Software Engineering

In a 21st century knowledge-based economy, computer software is of vital importance. For instance, the UK software industry is worth around £30 billion[1] per annum, which equates to 3% of the total UK economy. Furthermore, software infrastructure of some kind underpins almost everything we do in modern society.

What is Software?

Software is the combination of instructions and data that controls electronic computer hardware. Almost all electronic devices are programmable, which means they rely on software to direct their behavior. Software is excitingly and dangerously unconstrained. Brooks[2] states: 'Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.'

What is Software Engineering?

We can instinctively recognize bad software engineering. We all experience frustration when computer program are unreliable and crash, losing our work. Again, we detest computer systems that refuse to allow us to carry out the supposedly simple tasks we require to do, such as unhelpful self-service checkouts[3] at supermarkets.

So, how can we encapsulate good software engineering? Which principles are at work to produce high-quality software, and what does quality really mean in this context?

Small programs can be produced by individuals, who have a sufficient grasp of their work that it can be self-consistent, useful and reliable. However the Linux operating system kernel has over 13 million lines of code[4], far more than even Linus Torvalds can manage single-handedly. Good software engineering practice gives Linux, and other large software projects, the following key properties:

1. *Utility* – it performs a useful task, as expected by its users
2. *Efficiency* – it works quickly on current hardware platforms
3. *Reliability* – it fails infrequently

[1] http://www.bis.gov.uk/policies/business-sectors/electronics-and-it-services/software-and-it-services, fetched on 20th Dec 2010
[2] **THE MYTHICAL MAN MONTH**, by Fred Brooks. Any edition will do. I have the 1995 Addison Wesley version.
[3] http://bit.ly/bnIvSN, fetched on 20th Dec 2010
[4] http://www.h-online.com/open/features/What-s-new-in-Linux-2-6-36-1103009.html?page=6, fetched on 20th Dec 2010

4. *Maintainability* – it can be repaired and extended as and when deemed necessary by its developers
5. *Elegance* – it has an architectural coherence

The discipline of software engineering concerns creating software products with the above qualities. One further issue is the managerial aspect of production: Well-engineered software must be delivered on-time and in-budget.

In summary, Sommerville[5] defines software engineering as:

The production of high-quality software with a finite amount of resources and to a predicted schedule,

with the following four attributes that relate to quality:

Maintainable, reliable, efficient, and with an appropriate user interface.

Software Disasters

High-quality software is the kind of entity that should never be noticed by its users. It empowers end-users to accomplish their tasks swiftly and smoothly. However all too often, we hear of software disasters, where the software engineering process has malfunctioned to some extent. Fred Brooks[6] gives a historical account of a 1960's software disaster. The NHS IT project[7] appears to be a modern-day parallel.

[5] **Software Engineering**, by Ian Sommerville. Any edition will be helpful, the most recent is the 9th edition, published by Addison Wesley in 2010.
[6] **The Mythical Man Month**, by Fred Brooks
[7] http://www.connectingforhealth.nhs.uk/, fetched on 20th Dec 2010. See also http://nhs-it.info/, fetched on 11th Jan 2010.

Software Scaling

A teenager working in his bedroom[8] can develop a small software project such as an iPhone app. However larger scale software such as an office application suite takes many person-years of development effort[9]. So how does Microsoft manage to release a new version of its *Office* product every 3 years? In this section, we consider the differing roles played by various members of the software team, and how they combine to produce high-quality, significant size software products.

Software Deliverables

Until now, we may have considered software to be merely an executable program, supplied on a DVD or via internet download. However a software project has many more deliverables. These include:

1. A high-level specification of the software
2. An architectural overview of the software modules
3. Source code
4. Documentation for the source code
5. Revision history of the source code
6. Bugs database
7. Test sets
8. Documentation for software users
9. Post-release maintenance updates or extensions

Each of these deliverable units is important in its own right, and requires effort on the part of the development team.

Modularity

To manage a large and complex piece of software, it must be divided into manageable components, often referred to as *modules*. Each module is developed by a small, closely-knit team. The *interfaces* between modules are clearly specified to allow work on multiple components to proceed in parallel. An interface is a form of *contract* between the inter-dependent modules, and their corresponding development teams. It specifies the behaviour that can be expected from a module.

For example, the Firefox web browser has around 30 modules[10] ranging from `AccountManager` to `Toolbars`.

[8] http://www.fastcompany.com/1621539/teen-iphone-app-developers, fetched on 21st Dec 2010

[9] http://www.ohloh.net/p/openoffice , fetched on 21st Dec 2010. OpenOffice is a free clone of MS Office.

[10] https://bugzilla.mozilla.org/describecomponents.cgi?product=Firefox , fetched on 7th Jan 2011. Note that Firefox also includes many of the Mozilla core modules.

Teams

Large teams have high communication overhead. Hence Brooks proposes that a software team should consist of ten or fewer members. Communication remains manageable at this size. While some of Brooks' original writing may sound hopelessly outdated in terms of filing assistants for punch cards, etc, many of his observations about software teams remain relevant today.

The team will be led by a **software architect**, who is responsible for the overall structure and integrity of the team's module. He will have responsibility for communicating with other teams who are developing other modules that interface with his module. The architect may double up as team **manager**.

A small number of individuals within the team will undertake the actual **software development**. These will be experienced software engineers, although novices may gain experience by pair-programming with a more experienced colleague. All software developers will peer-review each other's coding efforts, since they will use a shared online code repository.

Software documentation is essential. This may be written directly by the developers, as they produce their code. The *literate programming* philosophy advocates this approach. Alternatively, documentation may be added by a **documentation specialist** who works alongside the developers. Code documentation allows other people to understand how the program works, which may be necessary if it has to be modified or extended at a later stage.

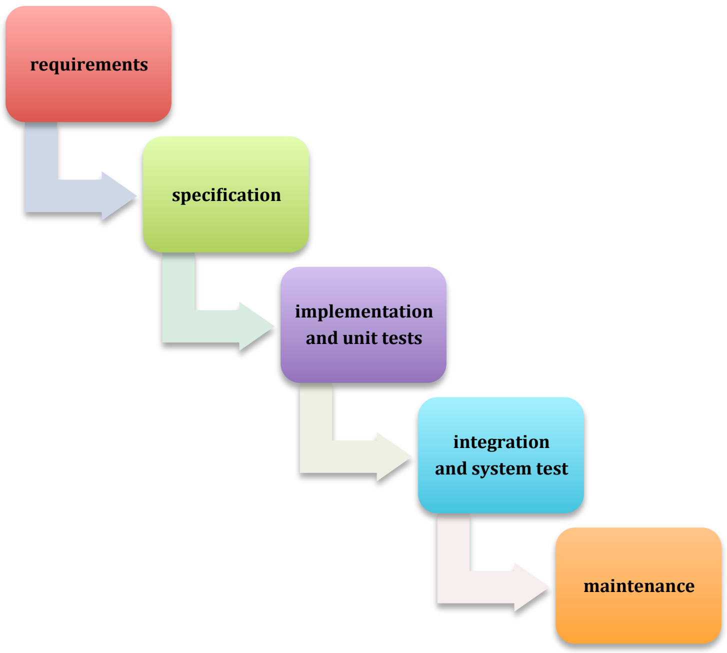
A small number of individuals within the team will devise and create **tests** for the software module. These are known as *unit tests*. Some of these will be black box tests (simply ensuring that the module behaves according to its interface, so that other modules can interact with it correctly). Other tests will be white box tests, when the tester assumes certain knowledge of the inner workings of the module and writes tests accordingly. An automated test suite, with regression tests running regularly, will be assembled and maintained by the testers.

A **tools specialist** or technician may provide further support for the team. This person ensures that the developers and testers have appropriate machines and development software to accomplish their tasks. Non-technical support is provided by a **secretary**. These support roles may be shared between multiple teams.

Project manager	Documentation writer
Software developer	Support technician
Software tester	Software architect
Secretary	

Software Development Lifecycle

The *software crisis* hit in the 1960s. In this second decade of computerization, processors had become sufficiently advanced and memory sufficiently large that ad-hoc informal software development techniques failed. Royce[11] suggested the Waterfall model of software development, in which a software project proceeds through various sequential phases. Phase *n* must be completed and approved before Phase *n+1* commences.



Advantages of Waterfall

1. Early capture of **requirements** sets the level of expectations for users and developers. The aim is to avoid feature-creep entirely, since the lifecycle does not admit changed requirements once this stage has been approved.
2. An explicit **specification** stage maps out the architecture of the software product. This enables modularization, and early assessment of technical challenges and performance issues.
3. Clear provision of **testing** of individual modules and the full system, which must be completed before the software is deployed.
4. An ongoing **maintenance** phase deals with software bugs, performance tuning and requested extensions.

Disadvantage of Waterfall

The **lack of flexibility** is a problem. Once a particular phase has been completed, there is no facility to revisit it and revise its deliverables. Waterfall is not able to handle a changing problem. Often system requirements do not become apparent

[11] http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf, fetched on 23rd Dec 2010

until the implementation phase. Sometimes customers revise requirements after early prototyping.

Flexibility can be patched into the process with the use of feedback paths from stage *n* to stage *n-1*, or by multiple iterations through the entire process.

Modern thinking on software development lifecycle

A more recent approach is the *Unified Software Development Process*. This overcomes the brittle nature of the Waterfall model, since it is both *iterative* (i.e. feedback and refinement within and between stages) and *incremental* (taking small steps that build on work of previous iterations within and between stages).

The Unified Process divides into four phases:
1. Inception
2. Elaboration
3. Construction
4. Transition

The *inception* is the starting point of the software project. It should be the shortest phase in the project. It generates a business case for the project, with initial estimates of cost, benefits and timescale.

In the *elaboration* phase, the team captures the majority of system requirements. Generated documentation includes use cases, high-level data structures and module specifications. This documentation may be presented in a standard format such as the *Unified Modelling Language* (UML).

Actual software implementation takes place in the *construction* phase. This is generally the longest phase of the project. System features are implemented in short development iterations. At the end of this phase, an initial operational system is ready for prototyping.

In the *transition* phase, the working system is delivered to end-users. These may provide feedback for another cycle of the development process.

A comparison between Waterfall and Unified Process may prove helpful. We can identify phases of each development model that correspond with the other. We can consider potential advantages that the Unified Process has over Waterfall, given the benefit of thirty years of hindsight[12].

[12] http://xkcd.com/844/ sums it all up really, fetched on 7th Jan 2011

Requirements Gathering

A high proportion of software failure[13] stems from building the *wrong* system, rather than building the system wrongly. This failure begins when the software engineers and their clients fail to co-operate in capturing the system requirements at the outset of the project.

Functional requirements specify the services that the system is expected to provide. *Non-functional* requirements specify the constraints under which the system is expected to operate (e.g. response time, reliability). As a general guide, Sommerville[14] states that, 'System requirements should set out *what* the system should do rather than *how* this is done.'

A software project might involve the computerization of a business process, or the upgrading of an existing system. All stakeholders should be identified early and involved in eliciting the requirements. These might include users and managers for the client company, and software developers and architects for the software company.

There are many different methods for gathering requirements. These include:

1. Field study - observation of existing practice
2. Interviews and questionnaires
3. Walkthrough scenarios

Again, there are various formats for expressing requirements. The most popular approach involves *use cases*. These fit well with modular and incremental development. We consider use cases fully later.

There are several desirable properties for a set of software project requirements.

1. Correct (what the user wants)
2. Complete (cover whole of system operation)
3. Unambiguous (clear, and means the same to bother client and developer).
4. Testable (enable later verification that the implemented system meets the requirement).

Requirements must be clearly documented and archived. They form the basis of the *contract* between the client and the software development company. The software engineers will need to refer to the requirements throughout the software development process. The client will need to refer to the requirements when prototyping the system, and deciding whether it is satisfactory for final roll-out.

---

[13] http://www.projectsmart.co.uk/docs/chaos-report.pdf, fetched on 25th Jan 2012

[14] http://www.cs.st-andrews.ac.uk/~ifs/Books/SE8/Syllabuses/INTRO-SLIDES/Requirements-2.ppt, fetched on 24th Dec 2010. Note that there are a full set of lecture slides on Sommerville's website, with material to support his textbook.

Use Cases

Use cases encapsulate specific scenarios in which a user interacts with the software system in a specific way. For a bare minimum, an individual use case requires:

1. Title (verb-noun)
2. Identifier (numeric?)
3. Goal (what is accomplished by successful completion)
4. Actor (who engages with system)
5. Preconditions (system state immediately before scenario)
6. Trigger (event that causes the scenario to begin)
7. Course of events (basic, normal flow through the scenario)
8. Postconditions (system state immediately after scenario)

In today's lecture, we will consider use cases from a supermarket self-service checkout. These might include: SCAN-ITEM, BUY-LOOSEFRUIT, PAY-BILL, AUTHORIZE-ITEM.

```
Use Case 007: PAY-BILL

Goal: correctly provide payment for items in shopping
      basket
Actor: supermarket customer
Preconditions: all items have been scanned or weighed,
               and placed in bagging area
Trigger: user clicks "Finish and Pay"
Course of events:
   1. User selects payment type (cash/card/voucher)
   2. System displays total bill
   3. User pays in appropriate way
   4. System indicates payment accepted
   5. System returns change and prints receipt
   6. System reminds user to collect shopping
Postcondition: shopping in bagging area, waiting for
               user to remove them before system can
               return to start screen.
```

Object-Oriented Software Development

Think about a real-world system or process. Every concrete part of it can be modeled as an object (cf *noun*), with associated attributes (cf *adjective*) and capabilities (cf *verb*). Objects have relations (cf *case*) with other objects. An example of a university lecture is presented below. The object-oriented approach appeals to our intuition. It is useful to organize software systems around clearly defined data structures that have real-world counterparts, maintaining controlled access to owned data via particular operations invoked on the owner.

For an example, consider this lecture. As a timetabled university event, it has attributes such as:

1. Starting time
2. Finishing time
3. Location
4. Subject
5. Lecturer
6. Class of students

Operations on the lecture may include:

1. Attending the lecture
2. Downloading a copy of the notes
3. Rescheduling the lecture time

Furthermore, this individual lecture instance is related to other lectures in the same series, or degree programme, etc.

All lectures will follow a similar template, i.e. they all have common types of attributes and operations. This template is referred to as the lecture *class*. However each individual lecture will have different data for each attribute – since no two lectures can be in the same place at the same time. Each individual lecture is referred to as a lecture *object*, which is an instantiation of the lecture *class*.

Unified Modeling Language

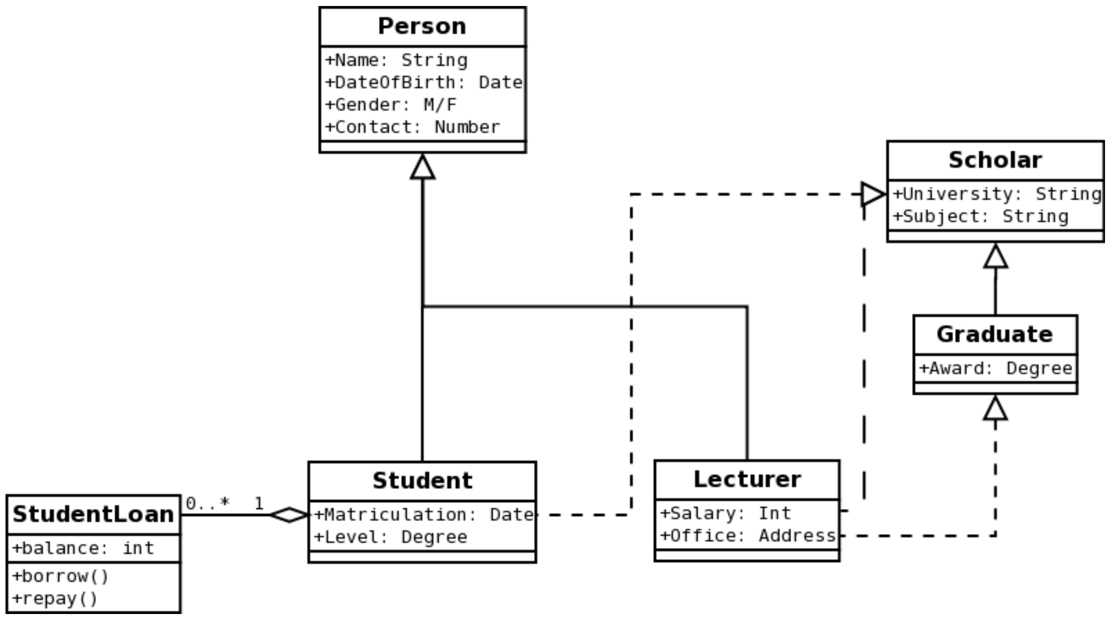
The unified modeling language (UML) is an industry-standard notation for object-oriented software development. It provides a graphical framework for describing objects and their interactions, in the context of a software system. UML is the standard language for modeling in the *unified software development process*, which was explained in a previous lecture.

There are more than 10 different kinds of UML diagram[15]. These can be divided into *structural* diagrams, which show how the software is constructed from various components, and *behavioural* diagrams, which show the dynamic interactions between the components as the software is executing.

[15] http://en.wikipedia.org/wiki/File:UML_Diagrams.jpg, fetched on 30th Dec 2010, gives a lovely montage of different UML diagrams

UML Class Diagrams

The simplest structural diagram is the *class diagram*. This describes the data fields and manipulation operations associated with each class (object template) in the system, and how the various classes relate to one another. Below is an example involving people in a university. For full details about the intricacies of UML diagrams, the UML cheat sheet[16] may be helpful.



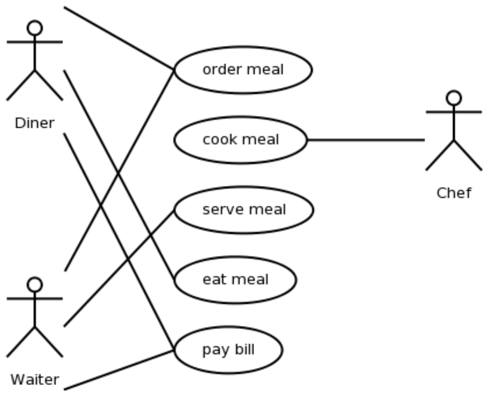
In the above diagram, each box denotes a single class. The top section gives the class name, the middle section gives the data fields, and the lower section gives the operations on that class. Lines between boxes represent object-oriented relationships. Lines ending in triangle arrow-heads indicate the "is-a" relation, which is known as *inheritance* in object-oriented programming. Lines ending in diamond heads indicate the "has-a" relation, which is known as *association.* For instance, a Student has a StudentLoan. The permissible frequency of loans per student is represented by the numbers on the line. Each loan is related to a single student. However each student may have zero or more loans.

16 http://loufranco.com/blog/assets/cheatsheet.pdf, fetched on 26th Jan 2012

UML Behavioural Diagrams

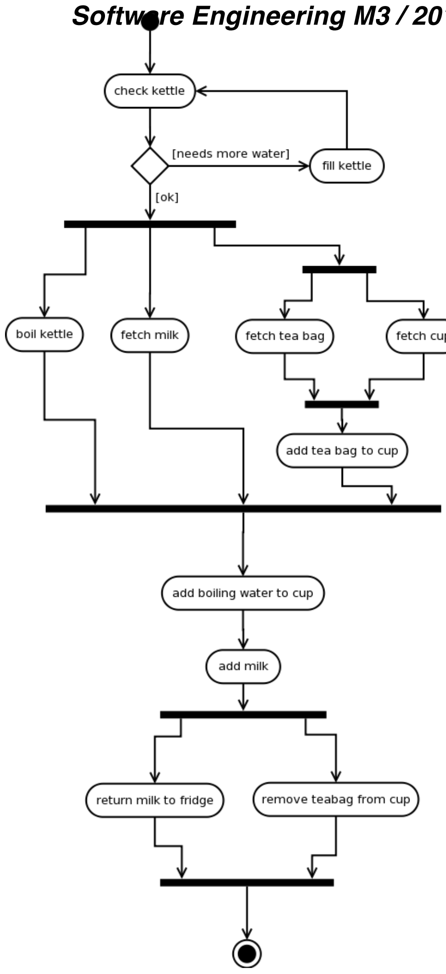
We have already considered structural diagrams, which describe the data structures and modules of the software system. Now we move onto behavioural diagrams, which describe the dynamic interaction between software components, as the software system executes.

The UML *use case diagram* is used during the early stages of development, to identify and encapsulate required system functionality. The diagram distinguishes between actors (people, represented as match-stick people) and use cases (goals to accomplish, represented as labeled ovals). Actors can trigger use cases by particular actions. The example below gives a use case diagram for a restaurant.

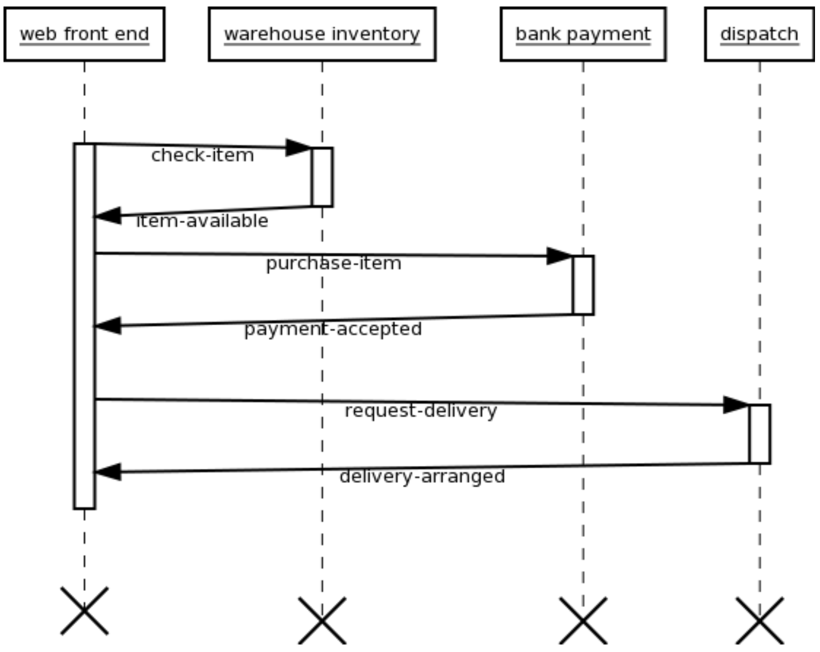


The UML *activity diagram* shows what has to happen to accomplish a task (typically an individual use case) and generally is implemented as a single procedure or method at the coding stage. The example below gives an activity diagram for making a cup of tea.

Each individual action is enclosed in a box. An edge between actions indicates control flow. A diamond is a conditional test, with labeled outcomes. Multiple independent activities can proceed in parallel with a *fork* horizontal bar. The end of the independent parallel activities is denoted by a *join* parallel bar.



The UML *sequence diagram* describes how several objects co-operate with each other to accomplish a task (typically a single use case). It shows the interacting objects and the sequence of messages that passes between them. The diagram below shows a web-based ordering system.

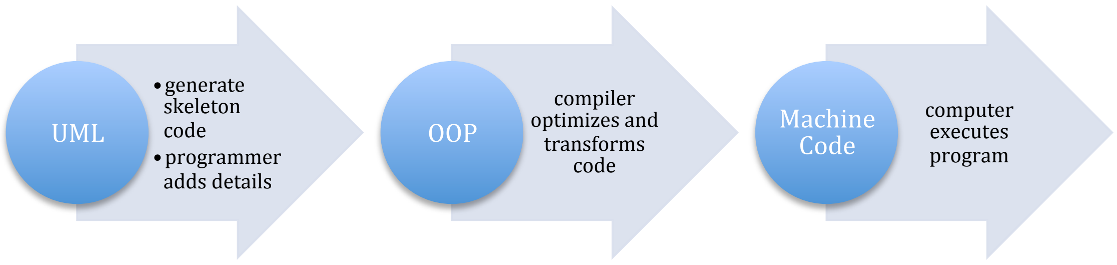


Object-Oriented Programming

Object-oriented programming languages have been around since the 1970s. Object-oriented programming (OOP) is currently the dominant programming paradigm. Major OOP languages include C++, C# and Java.

Although UML / object-oriented software engineering is more recent than OOP, it has been retro-fitted into the process. So all structural and behavioural concepts in UML should have corresponding implementations in OOP. Some UML tools have the ability to auto-generate OOP *skeleton code*, allowing the software developers to fill in the implementation-specific low-level details.

A program called a *compiler* translates the program into *executable machine code* for the system to run.


UML
• generate skeleton code
• programmer adds details
OOP
compiler optimizes and transforms code
Machine Code
computer executes program

Principles of OOP

Here are the main principles of OOP, which are demonstrated below in Java / C# syntax:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Abstraction is the ability to create complex, compound data types (classes), which group together related properties or behaviours.

```
class Rectangle {
  double width;
  double height;
}
```

Encapsulation is the ability to hide underlying implementation details from clients. If an object *a* of class A invokes a certain behaviour `area()` on an object of class B (calls the `area()` method), then *a* does not need to be aware of how `area()` is implemented.

```
class Rectangle {
    …
    double area() {
      return width*height;
    }
  }
```

Inheritance is the generalization/specialization relationship between classes that we considered in UML class diagrams. A subclass inherits the attributes and behaviour of its parent class. However it may add new properties, or modify existing ones.

```
class Square extends Rectangle {
  public Square(double sideLength) {
    super(sideLength, sideLength);
  }
}
```

Polymorphism is a powerful concept. Literally it means *many forms*. A class is polymorphic because it can be treated as if it belonged to any of its superclasses (i.e. although it is a specialized class, it can be used in the general case too).

```
class Person {
  void sayHello() {
    System.out.println("hello");
  }
}

class FrenchPerson {
  void sayHello() {
    System.out.println("bonjour");
  }
}
```

Design Patterns

Design *re-use* is a standard engineering principle. Re-using existing designs is efficient since it can result in:
(a) reduction in planning/design time, i.e. lower development costs.
(b) better understanding of system behaviour, i.e. reduced risk.

Software re-use is common practice. Sometimes program modules may be re-used directly. At other times, the high-level architectural design may be re-used, and customized for a particular implementation scenario. This field is known as software *architecture design patterns*.

Christopher Alexander introduced design patterns for the architecture of buildings, in the context of town planning, in the late 1970s. Alexander explains the motivation for design patterns as follows:

> '*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*'

Now design patterns are popular in many disciplines. They have been applied to software engineering since the 1980s. Software architecture design patterns are intended to capture common structure and behaviour of systems. Many libraries and catalogues of design patterns are available, in textbooks and online repositories[17].

An entry for a single pattern in a catalogue might have the following information:

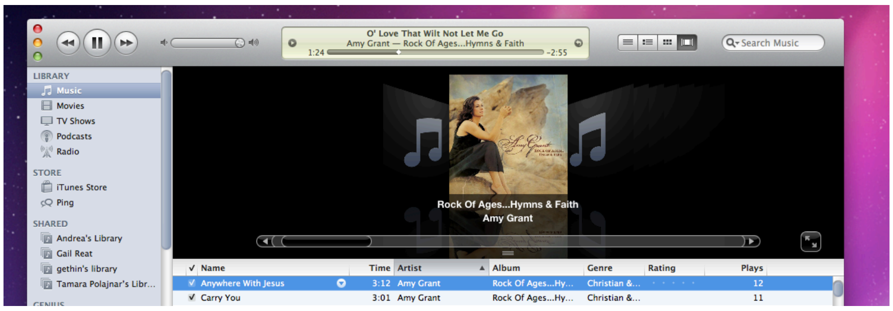
1. Pattern name
2. Intent – what is the goal of the pattern
3. Applicability – in which contexts might this pattern be used
4. Structure – software components to implement pattern, in UML.
5. Collaboration – interaction between components, in UML.
6. Implementation – textual description of how the pattern works.
7. Sample code – if appropriate, snippets of source code.
8. Deployed instances – actual examples of this pattern in use.

In this lecture, we consider a very common architectural pattern known as model-view-controller[18] (**name**). This pattern is relevant for graphical displays where a human user must interact with the system (**applicability**). The goal is for the user to easily understand the current state of the system, and be able to

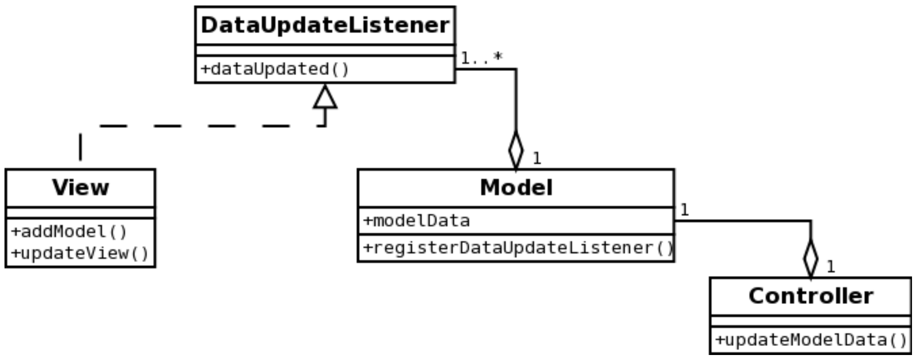
[17] See http://c2.com/cgi-bin/wiki?DesignPatterns (fetched 15 Feb 2011) for an overview. See also http://en.wikipedia.org/wiki/Architectural_pattern_(computer_science) (fetched 15 Feb 2011) for some example patterns.

[18] More details at http://c2.com/cgi-bin/wiki?ModelViewController and http://www.dkrypt.com/home/mvc (both fetched 15 Feb 2011).

quickly and easily modify this state (**intent**). The system involves three distinct components, the model, the view and the controller (**structure**). Intuitively, the computer understands the model and the user understands the view. These are both different representations of the same underlying system state. The user is able to change the system state (updating both the model and the view) by interacting via the controller (**implementation**). Generally, the controller will invoke the model to update, and the view will either query the model for updates or be refreshed automatically when the model changes. An example model-view-controller system is a digital music player (**deployed instance**). The model is the underlying data for an MP3 music file, together with your current playing position. The view is a pretty picture showing the album artwork together with a progress bar for the current track. The controller comprises all the buttons (back, pause, ...) that you can use to change the current music being played.



The key point about the model-view-controller architecture is that the model does not have to be aware of who is controlling it, or who is viewing it. This data abstraction, or separation of concerns, is ideally implemented in an object-oriented programming model. Consider the UML class diagram:



The model can register an abstract DataUpdateListener. Every time the model's data changes, it simply calls any DataUpdateListeners' updatedData methods. The view implements this Listener class, and provides suitable methods that re-render the view after a data update. The controller hooks into the model to update the data based on its inputs.

This design pattern is well understood and widely used. The components can be developed and tested separately (recall the modular development and unit testing stages of the software development process). Then the components can be integrated using this clearly defined architectural pattern.

History of Design Patterns

In the early years of software development, every problem was new – there were no 'standard' approaches to solving problems. As time progressed, software developers gained experience to re-use successful patterns. They also wanted to communicate their pattern-based designs to novice software engineers.

The appropriate descriptive technique for documenting and implementing these patterns relies on object-orientation. Object-oriented programming languages (like C++) became common in the 1980s. Furthermore, object-oriented software development models (like UML) reached maturity in the 1990s.

The so-called 'gang of four' software engineering researchers (Gamma et al) collected a catalogue of design patterns into a textbook resource[19]. They give an analogy to help understand their philosophy of design patterns:

> *Novelists and playwrights rarely design their plots from scratch. Instead, they follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels).*

A design pattern is essentially a solution to a problem in a particular context. In the previous lecture, we considered the Model/View/Controller pattern, which is a software *architectural* design pattern. In this lecture, we focus on *implementation* patterns, which might be appropriate for functionality of specific modules within an overall software system.

We have a running example application throughout this lecture, which is a text-adventure game based on navigating through a maze. We will employ three design patterns:

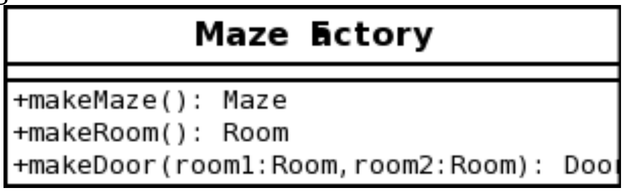
1. Abstract Factory: provides an interface for creating families of related or dependent objects without specifying their concrete classes.
2. Singleton: ensures a class has one instance, and provides a global point of access to it.
3. Command: encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Abstract Factory[20]

The maze will consist of **`Room`** objects, with **`Door`** objects connecting one **`Room`** to another. The **`Room`** and **`Door`** objects all 'belong' to the maze (in the sense of the UML 'has-a' relation). An Abstract Factory simplifies the creation of Maze

[19] *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides. Addison-Wesley. 1994.
[20] http://en.wikipedia.org/wiki/Abstract_factory_pattern, fetched on 17 Feb 2011.

instances and Maze elements like **`Room`** and **`Door`** objects. The class diagram for a MazeFactory is given below:



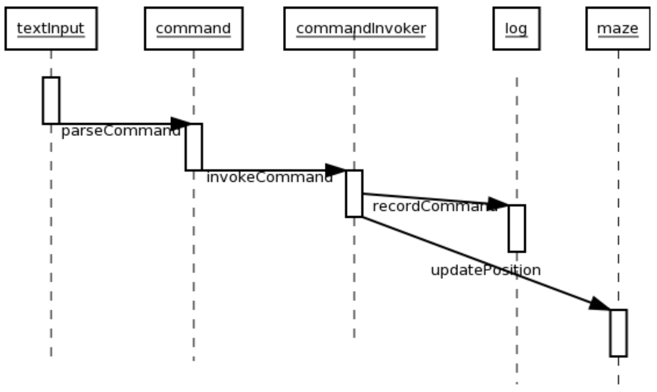
Now, the **`createMaze`** procedure can use the methods provided by the **`MazeFactory`** to instantiate a maze with rooms and doors as appropriate. The **`createMaze`** routine does not need to know the specific implementation details of the **`Room`** and **`Door`** objects. Indeed, there could be a specialized version of **`MazeFactory`** that created specialized **`Room`** and **`Door`** objects.

Singleton[21]

Recall the difference between class (cookie-cutter) and objects (cookies). A singleton class is a class that only ever has one instance – a cookie-cutter that only ever cuts a single cookie. This is applicable in our text-adventure game – we only ever want a single maze to be created in a game. We can use the Singleton pattern to accomplish this. It allows the system to instantiate a single Maze object, and causes an error if the system attempts to make more Mazes.

Command[22]

In the text-adventure game, it is possible to navigate through the maze by issuing commands like **`north`**, **`south`**, **`east`**, **`west`**. Other commands might be possible, for interacting with objects, etc. The text-interface allows the user to type commands and press enter. The command text will be parsed, and a Command object will be created. This command will be invoked, which involves executing it and storing its action in a history of commands. The execution may involve navigating to a different room, so the command will have to be modify the user's current position in the maze. A sequence diagram showing the messages passing between objects might look like:



[21] http://en.wikipedia.org/wiki/Singleton_pattern, fetched on 17 Feb 2011.
[22] http://en.wikipedia.org/wiki/Command_pattern, fetched on 17 Feb 2011.

Program Verification Techniques

According to Sommerville, 'Verification involves checking that the program conforms to its specification.' In this course we focus on post-implementation verification techniques, although ideally verification ought to take place at all stages of the software development process.

Once the program source code has been produced, but before it has been compiled into executable code, static program verification is applied. This can be either manual (performed by human experts) or automatic (performed by computer software).

Manual Program Inspection[23]

Once a developer has completed writing a unit of source code, he or she may submit it to others for review. This process of manual program inspection is a requirement in many companies (e.g. IBM) and open-source projects (e.g. Linux). The inspector's role is to find errors, omissions and inconsistencies in the program source code. These can then be queried with the original developer, and fixed if necessary. Sommerville cites a study that suggests 60% of program errors can be discovered by informal program inspections.

Automated Verification Techniques

Static analysis refers to software tools that automatically inspect program source code for errors. Whereas manual inspection is expensive, slow and occasionally unreliable, static analysis tools can be cheap, fast, and precise.

The simplest tools operate using:

- control-flow analysis: to detect unreachable sections of the program, and query these with the programmer.
- data-flow analysis: to detect unused variables, and query these with the programmer.

These anomalies indicate that the programmer has either incorporated unnecessary code into the program, or else misunderstood the conditions governing execution of certain parts of the program. Either way, the software tool is able to highlight the problem in such a way that the programmer can fix it.

An example program is shown below:

```
    int f(int x) {
```

[23] More information at http://en.wikipedia.org/wiki/Code_review, fetched on 1 Mar 2011.

```
          int y, z;
          z = x+1;
          return z;
        }

        int g(int x) {
          int y;
          return x+y;
        }
```

with the corresponding output from the analysis tool:

```
     $ gcc -O -c -Wunused -Wuninitialized undef.c
     undef.c: In function ‘f’:
     undef.c:3: warning: unused variable ‘y’
     undef.c: In function ‘g’:
     undef.c:12: warning: ‘y’ is used uninitialized in
this function
```

More complex static analysis techniques include mathematical reasoning about program behaviour. For instance, given a recursive factorial program:

```
     int factorial (int x) {
       if (x==0)
         return 1;
       else
         return x * factorial (x-1);
     }
```

A mathematical tool might be able to reason that, if x is a non-negative integer, then the program will eventually terminate. On the other hand, if x is negative, then the program will loop forever.[24]

As computer hardware has become ever more powerful, so automated software analysis tools have become more capable to detect errors in programs, with increasingly complex mathematical and algorithmic techniques. In 2002, one researcher reported that he could carry out a fairly sophisticated analysis of a million lines of code in less than one second.[25] Airbus use formal static analysis tools to verify all their flight control software.[26]

[24] Alan Turing showed that computer software cannot prove that programs will terminate, in the general case – see http://www.bbc.co.uk/dna/h2g2/A1304939 for details. However our example is sufficiently trivial.
[25] Nevin Heintze. 2002. Aliasing analysis for a million lines of C. In *Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*
[26] http://www.di.ens.fr/~cousot/COUSOTtalks/AIRBUS-2010-06-24.shtml, fetched on 1 Mar 2011.

Recent trends in Software Development

In this lecture, we consider two recent changes in the computing landscape. These developments have been made possible by the ubiquitous availability of the internet. Below is an overview of *Web Services* and *Software as a Service* (SaaS).[27]

Web Services

When it was originally created in the 1990s, the world wide web was intended to be a medium for humans to access information from computers. The web provides standard data transfer protocols for text-based information to be transmitted across internet links. However, more recently, the web is increasingly being used for computer software to communicate with other software. *Distributed systems* have software components running on different physical machines, which communicate with each other over the network. Web services are an example of such software components that may be composed to make a distributed system running over the internet.

Nowadays, web services underpin all kinds of business transactions, and user interaction with computers. A simple example of a web service is a weather app. Suppose a weather forecast provider supplies weather information via a web service. We can have an app installed on our machine, connected to the internet, that queries the weather info web service, then displays the appropriate weather symbol on our desktop screen.

Software as a Service

The recent hardware trend has been towards *thin clients*. These are relatively inexpensive computing devices with low-power processors and small memory capacities. (Think TV set-top boxes or smart phones.) The power of thin clients comes from their connectivity. They can fetch programs and data that the user requires from networked servers on-demand. This model is referred to as *utility computing*. In the same way that your gas or electric supply is metered, and you pay a monthly fee to maintain the connection, utility computing requires a

[27] Further details, as ever, can be found on wikipedia at http://en.wikipedia.org/wiki/Web_service and http://en.wikipedia.org/wiki/Software_as_a_service, both fetched on 9 Mar 2011.

subscription to support the ongoing service. The subscription may be a flat fee, or you may pay for data transferred, and compute cycles used. This utility-based approach has both benefits and drawbacks.

Benefits of utility computing

1. No software installation overhead
2. No need to maintain, upgrade or repair software
3. More efficient provisioning reduces energy consumption
4. Accessible from anywhere

Drawbacks of utility computing

1. No control over software customization
2. Ongoing expense of subscription
3. Security and reliability is not under your control

Many large software companies provide software-as-a-service. These include Amazon, Google and Microsoft. For individual, private users, the software may be provided for free. (Examples include Windows Live and Google Docs, which are funded by adverts, etc.) For businesses, who require more intense usage and guarantees about service reliability, there are per-user charges for the services. However for all but the largest companies, it may be more economical to adopt the utility computing model and outsource the software to a larger, expert company.

Open Source Software Development

An operating system (OS) is the underlying software that runs on a computer. It manages the various hardware resources available, and provides a set of standard services for applications. Example OSs are Windows and Mac OS.

Linux is a member of the Unix family of operating systems. Unix was originally devised by the Computer Science research community in the 1970s, and adopted by major US universities. At first, it was developed by enthusiasts, who had little or no monetary interests. However as Unix became more influential, it was rapidly turned into a commercial asset. This disenfranchised many people in the programming community, and sparked off the open-source revolution.

GNU Software

The key design philosophy of Unix is that everything should be done as simply as possible. Small, re-usable, modules can be chained together to effect complex behaviour. So when Richard Stallman decided to reimplement a free version of Unix in the 1980s, he could pick off individual modules one by one, and create his own versions[28]. Stallman named his project GNU (Gnu's Not Unix), which quickly gathered impetus in the software development community. Stallman was (and is) a genius programmer. He single-handedly worked on GNU software development for a decade or more. However his most important innovation was a legal one – the GNU Public Licence (GPL). Since Stallman had seen how the commercialization of Unix had stifled its development, he wanted to avoid the same fate for his GNU project. The GPL is the legal licence for his software[29]. The source code of GPL software must be freely available. Anyone can copy it and modify it, however the derivative software must remain under the GPL. (Its detractors refer to GPL as a viral licence.)

By the early 1990's, the only major component missing from a free GNU version of UNIX was the kernel. This is the core of the operating system, which manages the underlying machine's physical resources and allocates them fairly to user applications. When Linux arrived, it was the ideal missing piece to slot into the GNU system.

The Linux OS

Linus Torvalds was a Computer Science undergraduate student at Helsinki University in the early 1990s. He began developing an operating system kernel as a hobby, to help him understand how such software works, and to improve his working knowledge of the new Intel 386 architecture. Originally, Torvalds based his design on the Minix kernel, which was an academic prototype operating system.

[28] A full list of the GNU software modules is available at http://www.gnu.org/software/software.html, checked on 16 Mar 2011.
[29] Full legal document at http://www.gnu.org/licenses/gpl.html, fetched on 16 Mar 2011.

In a few months, Linux was working as a basic OS kernel. Although it was lacking some key features, such as internet communication, it was sufficient to be a suitable GNU kernel. Around the time that Torvalds was developing Linux, the internet was becoming an increasingly popular distribution medium for new software. Torvalds stored source code versions of his experimental Linux code on a Helsinki ftp server, and encouraged people to download his code and try it out. The response was overwhelming. Many other enthusiastic amateur developers began to use, and contribute to, the nascent Linux project.

Now, 20 years later, Linux is a major force in computer systems[30]. Most internet servers run Linux. Many mobile phones use a variant of Linux (Android). Although it has not yet achieved widespread popularity on the desktop, it has a growing market share (currently around 2%).

Reasons for Open Source Success

Eric Raymond has conducted an analysis of the popularity of Linux, and the reasons for its success[31]. He gives the following major points:

1. **Release early, release often.** This is the way to build up a community of engaged users.
2. **To solve an interesting problem, start by finding a problem that is interesting to you.** This is the *motivation* for developing without financial remuneration.
3. **Provided the development coordinator has a communications medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.** This is the opposite of Brooks' received wisdom on software development, but Linux is a shining example of how parallel, distributed, global, open source software development can work with outstanding results.

This philosophy of software development is entirely different to the one practised by major software companies over the last 50 years. However, it has a growing influence. When companies recognised the success of Linux and similar open source projects, they began to harness to power of the open source development model. For example, Sun (now Oracle) have released the Java system under an open source licence. Also Apple have the core components of their Mac OS X system as open source.

[30] Market share data available from http://en.wikipedia.org/wiki/Usage_share_of_operating_systems, checked on 16 Mar 2011.

[31] http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/, this is an open-source book, fetched on 16 Mar 2011.