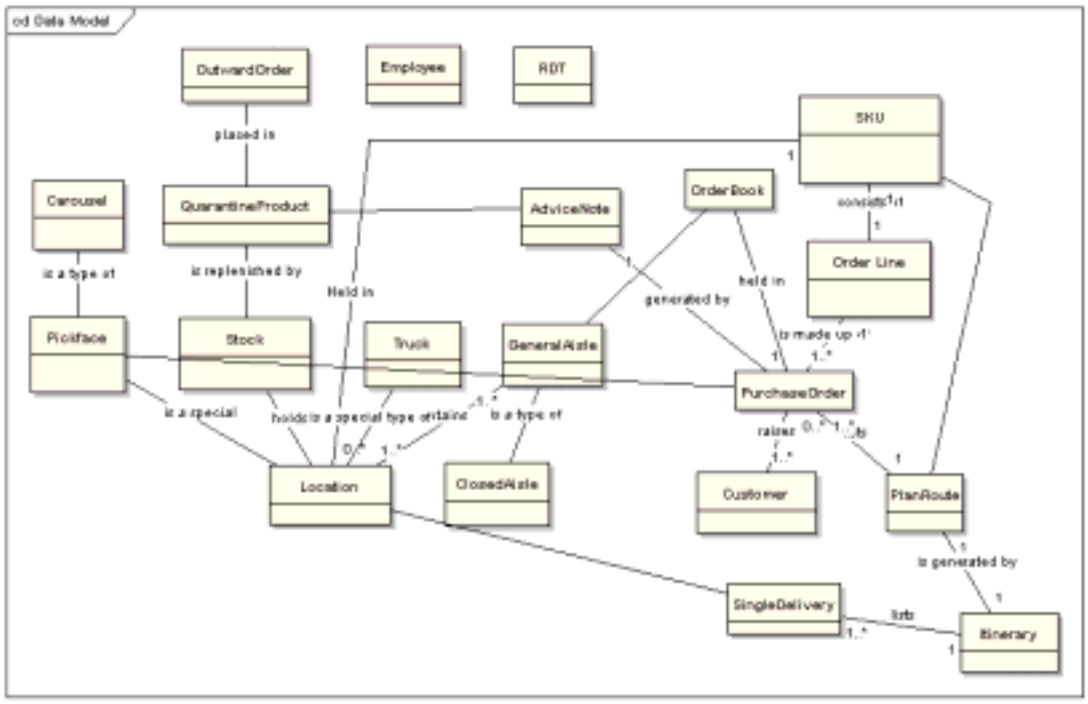


Figure 134 - Bad Smells abound here...

Partitioning the Design

Perhaps we ought to consider breaking down the complexity of the model by breaking it up into separate chunks. It looks like many of the classes are tightly related – for example the three classes at the bottom right are dealing with the deliveries of orders…

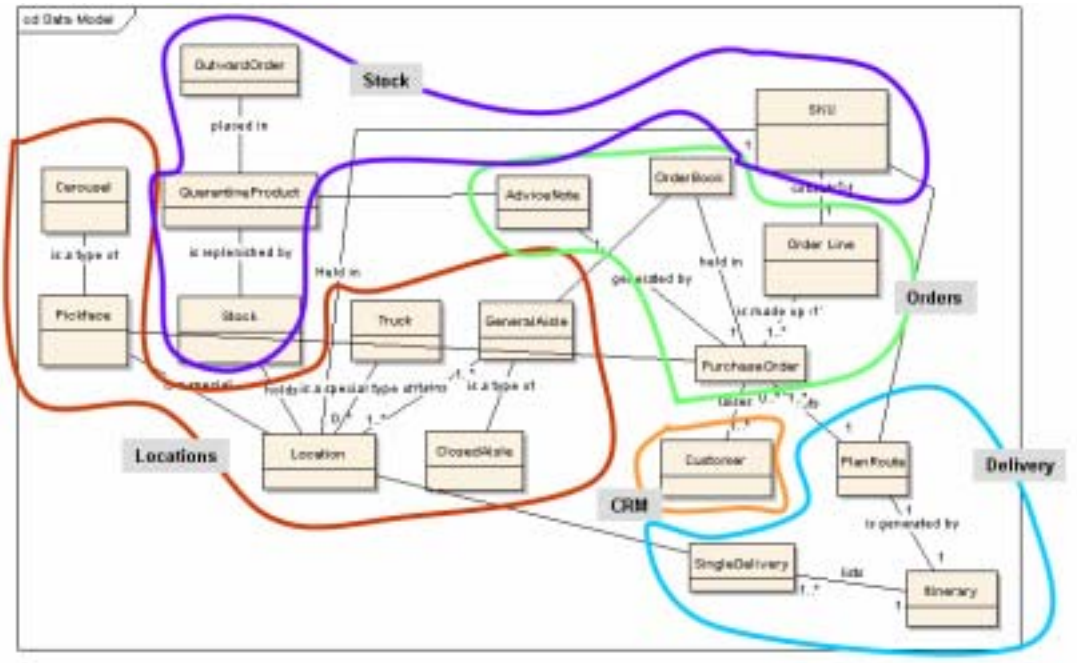


Figure 135 - An attempt at partioning the design

Here we have used Powerpoint to highlight closely related classes – but there is a formal UML tool to enable us to do this properly…

Packaging

The UML Contains a lightweight containment mechanism called a “Package” – think of it as being like a Folder in Explorer…

We could express the portioning from the previous figure using this notation as follows…

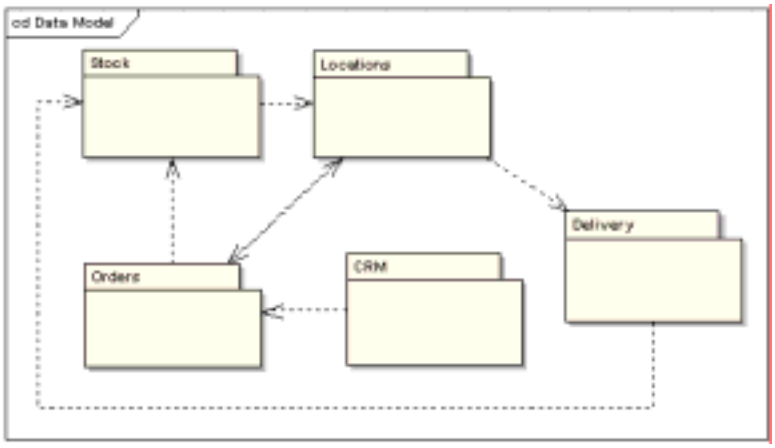


Figure 136 - The Package Diagram

These package symbols are simply symbols appearing on a class diagram. However, we do this so commonly that a term has been coined for a diagram that, like this one, contains packages only. We call this a **package diagram**.

The lines between the packages are called “dependencies” and they are used to denote where a package is dependent (in some way – we can’t be specific at this level) on another package.

Each folder can contain many classes - (most tools will allow a diagram to be attached to the package). In the tool we are using to draw the pictures here, if we double click on the “Delivery” package, a new class diagram opens up…

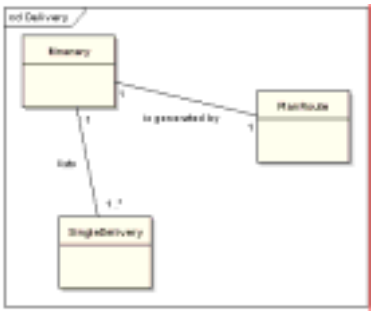


Figure 137 - The delivery package in detail

Now let’s head towards our design pattern and introduce a problem…

Problem

You are working on a class in the orders package, and you need to work with the locations package. There could be **hundreds** of classes in there (and we don’t want to think about those details - another team worked on the package).

How could we improve the design to avoid this complexity?

Solution – A Facade

One of the Gang of Four Design Patterns – the Façade - has the following intent:

“*Provide a unified interface to a set of interfaces in a subsystem. **Façade** defines a higher level interface that makes the subsystem easier to use*”

The application of the Façade is usually pretty simple; in our case we can add a class to each package and treat this class as an interface to the package…

Façade Implementation

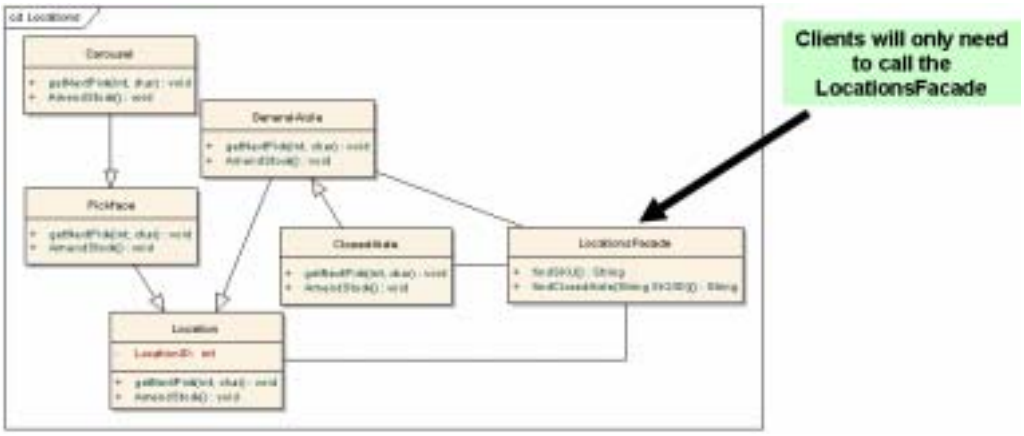


Figure 138 - We can "hide" the complexity of the package behind a simple class with simple, high level methods

Façade "Field Notes"

Design Patterns are not slavish "cut and paste" jobs – you will see Facades in different guises with slightly different uses. As long as the intent of the pattern (unifying interfaces) is upheld, you are implementing a façade.

For example, you could:

- Insist that communication between packages is ONLY through the Facades – this turns your packages into Components
- Make the facades optional – this means the Façade is used purely to simplify a complex package for clients
- Make the facades pure "pass through" classes that introduce no business logic
- Introduce business logic into the façade –a security or caching layer added before the calls to the lower level classes are made

A Design Situation

Some of the Design Patterns are a little harder to understand than Façade and Adapter. We will now provide a context and motivation for a harder Design Pattern – we won't reveal the pattern we are using until we have hit the problem. We borrow heavily here from Shalloway and Trott's superb Design Patterns book (see reference [3]).

For this system, we are tasked with building a drawing application that must be capable of drawing a square on the screen.

Furthermore, we have been supplied with a third party graphics library fronted by a class called "RapideDraw".

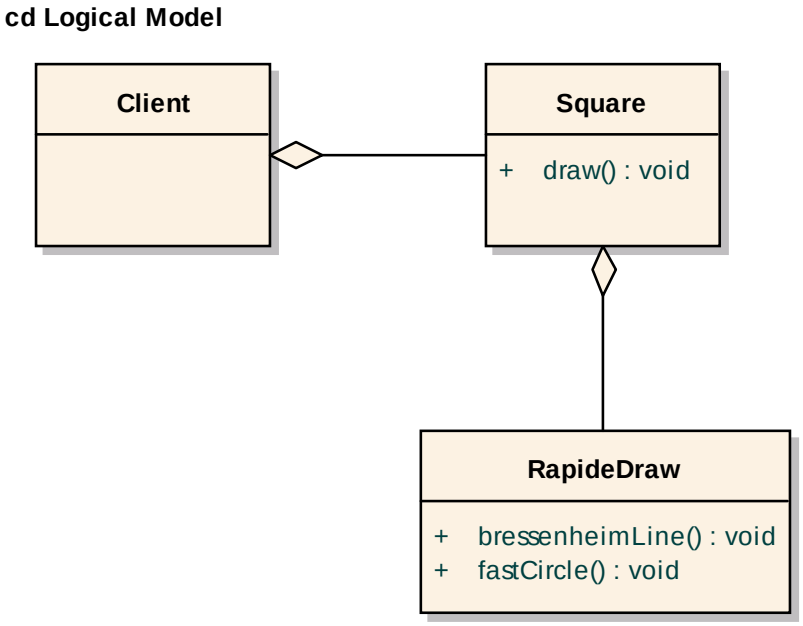


Figure 139 - Our first attempt at a Class Diagram for the Drawing Application

The RapideDraw package supports the drawing of straight lines implemented using an algorithm called "Bressheim's Algorithm", a method of drawing straight lines using pixels developed in 1965 (drawing a convincing straight line on a grid of pixels is not as easy as you might think!)

A New Requirement

We are now given the requirement that the application must also support different low level libraries – we'll add in a second library called AccuDraw. As

a first pass, Inheritance seems like a good choice to provide these two special types of Square (they pass is-a-kind-of and 100%):

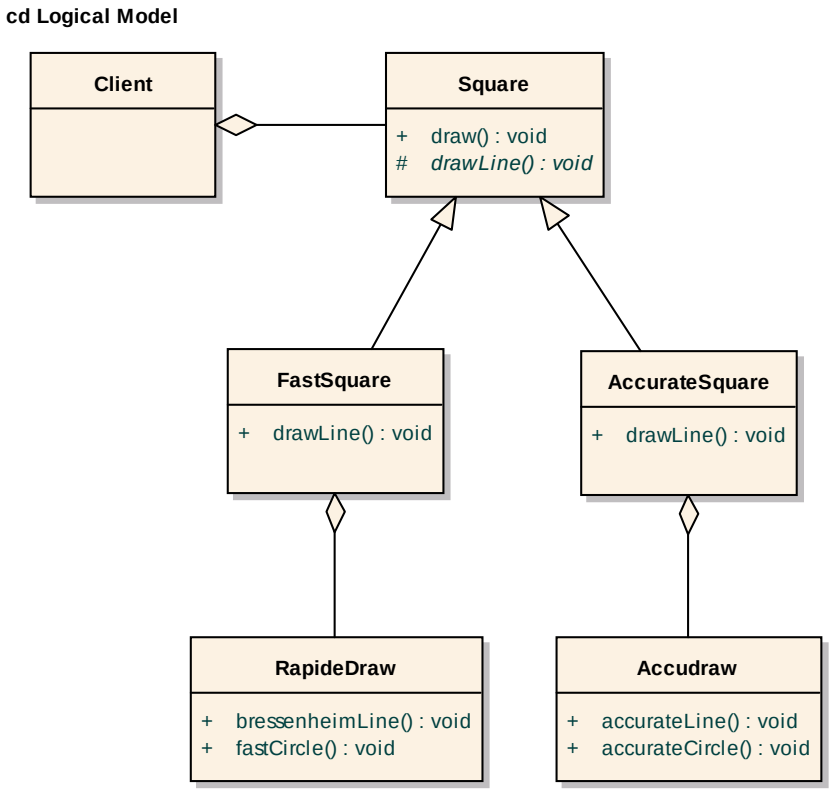


Figure 140 - Adding the new Accurate Drawing Package

On construction, the client creates a either a FastSquare or AccurateSquare. The client doesn't care about the difference, since they both support the draw() operation. By polymorphism, the correct concrete code will be called.

As is always the case, we now get a new requirement – naturally we want to implement circles in our application (which sounds reasonable, given that both RapideDraw and AccuDraw support the requirement). But we need to add a new class called Circle – where should it go?

After some pain, we recognize that we also have an abstract class at play here, called "Shape"…

Adding Circle Support

This is how the new class design has panned out…

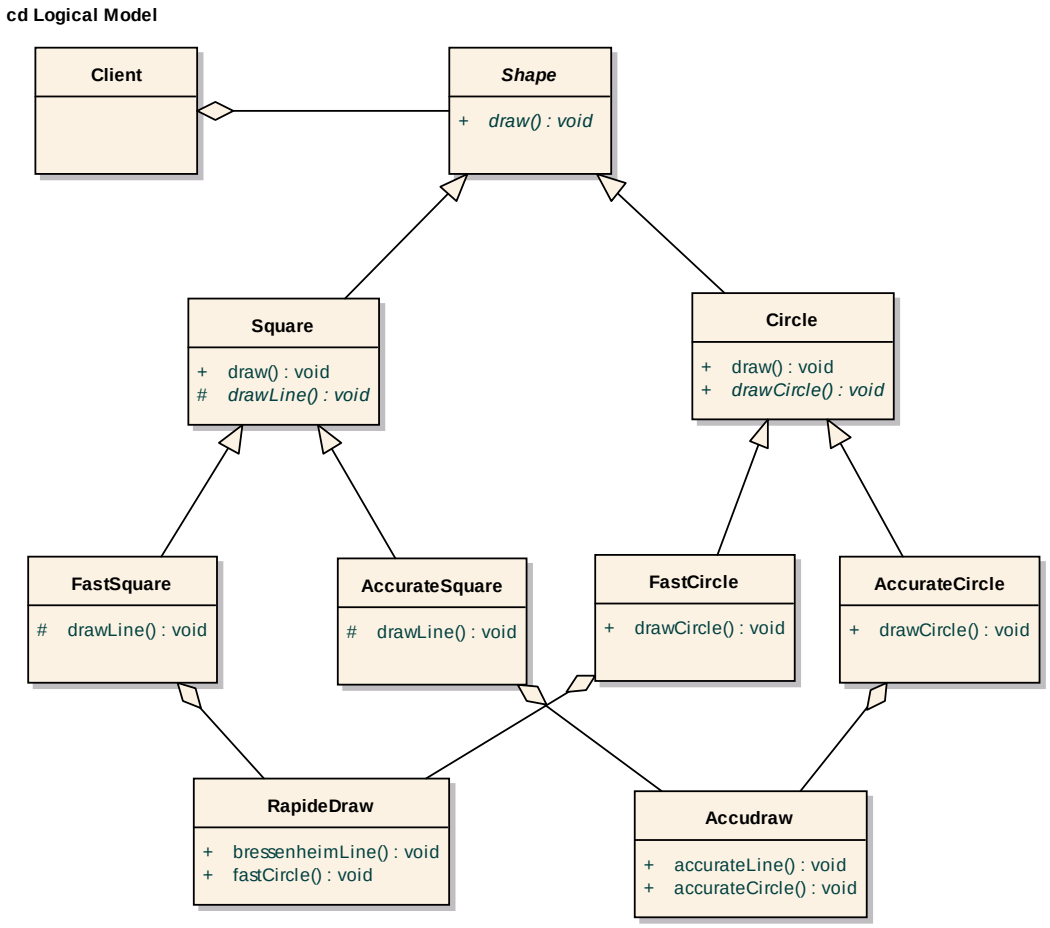


Figure 141 - Using simple inheritance to add support for Circles

Things suddenly look worrying – what about if we need to support another Drawing Library? The answer is chaos…

Chaos!

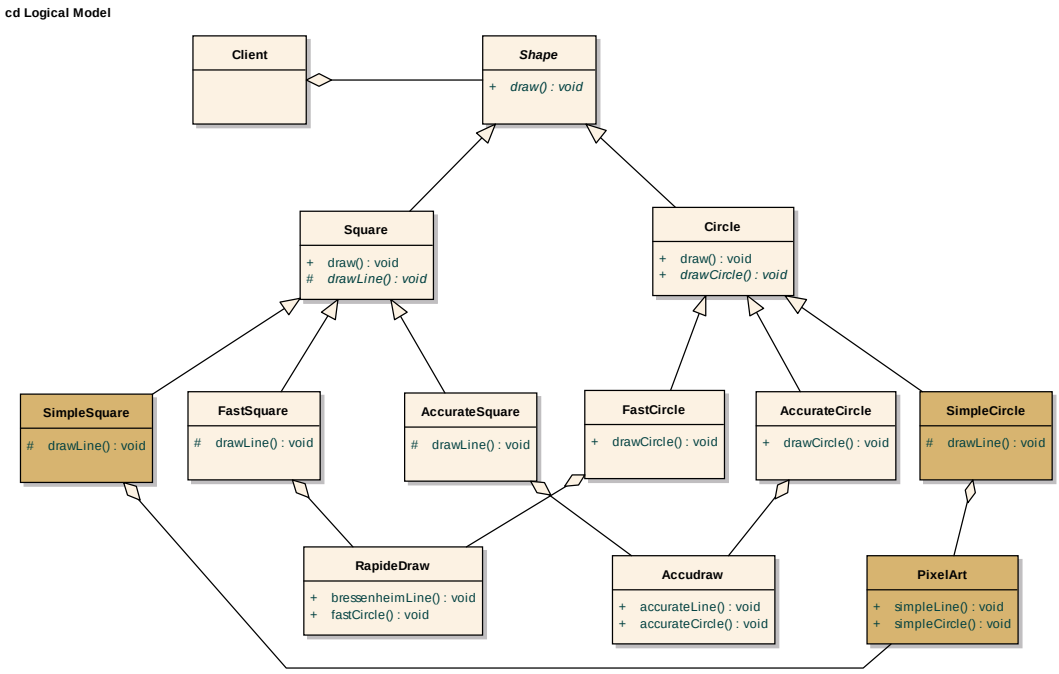


Figure 142 - Adding the new drawing package

This has had an unfortunate effect on our design – we have what is referred to as a **Combinatorial Explosion** in that for every new class we add we have to support various different combinations of the existing classes.

Surely this can be done differently, and better? (By the way, if you don't believe that this design is now unwieldy, try adding support for a triangle!)

What's Gone Wrong?

This is a good example of where a design starts simple enough, but suddenly collapses into chaos. The problem is that we have two different concepts in our design that we have mixed together - furthermore, the two concepts are very likely to *change independently.*

The concepts are the **Shapes** and the **Drawing Tools.**

As we saw earlier in the session on MVC, the solution is probably to isolate the two things that are vulnerable to change, and to separate them…

Optional Exercise

Have an attempt at refactoring the solution.

We are assuming that you are not familiar with the design pattern that would help, so we are asking you to engineer a Design Pattern from first principles!

Therefore – don't worry if you come up with nothing more than a sheet of paper with some frustrated scribbles on – this is only to get you thinking about the problem!

The Bridge Design Pattern

The intent of the Bridge Design Pattern is:

"*Decouple an Abstraction from its Implementation so that the two can vary independently*"

This is what we need - here the abstraction is the **Shape**, and the implementation is the **Drawing Tool.**

Let's see if the Bridge helps us…

Bridge (1)

First of all, consider the two independent aspects of this design in isolation…

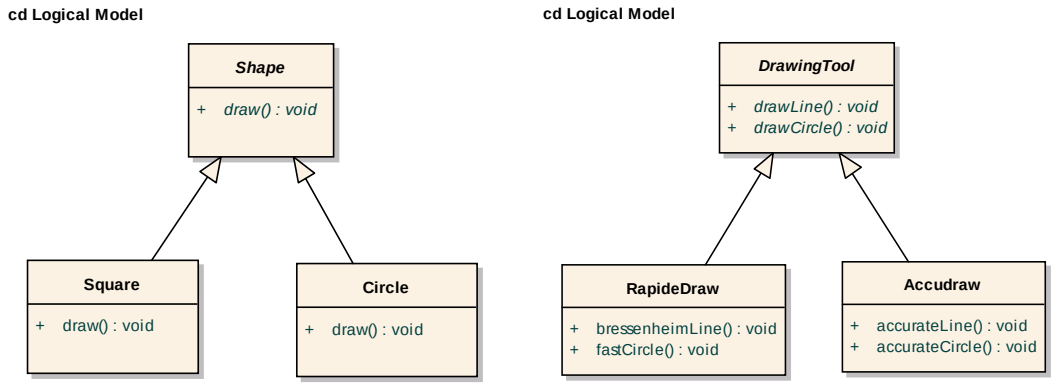


Figure 143 - The Shape and DrawingTool hierarchies

Bridge (2)

Simply stated, we need to tie these two hierarchies together.

We also need a Shape to be able to call on some drawing services - so we give Shape access to a DrawingTool via the **Bridge…**

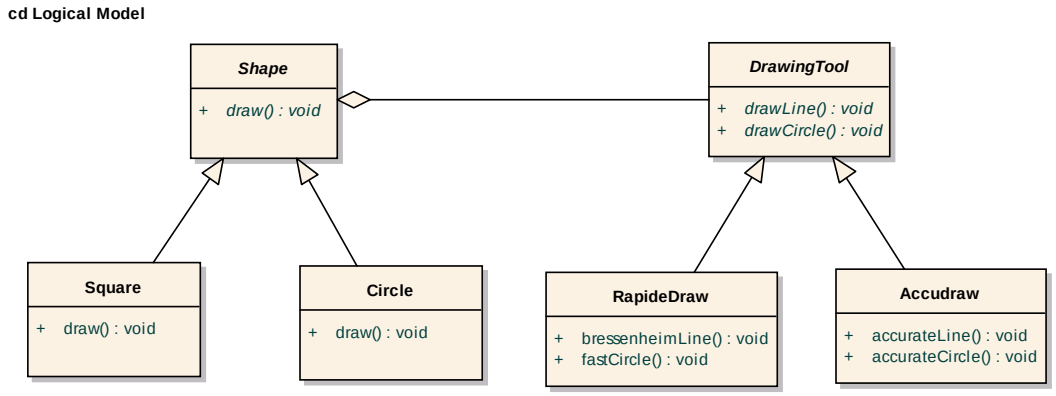


Figure 144 - Adding the Bridge

Bridge (3)

We can now add the implementations of the primitive behaviour drawLine() and drawCircle() in the Shape class.

The implementations of these methods simply delegate across to the DrawingTool[31]

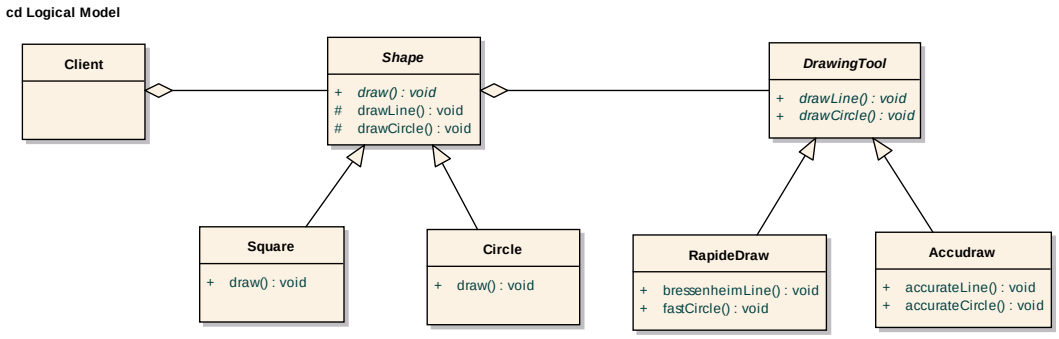


Figure 145 - Full implementations of the basic line and circle drawing abstract methods

Final Improvement

We have abstract methods called drawLine() and drawCircle() which are currently not implemented in the Accudraw and RapideDraw classes.

[31] (This violates the 100% rule by the way – ho hum)!

Can we solve this problem?

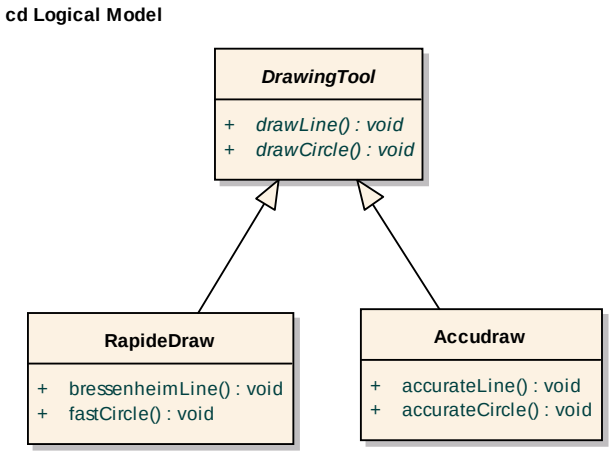


Figure 146 - We haven't properly implemented the DrawingTool abstract classes

The Adapter

We have a class here that needs to use another class but cannot because of incompatible interfaces. This should ring a bell - a job for the **Adapter** – a pattern within a pattern(!)

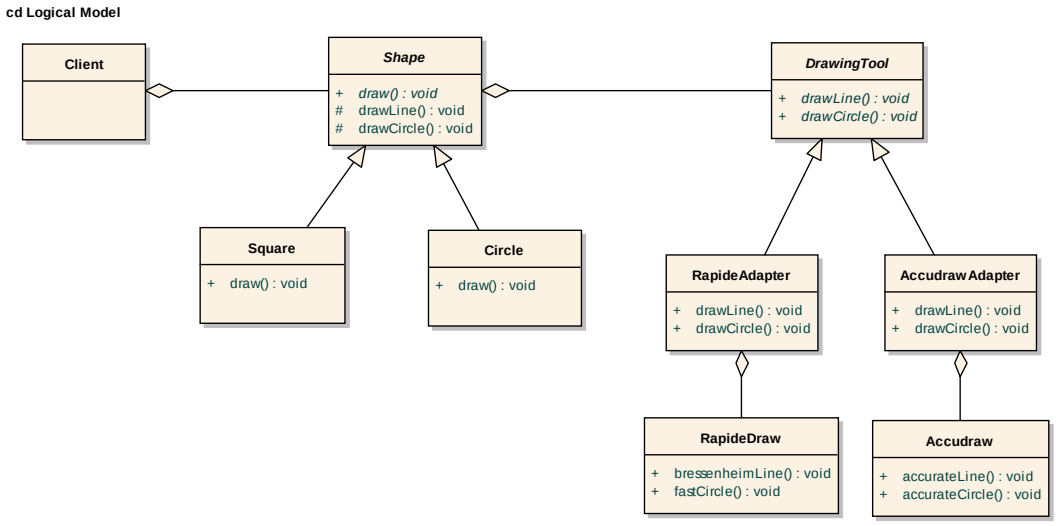


Figure 147 - The full solution complete with a pair of Adapters

Extending the Design

Now let's see how easy it is to extend the design – let's add the new PixelArt drawing package…

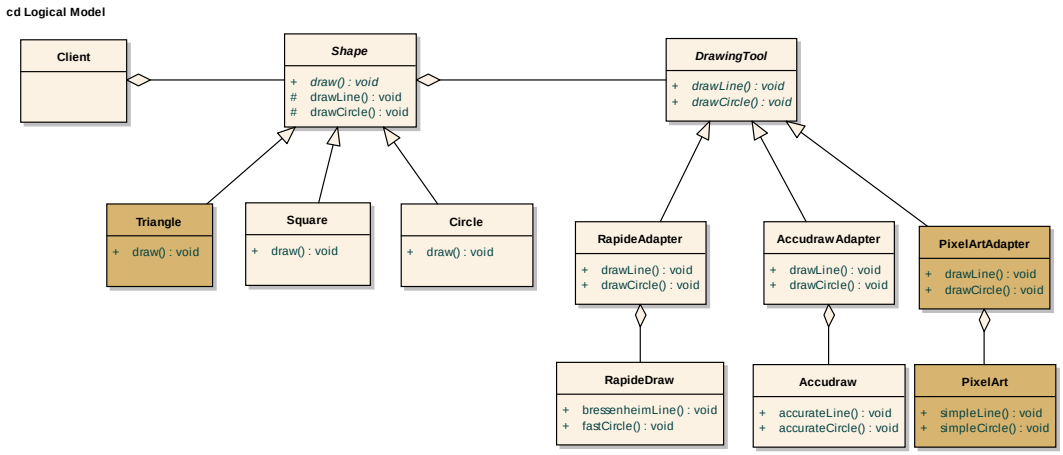


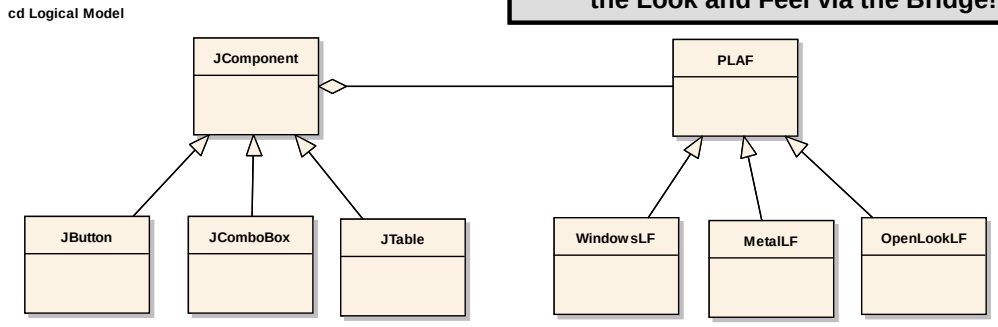
Figure 148 - Adding the PixelArt package

We've also added a Triangle class to the system – with minimal change!

The Bridge in Java

The is just for Java users…the Bridge Design Pattern is used as part of the fundamental structure for the Swing (JFC) Library. Can you recognise it?

Java effortlessly provides the ability to switch between different Look and Feels (PLAF's) to support different platforms. The GUI components themselves are unaffected because they are isolated from the Look and Feel via the Bridge!



More Java Notes

Design Patterns are used extensively in the design of the Java API:

- The Iterator is used to iterate across collections
- The Command is used (in the guise of an Action class) to replicate behaviour across Swing GUI's
- An instance of a JDBC Driver class is a Singleton
- The IO Library uses Decorators (hence the need for the cumbersome constructors!)
- Swing uses MVC (not a GoF pattern)
- The creation of Borders in Swing requires the use of a BorderFactory (the Factory Pattern)

Patterns – Last Words

Now for some final words on patterns. We've looked at just four of the "Gang Of Four" patterns – but don't forget there are 19 more – although not all of them are as commonly used as these four.

GoF started off the "Patterns Movement", but the movement is growing. Look around and you will find all kinds of Patterns related books, based on:

- Real Time Patterns; where common problems in real time systems are solved
- Analysis Patterns; where common Analysis problems (such as how to deal with multiple currencies) are solved
- J2EE Patterns
- Architectural Patterns
- Enterprise Patterns (see reference [8])

Summary

Design Patterns are generic, elegant solutions to commonly occurring problems. There are 23 "Gang of Four" classic patterns - the book is not an easy read but is reckoned to be the classic, definitive work on OO.

Chapter 16

UML 2.0

UML 2.0 has now been fully adopted by the OMG, and tool support is now beginning to arrive for the new version.

We have deliberately stuck to UML1.5 on this course as most projects are still working with UML1.5. However, there are few changes that impact us directly – there are three new diagrams added and some of the names of the diagrams has been changed.

Many of the loose areas of UML have also been tightened up, and internal changes to the specification have been made to support future technologies.

Apart from the minor renaming of some diagrams, the new version of UML has been designed to be backwards compatible.

The diagrams are now divided into two categories : Behavioural and Structural…

Behavioural Diagrams

There are seven diagrams under the Behavioural category:

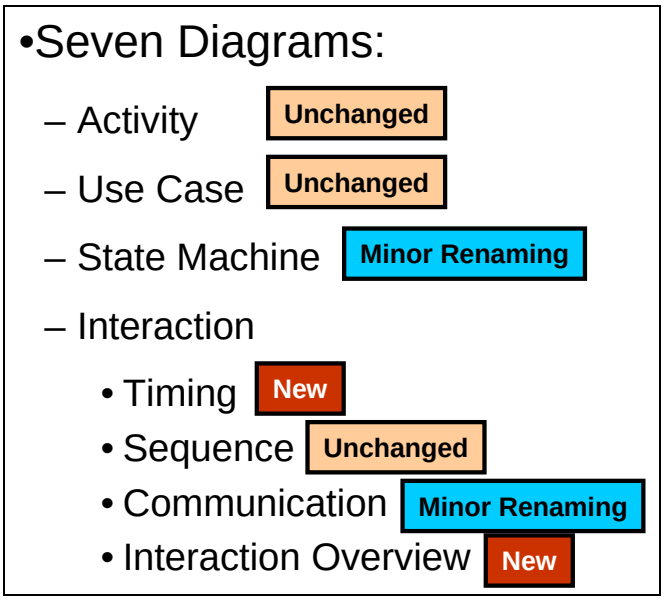


Figure 149 - Behavioural Diagrams

The Statechart has been renamed “State Machine”, and the Collaboration Diagram has been renamed “Communication Diagram”.

The Timing and Interaction Overview diagrams are new, and we will describe them shortly.

By the way, many of the existing diagrams now support new features – but you will easily pick these up via your CASE tool if you find them useful[32]

Structural Diagrams

There are six diagrams in the Structural category. Only the Composite Structure Diagram is new…

[32] For example, the System Boundary has now been officially recognised in the Use Case Diagram – people have been using these for years without realising the UML1.x didn’t mention them!

- Six Diagrams:
  - Package Diagram **Unchanged**
  - Object Diagram **Unchanged**
  - Composite Structure Diagram **New**
  - Component Diagram **Unchanged**
  - Deployment Diagram **Unchanged**
  - Class Diagram **Unchanged**

Figure 150 - Structural Diagrams

We'll now take a very quick look at the new Behavioural Diagrams

Timing Diagram

The Timing Diagram shows how an objects behave (change state and interact) over time:

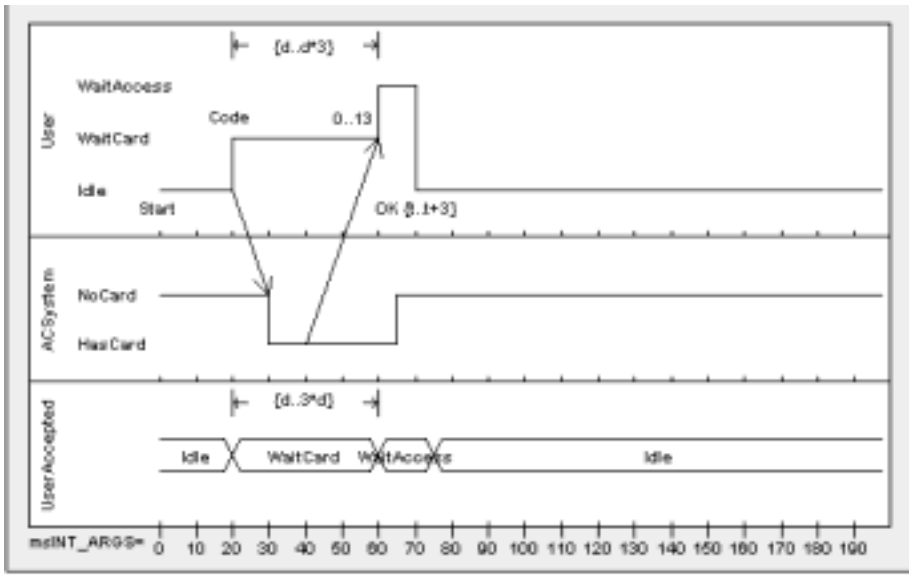


Figure 151 - Example Timing Diagram

Interaction Overview Diagram

Interaction Overview Diagrams are used to denote the flow between related sequence diagrams. The syntax is almost identical to the Activitiy Diagram, except instead of activities, boxes are used to denote “sub” behaviour diagrams, or links to existing behaviour diagrams - this diagram will be used to tie behavioural diagrams together:

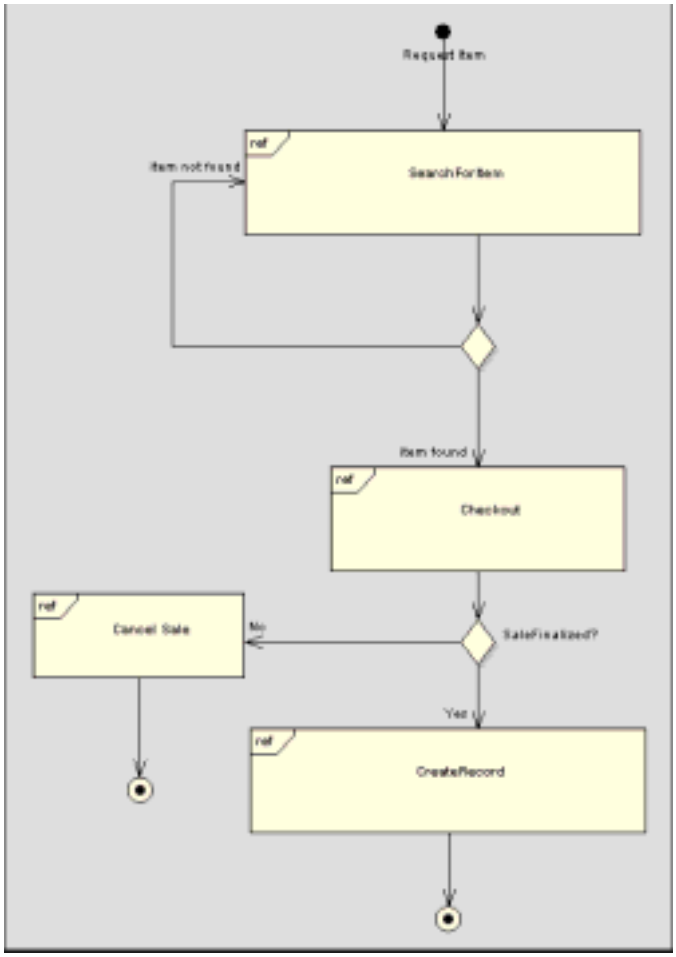


Figure 152 - Interaction Overview Diagram

Model Driven Architecture (MDA)

One of the major changes in UML2.0 are internal structural changes to drive forward the possibility of building systems using MDA. Very simplistically, MDA effectively aims to automate the process of generating code from the UML Model - making your UML Model “executable”.

The promises made by MDA aren't yet here; complete tool support will take time to arrive - but keep an eye on this exciting new technology.

Summary

- UML 2.0 is now here, and we must begin the process of getting used to the new models if we are going to work with modern CASE tools
- We can fairly easily continue using UML as most of the changes are backwards compatible
- MDA promises much for the future

Chapter 17

Transition to Code (not covered on course)

Note: this chapter won't be covered explicitly on the course – it is quite simple and covers several different languages. We have provided it here for reference.

In this section, we shall:

Look at some basic coding of UML Models, and quickly look at how UML maps to Java, C#, C++, VB, VB.Net and Ada. This is not a programming course, so unfortunately we cannot go into any great detail, but we do hope to show that as the UML has been designed with programming languages in mind, the transition code is relatively simple (but this does depend on your language, as we shall see).

We'll look in more detail at coding a Use Case, and present a full code example in an Appendix (we've shunted the full code to the back of the book to avoid clogging the body of the book with code).

We'll also briefly introduce the concept of *forward and reverse engineering*.

Mapping a Class

The most fundamental concept is how to map a class from UML to the programming language. We'll take our simple SKU class; we have removed many of the attributes and methods for clarity.

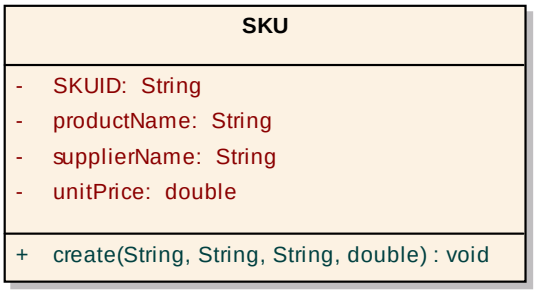


Figure 153 - The simplified SKU Class

We'll begin by looking at how the class is coded in Java…

Defining a Class (Java)

The mapping to Java is very clean and simple…

```
package pharmacare.domain.stock;

/**
 * Definition of a Stock Keeping Unit
 */
public class SKU
{
   private String SKUID;
   private String productName;
   private String supplierName;
   private double unitPrice;

   public SKU(String SKUID, String proudctName,
              String distributor, double unitPrice)
   {
      this.SKUID = SKUID;
      this.productName = productName;
      //etc
   }
}
```

Figure 154 - SKU Class in Java

Two main things to note:

- Java fully supports the concept of a package. The declaration at the top states that this class belongs in the Stock package (which is contained in a higher level package called domain, and the parent package is called pharmacare).

- The "create" method becomes a special method called a **constructor**. There's nothing special about this method other than that this method is automatically run when a new SKU is created (using the Java keyword **new**)

Finally, omitting the word “public” would have made the class package protected, as we needed when we were set up the component/façade architecture.

To create an instance of this SKU class, use the **new** keyword:

```
import pharmacare.domain.stock.*;

public class Test
{
   public static void main (String[] args)
   {
      SKU newSKU = new SKU(“SKU01”,”Palitoy”,”SKN Retail”,3.74);
      newSKU.methodCall(); //etc
   }
}
```

Figure 155 - Creating an instance of an Object

Defining a Class (C#)

C# (C-Sharp), the new language that forms part of the new Microsoft .NET framework, looks very similar to Java:

```
using System;

namespace pharmacare.domain.stock{

public class SKU {
        private String SKUID;
        private String productName;
        private String supplierName;
        private double unitPrice;

        public SKU(String SKUID , String productName,
                           String supplierName, double unitPrice)
        {
                this.SKUID = SKUID;
                this.productName = productName;
                this.supplierName = supplierName;
                this.unitPrice = unitPrice;
        }

    }//end SKU
}//end namespace stock
```

Figure 156 - SKU Class in C#

The namespace concept is exactly the same idea as a package, and once again, we could have omitted the “public” keyword to make the class package protected. Once again, the create method becomes a “constructor”; this method is automatically called when the object is created. As with Java, the name of the constructor is the same as the name of the class.

Creating an instance of a class in C# is identical to Java:

```
SKU newSKU = new SKU(“SKU01”,”Palitoy”,”SKN Retail”,3.74);
```

Figure 157 - Creating an Object in C#

Defining a Class (C++)

Nothing is simple in C++ (!); in C++ we require a header and an implementation file, as follows:

```
// SKU.h

class SKU
{

private:
        string SKUID;
        string productName;
        string supplierName;
        double unitPrice;

public:
        SKU(string, string, string, double);
};
```

Figure 158 - The C++ Header file, defining the structure of the class

The implementation of the class is placed in the .cpp file:

```
//  SKU.cpp

#include "SKU.h"

SKU:SKU(string SKUID, string productName,
        string supplierName, double unitPrice)
{
   this->SKUID = SKUID;
   this->productName = productName;
   this->supplierName = supplierName;
   this->unitPrice = unitPrice;
}
```

Figure 159 - Implementation of the class

Once again, the create method becomes a constructor; a method with the same name as the classs.

Sadly, C++ does not support packaging. A recent addition to C++ is the concept of a **namespace**, which is roughly analogous to packages in UML/Java, and namespaces in C#, except there is no way to enforce any "protection" on a package. So placing the SKU class in a namespace called "stock" only really prevents name clashes with identically named classes in other packages.

One way to create packages in C++ is to compile each package into its own DLL (Dynamic Link Library), or whatever type of library your platform supports. The interface to the DLL takes the place of the façade class.

Defining a Class in VB.NET

Visual Basic.NET (effectively VB 7) now takes the same kind of approach as Java, C# and C++. Classes, objects and constructors are fully supported (previous versions had weak implementations of OO that didn't quite get there).

Here's an example of a class module in VB.NET:

```
Namespace pharmacare.domain.stock

Public Class SKU

          Private SKUID As String
          Private productName As String
          Private supplierName As String
          Private unitPrice As double

            Public Sub new(ByVal sSKUID As String, ByVal sproductName As String,
                           ByVal ssupplierName As String,
                           ByVal dunitPrice As double)
               SKUID = sSKUID
               productName = sproductName
               supplierName = ssupplierName
               unitPrice = dunitPrice
          End Sub
End Class

End Namespace
```

Figure 160 - VB.NET Class

Although the syntax is slightly different to our previous examples, the way of thinking is exactly the same.

Defining a Class (Ada)

Ada has always featured Objects and Classes (or at least, a direct analogy to them). In Ada, a class maps onto an “Abstract Data Type” with a *private part.*

The rather annoying restriction in Ada is that all methods in a class have to be converted into standard functions or procedures with the first parameter being an instance of the type. By the way, this is done automatically in many other languages – it is the “this” or “self” param d(.22 TwTy.2((om)4.8(a)-6.3(t)i)-7.4(ngj[(Definp“t)8.3(h)-5.r

```
package Stock is

   type SKU is private;

   procedure initialise (this   : SKU;
                         SKUID  : String);
   private
      type SKU is record
         SKUID      : String(1..50);
         -- etc...
      end record;

end Stock;
```

Figure 161 - SKU Definition

Sadly, there is no concept of a constructor in Ada, so we have provided a method called initialise. The only problem with this is that we will have to call it manually ourselves.

Here is another module creating an instance of the SKU and calling its methods:

```
with stock;

procedure main is
   SKU_Record : Stock.SKU;
begin
   Stock.initialise(SKU_Record,"SKU01");
end main;
```

Figure 162 – Creating an SKU and calling a method

Ada supports packages, but once again it is impossible to impose “package protection” – so in this example, the SKU “class” is visible to the entire project (as long as it is **with**ed in).

Adding Reference Attributes

We’ll now look at how to relate classes. We won’t cover all of the languages as that would be tedious, so we’ll use Java as an example.

Consider the following example, where each SKU is stored at exactly one location:

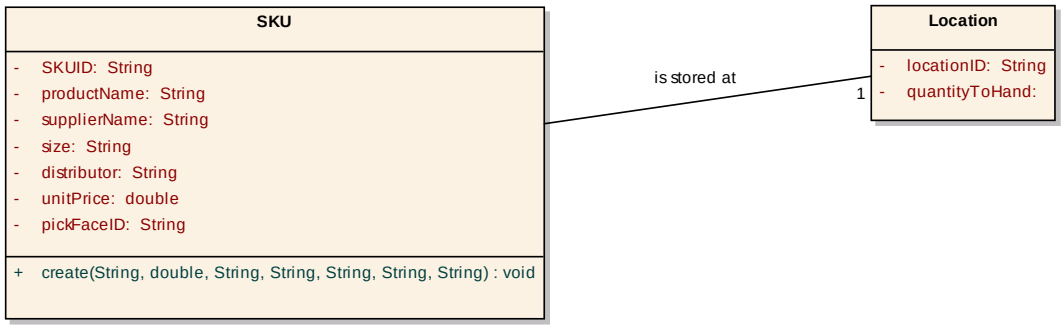


Figure 163 - Each SKU is stored at one location

Recall that during design, we didn't hold "referenced" classes in an attribute using a "foreign key". At coding, however, we do have to reference the other class. We have to decide in which direction the visibility should be (can the SKU see the location, or can the location see the SKU, or both?) In this example, lets assume that the SKU needs to be able to see the location. Also, in some languages, a "by value" or "by reference" decision needs to be made.

```
public class SKU{
   private String SKUID;
   private String productName;
   private String supplierName;
   private String size;
   private String distributor;
   private double unitPrice;
   private String pickFaceID;

   private Location storedLocation;
}
```

Figure 164 - holding the location

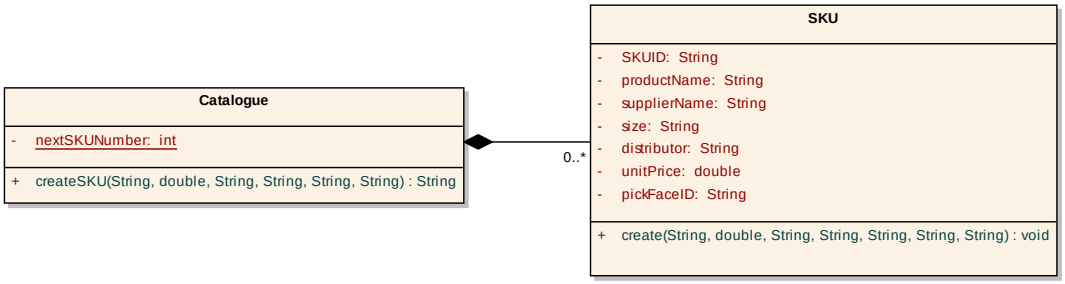
Essentially, the Location object is simply held as an attribute of the SKU class.

Containers/Collections

When we have an aggregation, these map on to the target language as either arrays or collections.

Arrays are the classic programming construct that are of a fixed size, and usually the size of the array has to be set at compilation time. Collections are a more flexible alternative, and are usually of an unbounded length (they can grow and shrink as elements are added and removed).

There's nothing new to the concept of Collections (Linked Lists for example are a very popular form of collection), but they are now becoming standardized in the libraries of modern languages. Java has a Collections framework featuring Lists, Map, Sets and so on, whilst C++ has a Standard Template Library (STL) offering similar features. The .NET framework contains collection classes, so programmers in any of C#.NET, VB.NET, C++.NET and any other .NET language, have access to them.




```
public class Catalogue
{
   private static int nextSKUNumber = 1;

   private SKU[]  skuList; // array
   private Map    skuList; // key/value map
   private List   skuList; // list
}
```

Figure 165- Example Java Implementation

The Java example above shows how the list of SKU's held in the Catalogue might be held in an Array, a Map (a kind of table of keys and values) and a List (a dynamic form of array).

Coding a Use Case

All we've shown so far is how to map a class to a file on your target language. This is really an easy step – the hard work is in building a full Use Case. We won't step through all of the tedious detail, and we won't fill this chapter with reams of code. Instead, we have provided an appendix and included implementations (in different languages) for a very simple Use Case. See "es at the ba", page 210).

Testing a Use Case

The test script for the Use Case test is already written - it is the Use Case Storyboard!

The “Actors Actions” form the steps to carry out in the test. The “System Response” forms the expected responses.

Code Generation

Some tools support Code Generation. Usually, this means the generation of class skeletons from the class diagram (roughly to the level of detail of the classes we’ve looked at in this chapter).

Some tools will do **100% generation,** where the model is identical to the source code, and changing one will change the other. This is an attractive idea, but be aware this will naturally mean “busy” diagrams in your model, with lots of detail.

Another concept that many tools support is **reverse engineering**. This is the process of reading existing code and automatically generating a model from the code. A jargon word in common use by tool vendors is “Round Tripping”; this just means the tool can perform both forward and reverse engineering.

Reverse Engineering sounds like a good idea, but it will not, in general, create a readable or understandable model. It is certainly **not** a substitute for up-front design. It is useful if you simply need to resynch a model that has drifted from the code, or to discover design flaws in existing code.

At Ariadne, we tend to be quite suspicious of code generation – if you have a success story to tell about it, please get in touch with us and tell all.

Implementation Frameworks

.NET (Microsoft) and J2EE (Java Enterprise Edition) are two of the big frameworks for implementing Enterprise Systems in the industry at the moment

We’ll have a very brief look at both of them now…

J2EE

J2EE features the concept of EJB's (Enterprise Java Beans), which are effectively classes that provide additional "enterprise" services - especially Transaction Management but they also hide network and distribution "plumbing".

A further feature of EJB's is that they can be made "transparently persistent" - in other words, data in an EJB class will be automatically persisted to a relational data store, closing the relational/object mismatch (but this concept is not a popular implementation choice at present).

Other technologies such as JDO (Java Data Objects) exist to close the object/relational gap.

J2EE and MVC

J2EE is built around the framework of the Model-View-Controller. Session EJB's and servlets fulfil the role of a Use Case Controller extremely well.

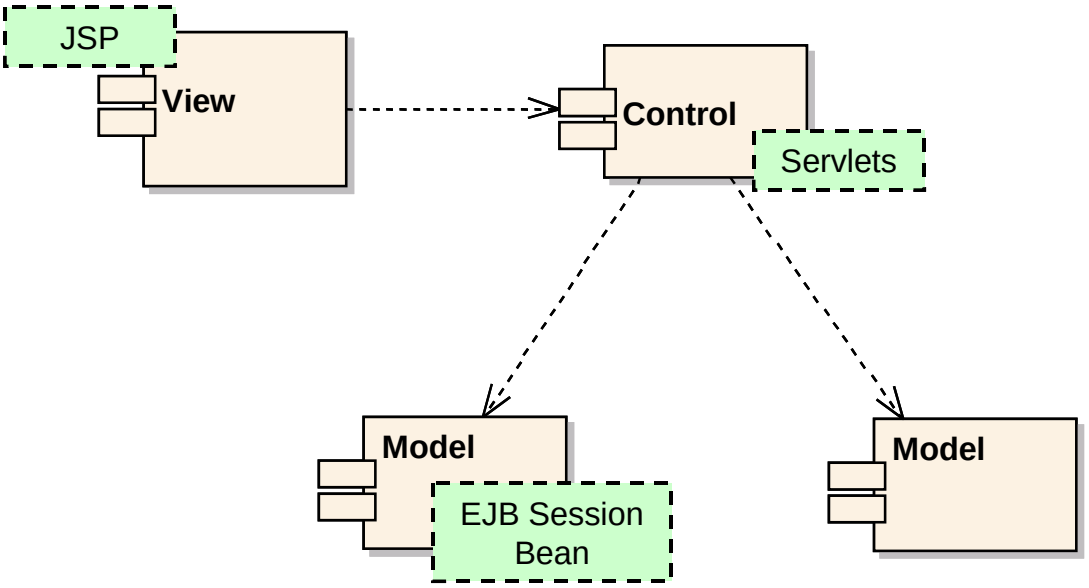


Figure 166 - A J2EE Component Model

.NET

.NET features ADO.NET(ActiveX Data Objects) to provide the relational database "bridge".

All of .NET's "prime" languages (C#.NET, VB.NET) are now fully object oriented and feature everything we have covered on this course. They also feature some syntactic sugar to hide get/set methods from the client programmer.

.NET Remoting and SOAP enables objects to be distributed.

Summary

The UML models should map cleanly into code, and most modern languages support the OO "way of thinking" directly.

When a Use Case is coded, it can be fully tested using the original Formal Description as (at least) a basis for the test description.

Forward and Reverse Engineering is possible in many Case tools; make sure your project carefully evaluates them before use however.

Appendix A

Example Code

As most attendees to the course are not interested in programming, we have decided to put full code examples at the back of the book, well away from the UML.

All we are trying to do with this example is to show that it *is* possible to transfer designs from UML to source code, and that the model/view/controller design pattern does work. We are *not* displaying best programming practice, so please don't take any of the examples too seriously. Running versions of this code (and a fuller implementation of the complete first iteration) will be provided on the course.

We've taken an extract from a Use Case that you will have designed during the course. This is the "Enter SKU" Use Case. You may have designed a fancier implementation, but we're just going for the simple case where the user enters a product name, a supplier name and unit price. Inside the Stock component, a new SKU is created with an auto generated ID, and the user is told that the SKU has been created, and they are told what the new SKU ID is.

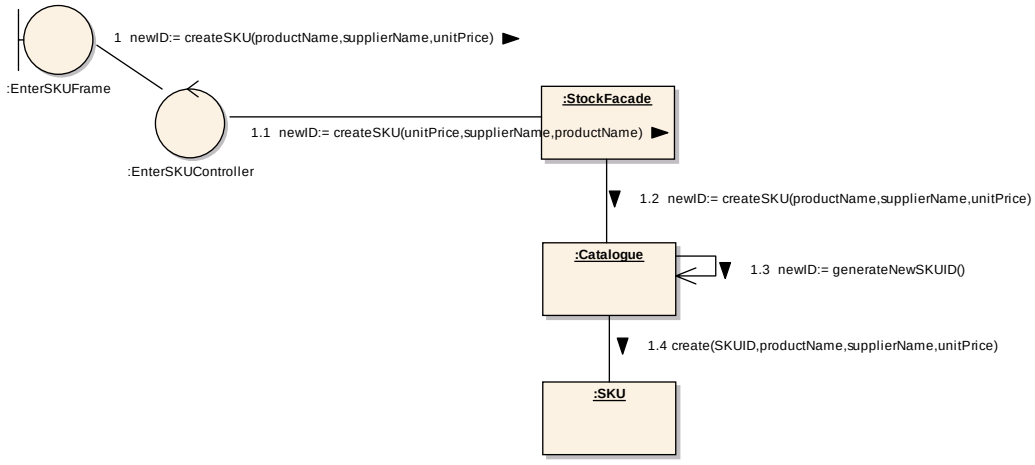
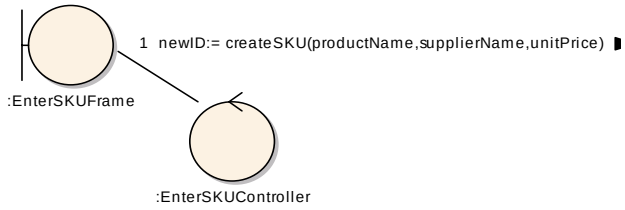


Figure 167 - The Use Case Collaboration

Java Code

We are going to start from the top and work downwards (just our personal preference). The user has entered a product name, a supplier name and a price in the user interface (we call this the “EnterSKUFrame”). A message is sent on to the EnterSKUController called “createSKU”, which returns the new ID for the SKU (this is because the customer has asked for the automatic generation of SKU’s).



Here’s our code for this. We have removed any GUI code as that is irrelevant to this discussion. We have “hardcoded” in some example data that the user might have entered…

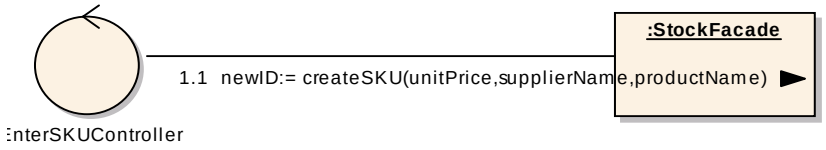
File : EnterSKUFrame.java

```
package pharmacare.ui;

import pharmacare.control.*;

public class EnterSKUFrame
{
   /*
    * The constructor method simulates GUI Activity
    */
   public EnterSKUFrame()
   {
     EnterSKUController control = new EnterSKUController();

     // assume the user has entered some data
     // details of GUI etc omitted

     String newID = control.createSKU("Atenolol","SKB Limited”,4.99);
     System.out.println("New SKU ID " + newID + " created.");

     newID = control.createSKU("Paracetemol","KCS Ltd",0.72);
     System.out.println("New SKU ID " + newID + " created.");
   }

   public static void main(String[] args)
   {
      EnterSKUFrame window = new EnterSKUFrame();
```

```
   }
}
```

File : EnterSKUController.java

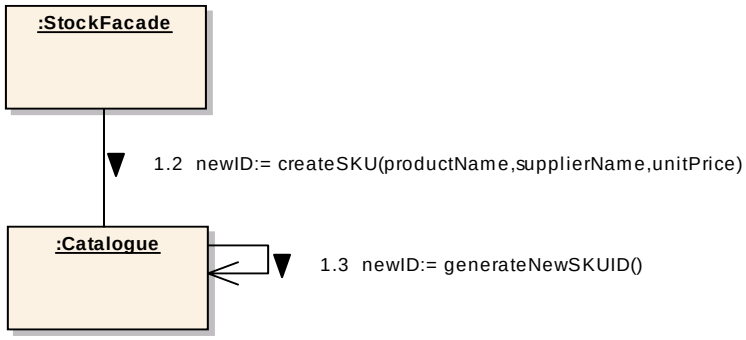
The implementation of the createSKU() Method in the EnterSKUController simply passes on the request to the StockFacade...




```
package pharmacare.control;

import pharmacare.domain.stock.StockFacade;

public class EnterSKUController
{
  private StockFacade stockComponent;

  public EnterSKUController()
  {
     // get visibility of the stock component
     stockComponent = StockFacade.getReference();
  }

  public String createSKU(String productName, String supplierName,
                          double unitPrice)
  {
     return stockComponent.createSKU
                         (productName, supplierName, unitPrice);
  }
}
```

To get access to the StockFacade, we haven't called **new** as usual; instead we have called the static method getReference(), because the StockFacade has been coded as a Singleton. See the chapter on Design Patterns for full details.

File : StockFacade.java



The implementation of the createSKU method in StockFacade simply forwards the request to the Catalgoue. The Catalogue class is our “collection” class that holds all of the SKU Records.

The complication in the Facade class is the mechanics of the Singleton. Once again, see the chapter on Design Patterns for details...

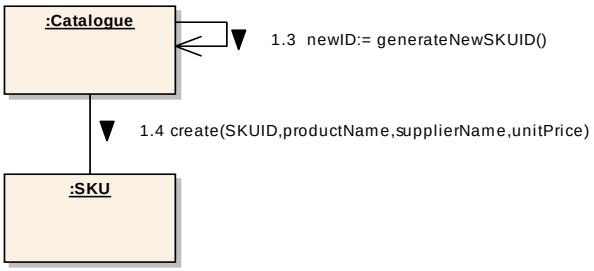
```
package pharmacare.domain.stock;

public class StockFacade{

   private static StockFacade reference = null;
   private Catalogue theCatalogue;

   // this is a singleton class, so the constructor is private
   private StockFacade()
   {
     theCatalogue = new Catalogue();
   }

   public static StockFacade getReference()
   {
     if (reference == null)
       reference = new StockFacade();
     return reference;
   }

   public String createSKU(String productName, String supplierName,
                           double unitPrice)
   {
      return theCatalogue.createSKU
                       (productName, supplierName, unitPrice);
   }
}
```

Notice that we construct the Catalogue in the constructor for the Facade. Remember that this will only ever happen **once**, regardless of how many times the facade is accessed.

File : Catalgoue.java

The createSKU method in Catalgoue needs to:

- Generate a new SKU ID (method 1.3 on the Collaboration diagram)
- Create the SKU Object (method 1.4)
- Store the new SKU in a Collection (this could have been an array, we've opted for a Map with the SKUID forming the key)




```
package pharmacare.domain.stock;

import java.util.*;

public class Catalogue
{

  private Map skuList;
  private static int nextSKUNumber = 1;

  public Catalogue()
  {
    skuList = new HashMap();
  }

  private String generateNewSKUID()
  {
    String returnValue = "SKU" + nextSKUNumber;
    nextSKUNumber++;
    return returnValue;
  }

  public String createSKU(String productName, String supplierName,
                          double unitPrice)
  {
    String SKUID = this.getNextSKUID();
    SKU newSKU = new SKU(SKUID, productName,supplierName,unitPrice);
    skuList.put(SKUID,newSKU);
    return SKUID;
  }
}
```

Notice that we have made the generateNewSKUID method **private**, as it should not be called from outside of the class. We've also provided a very bland and boring implementation of the method, just for illustration!

File : SKU.java

All that remains is the creation of the SKU Class. There is very little behaviour in here at present; only data is stored in the SKU. We would expect this to change as future iterations add more behaviour to the class diagram.

Notice that the class is **package protected**, as recommended by the Facade design pattern.

```
package pharmacare.domain.stock;

import java.util.*;

/**
 * Definition of a Stock Keeping Unit
 */
class SKU
{
   private String SKUID;
   private String productName;
   private String supplierName;
   private String size;
   private String distributor;
   private double unitPrice;
   private String pickFaceID;

   public SKU(String SKUID, String proudctName, String supplierName,
              double unitPrice)
   {
      this.SKUID = SKUID;
      this.productName = productName;
      this.supplierName = supplierName;
      this.unitPrice = unitPrice;
   }
}
```

C Sharp Code

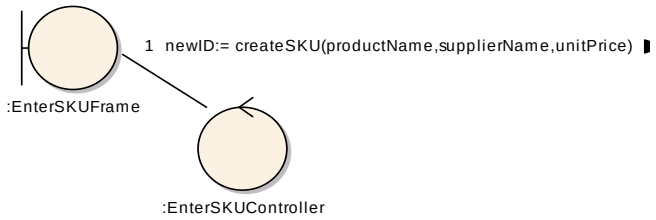
The code for C# is almost identical to the Java implementation, except:

- Use namespaces instead of packages
- Use a .NET collection class instead of the Java HashMap (for example, hashtable)

C++ Code

It is impossible to write compiler independent C++, so the following should be used as a guide only. For the record, we used Borland C++ Builder 6 on Windows XP.

We are going to start from the top and work downwards (just our personal preference). The user has entered a product name, a supplier name and a price in the user interface (we call this the “EnterSKUFrame”). A message is sent on to the EnterSKUController called “createSKU”, which returns the new ID for the SKU (this is because the customer has asked for the automatic generation of SKU’s).



Here’s our code for this. We have removed any GUI code as that is irrelevant to this discussion. We have “hardcoded” in some example data that the user might have entered…

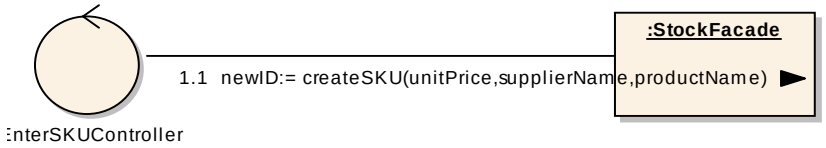
File : EnterSKUFrame.cpp

```
//---------------------------------------------------------
#include <iostream.h>
#include <conio.h>
#include "EnterSKUControl.h"

//---------------------------------------------------------

int main()
{
    EnterSKUController control;
    string newSKU = control.createSKU("Atenolol","SKN Ltd",3.24);
    cout << "New SKU Reference " << newSKU << " created." << endl;
```

```
    newSKU = control.createSKU("Paracetomol","SKN Ltd",0.23);
    cout << "New SKU Reference " << newSKU << " created." << endl;
}
//-------------------------------------------------------
```

File : EnterSKUControl.h / .cpp

The implementation of the createSKU() Method in the EnterSKUController simply passes on the request to the StockFacade...

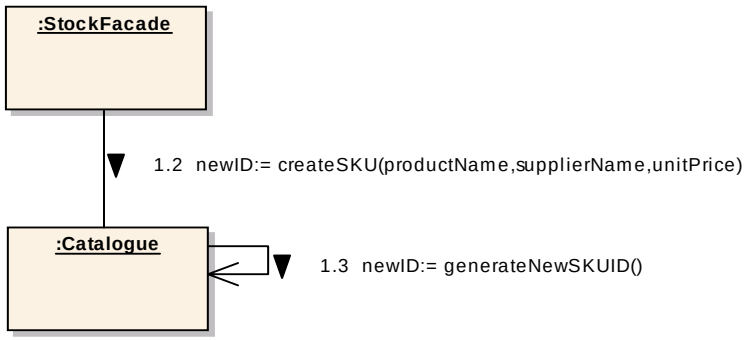



```
// EnterSKUController.h

#include "StockFacade.h"

class EnterSKUController
{
public:
EnterSKUController();
      string createSKU(string,string,double);

private:
      StockFacade* stockFacade;
};
```

```
// EnterSKUController.cpp

#include <iostream.h>
#include "StockFacade.h"
#include "EnterSKUControl.h"

EnterSKUController::EnterSKUController()
{
   // get a reference to the StockFacade
  stockFacade = StockFacade::getReference();
}

string EnterSKUController::createSKU(string product,
                                      string supplier,double price)
{
    return stockFacade->createSKU(product, supplier, price);
}
```

To get access to the StockFacade, we haven't called **new** as usual; instead we have called the static method getReference(), because the StockFacade has been coded as a Singleton. See the chapter on Design Patterns for full details.

File : StockFacade.h / .cpp



The implementation of the createSKU method in StockFacade simply forwards the request to the Catalgoue. The Catalogue class is our "collection" class that holds all of the SKU Records.

The complication in the Facade class is the mechanics of the Singleton. Once again, see the chapter on Design Patterns for details...

```
class StockFacade
{

private: // singleton
    StockFacade();
    Catalogue* theCatalogue;

public:
    static StockFacade* getReference();
    string createSKU(string,string,double);
};
```

.cpp file follows on the next page...

StockFacade.cpp

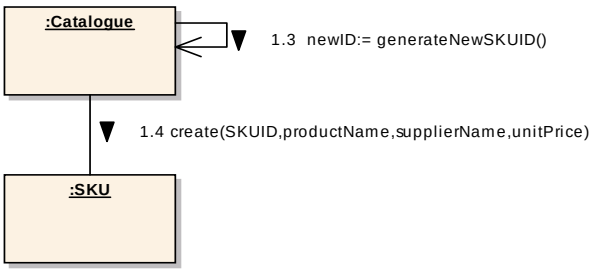
```
#include <iostream.h>
#include "Catalogue.h"
#include "StockFacade.h"

StockFacade::StockFacade()
{
 //private constructor
      theCatalogue = new Catalogue();
}

StockFacade* StockFacade::getReference()
{
     static StockFacade reference;
     return &reference;
}

string StockFacade::createSKU(string product,
                              string supplier,double price)
{
   return theCatalogue->createSKU(product, supplier, price);
}
```

Notice that we construct the Catalogue in the constructor for the Facade. Remember that this will only ever happen **once**, regardless of how many times the facade is accessed.

File : Catalgoue.h / .cpp

The createSKU method in Catalgoue needs to:

- Generate a new SKU ID (method 1.3 on the Collaboration diagram)
- Create the SKU Object (method 1.4)
- Store the new SKU in a Collection (this could have been an array, we've opted for a vector)




```
#include <iostream.h>
#include <vector>
#include "SKU.h"

class Catalogue
```

```
{
private: // singleton
    static int nextSKUNumber;
    vector<SKU> skuList;

public:
    Catalogue();
    string generateNewSKUID();
    string createSKU(string,string,double);
};
```

.cpp file follows on next page...

Catalogue.cpp:

```
#include <iostream.h>
#include <sstream.h>
#include "Catalogue.h"
#include "SKU.h"

int Catalogue::nextSKUNumber = 1;

string Catalogue::generateNewSKUID()
{
   string retStr("SKU");
   std::ostringstream o;
   o << nextSKUNumber++;
   retStr += o.str();
   return retStr;
}

string Catalogue::createSKU(string product,string supplier,
                            double unitPrice)
{
   string SKUID = this->generateNewSKUID();
   SKU newSKU (SKUID, product, supplier, unitPrice);
   skuList.push_back(newSKU);
   return SKUID;
}
```

Notice that we have made the generateNewSKUID method **private**, as it should not be called from outside of the class. We've also provided a very bland and boring implementation of the method, just for illustration!

File : SKU.h / .cpp

All that remains is the creation of the SKU Class. There is very little behaviour in here at present; only data is stored in the SKU. We would expect this to change as future iterations add more behaviour to the class diagram.

Unfortunately there is no way to prevent other classes in the system from accessing this class, even though our design placed this class behind a facade.

```
#include <iostream.h>

class SKU
{

private:
  string SKUID;
        string productName;
        string supplierName;
        double unitPrice;

public:
```

```
        SKU(string, string, string, double);
};
```

```
#include "SKU.h"

SKU::SKU(string SKUID, string productName, string supplierName,
         double unitPrice)
{
   this->SKUID = SKUID;
   this->productName = productName;
   this->supplierName = supplierName;
   this->unitPrice = unitPrice;
}
```

Ada Code

In Ada we can dispense with Facades and Singletons, as we think that Ada's package construct provides a very effective mechanism for achieving the same thing.

We built this system using two packages, **control** and **stock**.

File : main.ada

```
with control;
with stock;
with Text_IO;

procedure main is
   newSKU      : stock.SKUID_Type;
   ProductName : stock.Product_Name_Type;
   Supplier    : stock.Supplier_Name_Type;
begin
   ProductName(1..8) := ("Atenolol");
   ProductName(9..stock.product_Name_Type'length) := (others => ' ');

   supplier(1..7) := "SKN Ltd";
   supplier(8..stock.supplier_name_type'length) := (others => ' ');
   newSKU := control.create_SKU(productName,supplier,3.24);

   ProductName(1..11) := ("paracetemol");
   ProductName(9..stock.product_Name_Type'length) := (others => ' ');
   NewSKU := Control.create_SKU(productName,supplier,0.32);
end main;
```

Control Package

The control package exposes the following method:

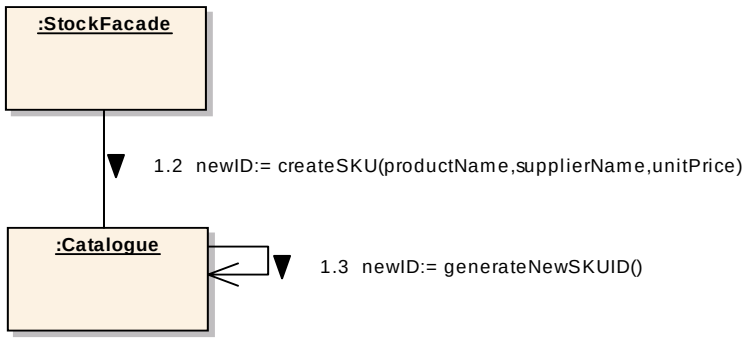
```
with stock;

package control is
```

```
   function create_SKU (product_Name     : stock.product_name_type;
                        Product_Supplier : stock.supplier_name_type;
unit_Price       : float)
                        return stock.skuID_Type;

end Control;
```

The package body implements the create_SKU method as follows:

```
with stock;

package body control is

   function create_SKU (product_Name     : stock.product_name_type;
                        Product_Supplier : stock.supplier_name_type;
unit_Price       : float)
                        return stock.skuID_Type is

   begin
return stock.create_SKU(product_Name,
                              product_Supplier,
                              unit_Price);
   end create_SKU;

end control;
```

Stock Package



The Stock Package has the following specification:

```
package Stock is

   type SKU       is private;
   type catalgoue is private;

   subtype SKUID_Type      is String(1..10);
   type Product_Name_Type  is new String(1..50);
   type supplier_Name_Type is new String(1..50);
```

```
   function create_SKU(Product_Name     : product_Name_Type;
                       Product_Supplier : Supplier_Name_Type;
     unit_Price       : float)
                       return SKUID_Type;

   private

     type SKU is record
         SKUID     : SKUID_Type;
         Product_name  : product_Name_Type;
   supplier_Name : supplier_Name_type;
   unit_Price    : float;
     end record;

     type catalgoue is array(1..50) of SKU;

end Stock;
```

Notice that the “generateNewSKUID” is not present in the package specification – this means it will be private to the package.

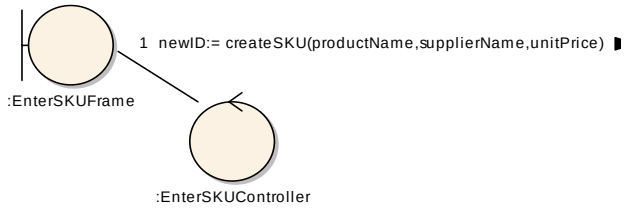
The package body follows on the next page...

Stock Package Body...

```
with Text_IO;

package body Stock is

   nextSKUID         : Integer := 1;
   catalogue         : array(1..50) of SKU;

   procedure Initialise (this         : out SKU;
       SKUID  : SKUID_Type;
 name         : product_Name_type;
 supplier     : supplier_name_type;
 price        : float) is
   begin
      This.SKUID          := SKUID;
      This.Product_Name   := Name;
This.Supplier_Name  := Supplier;
This.Unit_Price     := price;
   end initialise;

   function generate_new_SKU return SKUID_type is
      newSKU : SKUID_Type;
   begin
      -- add your own implementation here
-- quick and dirty for illustration only
newSKU(1..3) := "SKU";
newSKU(4..SKUID_Type'length) := (others => ' ');
return newSKU;
   end generate_New_SKU;


   function create_SKU (product_Name      : stock.product_name_type;
                        Product_Supplier : stock.supplier_name_type;
 unit_Price       : float) return skuID_Type
   is
      new_SKU   : SKU;
      NewSKU_ID : SKUID_Type;
   begin

      NewSKU_ID := generate_New_SKU;

      Initialise(This     => New_SKU,
           SKUID    => NewSKU_ID,
           Name     => product_Name,
     Supplier => product_Supplier,
     Price    => Unit_Price);

      Catalogue(nextSKUID) := new_SKU;
      nextSKUID := NextSKUID + 1;

return newSKU_ID;

   end create_SKU;

end stock;
```

Visual Basic.NET Code

We are going to start from the top and work downwards (just our personal preference). The user has entered a product name, a supplier name and a price in the user interface (we call this the “EnterSKUFrame”). A message is sent on to the EnterSKUController called “createSKU”, which returns the new ID for the SKU (this is because the customer has asked for the automatic generation of SKU’s).



Here’s our code for this. We have removed any GUI code as that is irrelevant to this discussion. We have “hardcoded” in some example data that the user might have entered.

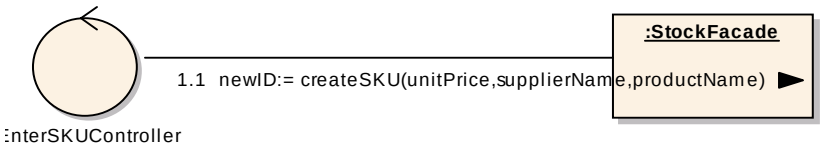
Note that all line breaks that appear in the middle of lines of code have been added by our Word Processor and weren’t present in the original source files.

File : EnterSKUFrame.vb

```
Module MainForm

    Private control As control.EnterSKUController

    Sub Main()
        control = New control.EnterSKUController()
        Console.WriteLine
              ("Created: " + control.createSKU("Atenolol", 7.42))
        Console.WriteLine
              ("Created: " + control.createSKU("Paracetemol", 0.23))
    End Sub

End Module
```

File : EnterSKUController.vb

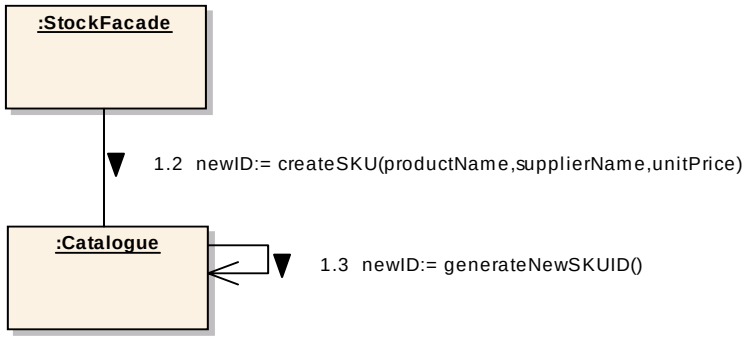
The implementation of the createSKU() Method in the EnterSKUController simply passes on the request to the StockFacade...




```
Namespace control

    Public Class EnterSKUController

        Public Function createSKU
                    (ByVal name As String, ByVal price As Double)
            Dim StockComponent As Stock.StockFacade
            StockComponent = StockComponent.getReference()
            Return StockComponent.createSKU(name, price)
        End Function

    End Class
End Namespace
```

To get access to the StockFacade, we haven't called **new** as usual; instead we have called the shared (static) method getReference(), because the StockFacade has been coded as a Singleton. See the chapter on Design Patterns for full details.

File : StockFacade.vb



The implementation of the createSKU method in StockFacade simply forwards the request to the Catalgoue. The Catalogue class is our "collection" class that holds all of the SKU Records.

The complication in the Facade class is the mechanics of the Singleton. Once again, see the chapter on Design Patterns for details...

```
Namespace Stock

    Public Class StockFacade
        Private Shared reference As StockFacade
```

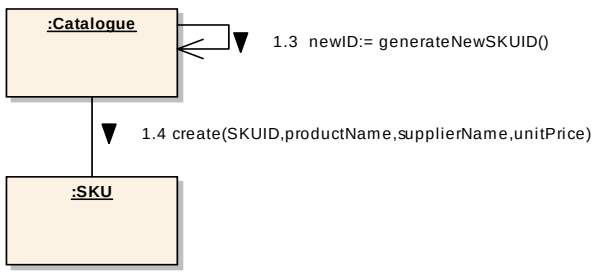
```
        Private Shared created As Boolean
        Private StockCatalogue As Catalogue

        Shared Function getReference() As StockFacade
            If created = False Then
                reference = New StockFacade()
                created = True
                Return reference
            Else
                Return reference
            End If
        End Function

        Private Sub New()
            '' singleton class => private constructor
            StockCatalogue = New Catalogue()
        End Sub

        Public Function createSKU
                        (ByVal name As String, ByVal price As Double)
            Return StockCatalogue.createNewSKU(name, price)
        End Function

    End Class

End Namespace
```

Notice that we construct the Catalogue in the constructor for the Facade. Remember that this will only ever happen **once**, regardless of how many times the facade is accessed.

File : Catalgoue.vb

The createSKU method in Catalgoue needs to:

- Generate a new SKU ID (method 1.3 on the Collaboration diagram)
- Create the SKU Object (method 1.4)
- Store the new SKU in a Collection (this could have been an array, we've opted for a Hashtable with the SKUID forming the key)




```
Namespace Stock

    Public Class Catalogue
```

```
        Private SkuList As Collections.Hashtable
        Private Shared nextSKUNumber As Integer = 1

        Public Sub New()
            SkuList = New Collections.Hashtable()
        End Sub

        Private Function generateNewSKUID() As String
            Dim ReturnString As String
            ReturnString = "SKU"
            ReturnString = ReturnString + Str(nextSKUNumber)
            nextSKUNumber = nextSKUNumber + 1
            Return ReturnString
        End Function

        Public Function createNewSKU
              (ByVal name As String, ByVal price As Double) As String
            Dim newSKUID As String
            newSKUID = generateNewSKUID()
            Dim newSKU As New SKU(newSKUID, name, price)
            SkuList.Add(newSKUID, newSKU)
            Return newSKUID
        End Function

    End Class

End Namespace
```

Notice that we have made the generateNewSKUID method **private**, as it should not be called from outside of the class. We've also provided a very bland and boring implementation of the method, just for illustration!

File : SKU.vb

All that remains is the creation of the SKU Class. There is very little behaviour in here at present; only data is stored in the SKU. We would expect this to change as future iterations add more behaviour to the class diagram.

```
Namespace Stock

    Class SKU

        Private SKUID As String
        Private productName As String
        Private unitprice As Double

        Public Sub New(ByVal m_SKUID As String,
                       ByVal m_productName As String,
                       ByVal m_unitprice As Double)
            Me.SKUID = m_SKUID
            Me.productName = m_productName
            Me.unitprice = m_unitprice
        End Sub
```

```
    End Class

End Namespace
```

Recommended Books

These are our favourite books related to the subject of this course. Not all of them are referenced directly by the course – many of them are completely independent of the UML and OO, but they all have some bearing on the topics we have covered.

[1] : Scott, Kendall 2002 ***The Unified Process Explained*** Addison-Wesley ISBN: 0201742047

A full overview of the UP.

[2] : Larman, Craig. 2001 ***Applying UML and Patterns An Introduction to Object Oriented Analysis and Design*** Prentice Hall ISBN: 0130925691

Larman *applies* the UML rather than getting lost in the mass of detail that is often associated with the UML. An excellent approach. The second edition uses the Unified Process rather than the homebrew process used in the first edition. Strongly recommended.

[3] Shalloway/Trott 2001 ***Design Patterns Explained: A New Perspective on Object-Oriented Design*** Addison Wesley ISBN: 0201715945

A much more readable guide to design patterns than the "Gang of Four" book. It lacks the rigour or the detail of GoF, but this book is definitely more approachable.

[4] : Collins, Tony ***Crash – Learning from the World's Worst Computer Disasters*** Simon & Schuster ISBN: 0684868350

If you don't believe that the Software Engineering industry is in a crisis, and that as an industry we need to sharpen up our game, read this entertaining book about some of the most widely publicised computer disasters of the last few years.

[5] : Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995 ***Design Patterns : Elements of Reusable Object Oriented Software*** Addison-Wesley ISBN: 0201633612

The classic "Gang of Four" catalogue of several design patterns. Hard work for sure, but useful if you wish to go further with your OO and give your brain some serious exercise.

[6] : Riel, Arthur 1996 ***Object Oriented Design Heuristics*** Addison-Wesley ISBN: 020163385X

A Heuristic is a "rule of thumb"; this book is a good guide to how to apply OO properly. The book is feeling its age a little now, but it certainly has some excellent points to make and whilst it lacks the depth or breadth of the Gang of Four book, it is certainly more readable.

[7] : Glass, Robert L ***Facts and Fallacies of Software Engineering*** Addison-Wesley ISBN: 0321117425

A superb book that covers 55 facts (at least, the what the author believes to be facts) about Software Engineering. We've included the book here because of the excellent coverage on components and reuse, but the rest of the book too is a must-read for all software engineers.

[8] : Fowler, Martin 2002 ***Patterns of Enterprise Application Architecture*** Addison-Wesley ISBN: 0321127420

Aimed at developers working on platforms such as .NET or J2EE, this book features some interesting thoughts on modern development issues – such as the mapping between objects and databases.

[9] : Cockburn, Alistair 2000 ***Writing Effective Use Cases*** Addison-Wesley ISBN: 0201702258

Alistair Cockburn's book is one of the few books that cover how to write Use Case Descriptions.

[10] : Elisabeth Freeman, Eric Freeman, Bert Bates 2004 ***Head First Design Patterns*** O'Reilly ISBN: 0596007124

At last! A book on design patterns that is readable, understandable and entertaining. And it is *deep* – it covers full OO design principles in FAR more detail than we can on a one week course.