Approaches to software development

Contents

Introduction

This free course, *Approaches to software development*, presents an engineering approach to the development of software systems – a software engineering approach, paying particular attention to issues of software quality, in terms of both product (what is built) and process (how we build it).

The material is organised in a distinctive way. We adopt an object-oriented approach to software development and assume you are familiar with the basic concepts of objects. This approach uses a fairly standard set of development techniques. We take a broad view, and techniques are discussed fairly independently of exactly where and when they would be used.

There has always been debate in the computing industry about just how useful any particular development technique is. We believe that the techniques introduced over the last 30 years or so, and many others currently being researched, are important because they address fundamental issues concerning software quality. You are probably already familiar with some of the techniques used in this course. Studying these techniques is more about how to use them to address the quality issues of what you are developing and how you are developing it, rather than about how to use them in practice.

In this course you will see a precise way of applying techniques, but we will also discuss, in parallel, a more light-weight approach to software development. You will become familiar with the purpose of utilising these techniques and will also develop an understanding of when their systematic use may or may not be appropriate. With experience you will be able to make decisions on which is the right combination of techniques for a particular project.

Software systems are built to meet requirements. (This leads to the developer's mantra, 'Software must be delivered on time, to budget and to specification.' You may know it in another form.) A successful software project must:

- resolve the diverse and possibly conflicting needs of users in a disciplined way
- satisfy the users' expectations
- have been developed and delivered in a timely and economical manner
- be resilient to the changes that will be introduced during its operational lifetime
- demonstrate good overall system quality.

This is a daunting prospect for those developing and maintaining software. A central aim of this course is to give you the intellectual tools to cope with the challenge.

This course provides an introduction to software engineering. Assuming you already have some experience of software development, some of this material will be familiar to you, though your existing knowledge will be consolidated and will begin to be extended to more advanced areas. We discuss some of the ideas that underpin software development in Section 1, and consider the basic activities of software development in Section 2. Section 3 looks at the role of models and modelling languages, introduces a well-known software development process.

This OpenLearn course is an adapted extract from the Open University course TM354 *Software engineering*.

Learning Outcomes

After studying this course you should be able to:

- describe the essential characteristics, and identify, using examples, the connections between the characteristics of a good software system
- describe the elements of a basic software development process and illustrate the variety of different life cycles
- understand the motivation for, and best practices of, an agile approach to software development
- explain the benefits of the Unified Modeling Language (UML) as a standard notation for modelling
- identify the different kinds of model used in the development of software and describe the relationship between models, viewpoints and software development.

1 Software and software engineering

This section describes the basic characteristics of a ‘good software system’, and considers how such software systems may be developed. Once built, software rarely remains static and can change on a regular basis, so maintaining software is a key activity in software engineering.

1.1 What is a system?

The word system is in regular everyday use. We talk about the social-security system, the telephone system, computing systems and even ‘The System’. The trouble is that our everyday language is often imprecise, and people use the same word to mean different things (that is one reason why software development can be an arduous process).

The word system is derived from the Greek word meaning ‘to set up’.

Definition

We define a **system** as ‘an assembly of components that are connected together in an organised way’. The term ‘organised’ is important here. For example, it tells us that components are affected by being part of a system.

Example 1

A pile of old telephones, exchange-switching gear and wiring that are awaiting disposal may once have been a system. But heaped together, they are *not* a system. Both the organisation and the connections have been lost.

A system, therefore, is greater than the sum of its parts, and it has properties that cannot be deduced or predicted by examining any one of its components in isolation (or even *all* of its components, if they are not organised and connected). From the above example, could you deduce the properties of the telephone system by examining a single, disconnected telephone (or even a big pile of disconnected telephones)? No, because it no longer works as expected, even though it has not changed physically (it is not broken).

The assembly of components that forms a system does something, it carries out some process. This **process** will work on inputs, carry out transformations and produce outputs in order to achieve a goal. If you view your body as a system, it consists of organs, it takes oxygen and food as inputs, and does transformations that keep you alive. As a system, it has a well-defined boundary and a control mechanism that will keep it adapting to changes in the environment.

Software systems

This course is specifically about software systems, systems where software plays a major role. However, software does not do anything without the hardware where it is installed and running, and software systems are usually part of a much wider context that involves not only other technical components, but also people, organisations and other social structures. The ‘whole’ of all these components is also known as a **sociotechnical**

system. Sociotechnical systems are complex systems that need to be understood as a whole rather than as a sum of their isolated components.

In focusing on the software we will therefore be ignoring many other components of the whole system in which software is just a part. However, as a software engineer you will need to be aware of how software is used and the interactions with hardware, people and organisations. The popular press is full of reports of software failures, many of which may relate directly to problems with the software but may also relate to other issues of the wider sociotechnical systems context, as in Example 2.

Example 2

You may find it interesting to read about other examples of software failures, such as the Therac 25 software-controlled radiation therapy machine, which was responsible for the death of several patients in the late 1980s (Leveson and Turner, 1993).

One well-studied case is that of the unmanned European Ariane 5 rocket, which failed on its launching flight due to a software problem. During the launch one of its computers stopped working due to a variable exceeding a limit – a data conversion resulted in an overflow software exception.

There is some controversy about where the exact blame should fall for the problems that occurred, but there is an agreement that they were related to the reuse of a module from a previous rocket without proper enforcement of constraints. This case became well known due to the huge expense of the rocket ($500 million), the amount of time it took to develop (10 years) and what it meant for European space projects (Lions, 1996).

Viewpoints

A system can be, and often is, a personal ordering of reality, the result of seeing some degree of orderly interconnectedness in some part of the world. So a system can be many different kinds of system simultaneously, depending on who is studying it and why. Different **viewpoints** of a system correspond to different sets of users and therefore different purposes. In this sense, it is a subjective ordering of reality.

Example 3

A telephone system is a communications system to its users. For the engineers who set it up and maintain it, it is a technical system (that is linked to an employment system – a job – in their view). Similarly someone who designs telephone switches considers each switch as a system in its own right – a switch can record usage data as well as route your calls.

As Example 3 shows, there is a notion of what is included within a system, and what is excluded. Naturally this notion depends on the stakeholders involved in the modelling and development of the system and the viewpoints that they have over it. Identifying the scope of a system is an essential step in the development process.

System boundaries

The **system boundary** is a conceptual line that divides the system that you want to study from ‘everything else’. It is useful to think of a system’s environment as being made up of those things that are not part of the system, but can either affect the system or be affected by it. Example 4 takes you into a particular area of interest, which is known as a **domain**, to look at system boundaries and how they can change.

Example 4

A hospital is a domain where software is put to a variety of uses. A hospital might, for example, join together a series of patient-monitoring systems with the database management system that manages medical records, creating a larger system with a different scope. A forward-looking hospital might wish to go further and add weather-forecasting software. This extension would allow planners to deal with the variations in flow of patients that arise according to the season. Beds may be allocated and other resources, such as drugs, bought in preparation.

Activity 1 Combining systems

Allow approximately 10 minutes.

In Example 4, we suggested that two systems, for patient monitoring and managing medical records, might be combined into a single system. Suggest an additional function that might be possible with the combined system that would not have been possible with either of the two original systems alone. What can you say about the boundary of the combined system compared with the boundaries of the original separate systems?

Answer

Suppose the monitoring system detected that a patient taking a common drug had a heart problem. If the two systems were combined, it would be possible to automatically check whether the heart problem might be due to a known allergy recorded in the patient's record.

The boundary of the combined system encompasses a wider scope than the combined boundaries of the separate systems because the combination of patient monitoring and medical records supports a wider range of verifications.

1.2 The nature of software

Software is often spoken of as being invisible or intangible, and hence is thought of as being different from physical artefacts, which can be measured, touched, broken, and so on. This invisibility can lead to unrealistic expectations concerning the capabilities of software, which in turn may contribute to some of the myths that surround software and its development, for example, that accommodating change is straightforward.

Software can, and does, contain errors. There are three important characteristics of software that affect its development and the likelihood of errors:

- *Malleability*. Software is easy to change (programmers are often tempted to tweak their code). This malleability creates a constant pressure for software to be changed rather than replaced. Every change introduces the possibility of new errors.
- *Complexity*. Software is often complex. Complexity can usually be recognised, but it is less easy to define. One item of software can be considered more complex than another if it requires more explanation. Part of that complexity arises from the

potential variety of pathways between the components of a system. The number of errors is likely to depend on the complexity of a system.

- *Size*. It is likely that there will be more errors in a large piece of software than in a small one.

You have already encountered an aspect of the intangible nature of software if you have programmed in, for example, Java. The instructions and statements that you write in Java are translated into bytecode, which you do not see.

You might think that, because it is 'invisible', software is inherently more difficult to develop than a physical artefact. However, various techniques can be used to model software and its behaviour – just as an industrial designer uses geometric abstractions and other tools to model a physical product before it is built.

Activity 2 Problems with software systems

Allow approximately 15 minutes.

For each of the three characteristics of software mentioned above, explain why errors might arise in a piece of developed software.

Answer

Malleability. As change is easy to make, often changes are introduced without thorough consideration of the full consequences of each new change introduced.

Complexity. The more complex a piece of software becomes, the more chances there are of a change affecting other parts of the software.

Size. The greater the number of lines of code in a piece of software, the greater the number of likely errors.

1.3 Characteristics of a software system

Having considered the basic terms 'system' and 'software', we can move on to the notion of a software system. There are two important questions that we want to address.

- What characteristics should we be looking for in a software system so that we can develop one that meets the needs of all its users?
- What attributes should a software system have in order to be of high quality?

We develop a software system in response to an identified need. For the purpose of this course, we assume a software system is to solve a customer's problem – this is not always the case as software can be developed without a specific customer, in the hope that someone will buy or use it in the future. The contract between customer and developer, as provider, includes the requirements specification, which identifies what the software should do and the environment in which it must work. We expect a software system to be *useful*, as otherwise it cannot meet the needs of its users.

To its users, the user interface represents the software system. No matter what functionality is contained in the software, it cannot be used to its full potential if that interface is difficult to use. We expect a software system to be *usable* – otherwise the user is hindered in their use of the software to carry out tasks.

We expect a software system to be *reliable*, in that errors are minimised, as otherwise its users will not be able to perform those tasks that need its support.

While developers are busy constructing a software system, it is probable that the users' needs will change. In addition new requirements are likely to be identified once a software system becomes operational. It is also possible that the developer may miss a requirement during the specification process. We expect a software system to be *flexible*, because it is important to be able to change it easily as time goes by. In addition, a flexible software system makes it easy to correct errors.

In order to operate in its target environment, any design solution to a requirements specification must be turned into machine-readable code. No matter how useful or usable a software system might be, we expect it to be *available* in the target environment, offering continually available services in the customer's environment.

The contract between the customer and the developer will also include delivery dates and costs. Whatever process is used, the developer is expected to meet such contractual constraints. We could also say that the developer's working practices affect the availability of a software system, including its initial delivery and any subsequent changes. From the customer's point of view, the software system must be *affordable* to buy and maintain.

From the developer's point of view, there must be a way of keeping control of a project. Labour resources are the most significant component of a developer's contract, and often become the two-edged sword that leads to the success or failure of a software development project.

Software development is concerned with the activities that lead to a useful software system – techniques that will help you to define a requirements specification and then produce a design solution to the problem contained in the specification.

Activity 3 Good software systems

Allow approximately 20 minutes.

(a) What is the defining quality of a good software system, and what are its main characteristics?

(b) How might greater flexibility make a software system more affordable over its whole life?

(c) Give two reasons why a delivered software system might not meet its users' needs.

Answer

(a) A good software system is one that meets its users' needs. We can characterise a good software system as useful, usable, reliable, flexible, available and affordable.

(b) Users' needs will change over time. The time taken to implement the changes in requirements in a flexible system is less than for less flexible software. As labour costs are the most significant component of software costs, flexible software is more affordable.

(c) Software systems are usually out of date even as they are being developed because:

- some needs are often missed during requirements capture
- users' needs change with time.

1.4 Maintainability and other software problems

A software system should be both available, so that users can decide whether or not it still meets their needs, and flexible, so that the developer can change it to meet its users' needs. Maintenance is the key activity for the coordination and control of changes to a software system. In order to be maintainable, a software system should be written and documented in such a way that changes can readily be made. This means that we must take into account the process used to develop a software system. What the developer does during development affects the ease with which it can be maintained. If a change is easy to make, the cost of that change (labour) can be minimised so that the software system continues to be affordable. The **maintainability** of software is greatly influenced by how software is designed, written and documented.

Problems of maintenance also apply to software that is purchased as an off-the-shelf package, or software that is offered as a service by another provider.

Example 5

For anything you buy and use, there are costs associated with its purchase and with the maintenance required for its continued operation. For example, there are costs associated with buying a car and with its continued operation. As the vehicle ages, the costs of maintenance rise, and there comes a point when you have to decide whether it would be preferable to buy a new car or continue with the existing one. Perhaps surprisingly, there are similarities between the maintenance of vehicles and the maintenance of software.

Legacy systems

A significant problem relates to software systems that have been in operation for some time. If a particular software system continues to meet its users' needs, there may be little or no motivation to replace it, especially if that software is associated with a critical service within a company. Such systems are called **legacy systems** and typically have the following characteristics:

- old
- large
- developed using outdated techniques
- implemented with old programming languages
- critical to the business
- changed a number of times since their inception
- difficult to understand because of either a lack of documentation about their internal structure or a lack of experience within the group responsible for them
- difficult to maintain because of the above factors.

One option would be to replace an ageing software system with a new one. But the change from old to new can have serious implications for the company involved. It is not just the users who need to be retrained to use the new software. In some companies the whole internal organisation is based around their major software systems. Changing these may require a costly company-wide reorganisation to reflect the new software systems more accurately. There is also the issue of maintaining continuity of service during the changeover and the risk that the new and therefore unknown system may not work – in contrast, the legacy system is a known quantity.

There is an additional twist that makes the problem of legacy systems worse. Staff turnover may mean that there is no one left in the company that developed the software who understands the legacy system enough to continue maintaining it. Even if those who developed the original software system have not left, they may be working on other projects or be unwilling to look after an old system. When faced with the choice of working on a new or an old system, many people prefer to produce something new.

There is every chance that tomorrow's legacy systems are being built today. We might say that today's solutions are tomorrow's problems.

Unsuccessful software systems

Unfortunately, successful software systems rarely make the news. Example 6 illustrates the kind of failure that does break through into the public domain (in this case, the failure became a series of headline stories).

Example 6

In 1992, the London ambulance service commissioned a computer-aided dispatch (CAD) system for getting ambulances and their crews to reported incidents. It was intended to replace an existing paper-based system. There were two notable failures of the new system in November 1992 that resulted in severe delays in ambulances reaching certain incidents.

The final report on the system failures (South West Thames Regional Health Authority, 1993) identified a number of major management problems with the project. A technical audit revealed that the CAD system was incomplete and not fully tested. Its ability to deal with heavy loads had not been tested and nor had a backup service been tested. There were also outstanding problems with the accuracy of the information needed to initiate each dispatch (South West Thames Regional Health Authority, 1993).

The London ambulance service had a software system that was lacking in usefulness, usability, reliability and availability.

This discussion of software problems has probably given you a rather pessimistic view of software development. You must not forget that many software systems work, and work well, given the fact that they are used in every aspect of modern life. However, the rate at which the role of software is evolving is so fast that there must be practices involved in developing software that can cope effectively with this expansion.

Activity 4 Maintainability

Allow approximately 20 minutes.

(a) Suggest a means of measuring the maintainability of a software system.

(b) What can we learn from legacy systems about developing a good software system?

(c) Suggest a reason why legacy systems will always be a problem.

Answer

(a) We could measure the effort required by a developer to locate and implement a given change to a software system. That effort can be classified in two components – the effort needed to locate and fix errors (bugs), and the effort needed to adapt the software system to meet its users' needs.

(b) A legacy system may have started out with all the characteristics of a good software system, yet those characteristics may have changed over time, resulting in a less flexible and maintainable product. As change is inevitable, the right processes should be in place to make change happen in a more controlled way. This requires the adoption of standards and documentation conventions that help decision making on changes and how to introduce them. Any changes need to be well documented so that software is still understandable and less dependent on the people initially involved with developing and maintaining the software.

(c) The inherent malleability of software makes it easy to change. You have already seen that a legacy system is lacking in flexibility as a result of the number of changes made to it during its operational lifetime. (The analogy with metalworking through malleability is useful. Once a blacksmith forms some component, usually in iron, there is a limit to the number of times that it can be heated, formed and cooled before that component becomes brittle and hence liable to failure.)This explains why our ability to bolt features and fixes onto a legacy system means that it will eventually become too fragile, and it will become precarious to go any further. The staff issues mentioned in (b) compound these problems.

1.5 Divide and conquer?

As computing technology has improved, we have tried to construct software systems that can deal with larger and more complex problems. In order to provide such solutions, the software systems themselves have become larger and more complex. Unfortunately there is a limit to how much we can take in and understand at any one time. The Romans had a strategy called *divide et impera* – divide and rule. However this covered the idea that it was easier to rule over groups in conflict with each other.

How can we cope with tricky problems or situations where there is just too much information? The main technique for dealing with such messy situations is **decomposition**. We can decompose a problem into smaller and smaller parts or chunks until each one can be comprehended or dealt with by an individual. In terms of our earlier definition of a system, we are looking for patterns and/or components within a system and creating internal boundaries around them to identify smaller subsystems, sometimes referred to as **modules**. We will expose some form of hierarchy in our attempt to simplify a complex system.

The concept of decomposition can also be applied to the way you develop a software system. As you will see in Section 2, you can identify a number of different activities or tasks that an individual or group of individuals might perform in a software development project.

An individual can successfully build small software systems because it is possible for that person to understand all that is needed about the problem and its solution. There is a long history of individuals attempting to develop more complex systems – so-called **heroic**

programming – where success has been less certain. Sometimes the circumstances resulting from poor planning lead to a situation where it is the only way out to achieve a piece of workable software. However there are systems that are so large and complex, such as those that monitor and control air traffic, that they cannot be built by a single individual. The development of such a software system requires a team of people whose work must be well coordinated and managed. There must be a well-defined process if they are to produce an appropriate solution – a software system that is useful, usable, reliable, flexible, available and affordable. In practice, the process of software development is partitioned to enable individuals to specialise in different development activities such as analysis, design and implementation.

Problem and solution

By dividing one large problem into a set of smaller sub-problems, we might expect to reach a point where we can capture, understand and describe each sub-problem. But there are two difficulties.

- How do we know that each identified sub-problem is any easier to understand and solve than the original problem?
- How do we know that all of our sub-problems will fit back together again and recreate that original problem?

The key to finding sub-problems that you recognise and understand is prior knowledge and experience. As a developer, you may know how to create a good design or a good program, but you may not be familiar with the **problem domain**.

As a developer, the problem that you face is in the world outside the computing system, where you aim to provide a solution. The software that you construct can provide solutions to problems because they are connected to the world outside, as illustrated in Figure 1.

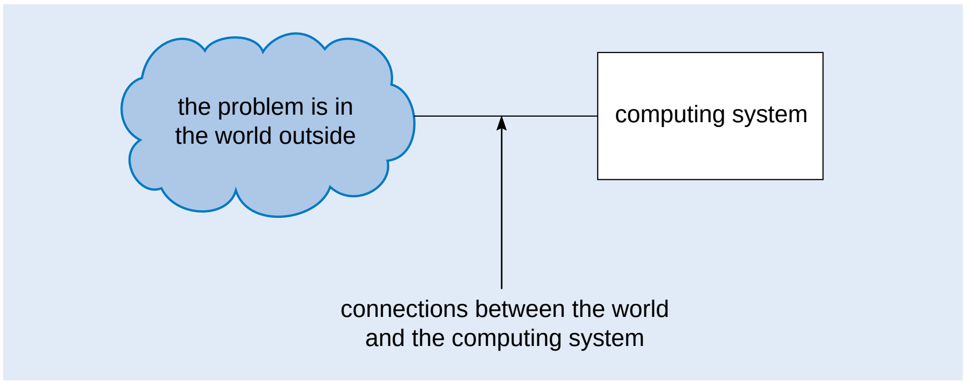

the problem is in the world outside
computing system
connections between the world and the computing system

Figure 1 Separating a problem from its solution

The main issue that you as a developer should focus on is solving the problems identified by the users, rather than worrying about how to write the program code. When projects fail it may be because of mistakes that were made quite early on in the design stage, often resulting from poor understanding of the problem domain, not because of what happens at the coding stage.

Example 7

One way to find out whether you have a good solution to a problem is to consider the kinds of complaint you would receive from users if your software did not work properly. For example, suppose someone had made a reservation to stay in a hotel but when they arrived, the receptionist told them that there was no reservation in their name. In another case, they might have reserved rooms for six people only to find that there was only room for one.

By doing such thought experiments you may often be able to spot flaws in your solution, although of course there may always be situations you have overlooked, so you can never be completely certain that your solution is comprehensive.

Modules and interfaces

In software development, there is a long history of decomposing a system into smaller modules. This **modularisation** is the standard technique for dealing with large and complex systems. The modules partition the system design or code. Some typical examples of modules are:

- whole programs or applications
- software libraries
- classes, in an object-oriented language such as Java.

Although modules may appear as self-contained elements, because they are all parts of a larger whole there must always be relationships between them that need to be taken into account, as Example 8 shows.

These relationships limit the ability to change one module without affecting other modules.

Example 8

In developing a software system for a typical manufacturing business, we can identify a number of possible subsystems – areas for development, such as systems to deal with customers, production, accounts (payments) and deliveries. While we might want to treat these areas as independent (as partitions), there are connections between them. For example, customers are expected to pay for the goods that they have ordered from the business, and the deliveries department is expected to deliver to the customer. Similarly in the production subsystem, the progress of each customer's order would be tracked through the factory and on to the delivery of the completed goods to the customer.

In general, we say that the **interface** to a module defines how other modules can use that module. Sometimes, you can define more than one interface for a module to allow yourself to be more precise about which services will be offered to different kinds of client.

An interface is the means of connecting one module to another. It tells you what to expect about the behaviour of a given module and what services it will provide, without telling you how those services will be provided. For example, the interface can define how a bank accounts module will respond to queries about the balance of an account or the types of accounts available.

The interface of a module is a description of all the externally visible operations and what other modules need to know and do to make use of them, but without any details of how the operations are implemented internally. From an object-oriented point of view, we say that the interface to the bank accounts module is an **encapsulation** of what we know about accounts in a bank.

A module that provides services to other modules may in turn need to use the services of yet other modules. These required services are called its **context dependencies**. A module's context dependencies and its interface, including any requirements that prospective **clients** need to meet, form a sort of **contract** with clients. The contract describes what the module does and what needs to be true for the module to do it. Clients can assume that if the necessary conditions are met, the module will fulfil its specified responsibilities.

The concept of an interface helps the developer to be more productive. If you can rely on the specification contained in another module's interface, there is less for you to understand because you do not need to know how it works. Furthermore, you have a better chance of understanding your part of the software system because you can focus on the things you need to perform your task. A side effect of this added understanding is that you are less likely to introduce errors. Furthermore, if a module is hidden behind an interface it is potentially replaceable.

Once your software system contains a set of modules that are well understood, each with its own interface and context dependencies, you can consider whether any of them can be reused. It may be that a popular set of modules is adopted as a standard.

The ability for different computing systems to communicate over a network has developed because of the adoption of such a standard set of modules. Control over the complexity of computer communications has been gained by decomposing the problem into a number of layers. Each layer has a well-defined interface through which the layer above it accesses its services.

Abstraction

Abstraction is a particular way of viewing a complex problem to arrive at some useful decomposition of that problem. The idea is to group together similar objects or situations and, while ignoring their differences, focus on one particular and common aspect of the problem that all these objects/situations possess and/or reflect. The key task in developing a software system is to decide upon the most suitable abstractions in the problem domain.

We can say that we have achieved a useful abstraction in a particular module if the potential software clients of that module do not need to know more than is contained in its interface. For example, a dedicated module to deal with date handling is a useful abstraction. The fact that it may be complex to implement is immaterial to clients that use the services defined in its interface.

Coupling and cohesion

As Example 8 shows, developers need to deal with the dependencies that arise as a result of their decomposition of a problem and its solution into a number of modules. We say that a module of a system depends on another if it is possible that a change to one module requires a change to another. For example, if a business changes its production methods this may cause a consequent change in the way it calculates the payments required for the goods it produces.

A developer must not only deal with the nature of each dependency but also the number of dependencies. In software engineering, '**coupling**' is used to refer to the degree of interdependence among the different parts of a system. It is easy to see that certain systems can have chains of interdependent modules where, for example, module A depends on module B, which depends on module C, and so on. In some cases these chains may join up and create a **circular dependency**, which is a particular form of strong (or high) coupling.

Developers try to construct loosely coupled systems because they are easier to understand and maintain. So a good software system has low coupling, which means that changes to one part are less likely to propagate through the rest of the system. A further benefit of low coupling is that components are easy to replace and, potentially, reuse.

Whatever the level of coupling in a software system, it is important to know which modules are coupled. If there were no records of the coupling between modules, a developer would have to spend time working through the modules to determine whether or not each was affected by a change. The result would be a lot of effort spent on checking, even if no changes were needed. Example 9 illustrates the danger of having more than one module making use of common or shared data.

Example 9

Date handling has always been a problem for software developers. For applications of a certain age, the most applicable storage format for representing a year was a number between 0 and 99. It made sense because 1966 was stored as 66, 1989 as 89, and so on, therefore less space was needed to store just two digits. Furthermore, if dates were stored as numbers, tasks that involved sorting by date order were simple – 22 January 1989 stored as 890122, is after 22 December 1966 stored as 661222.

Unfortunately, a number of these applications were still in use as the year 2000 approached, so every module in every application that used the short form of year had to be investigated.

A major aspect of the problem in Example 9 was that different developers had different ways of reading and manipulating the values stored in variables that used the six-figure date format. This increased the effort required to resolve the so-called millennium bug. If developers had had a consistent way to manipulate dates that did not rely upon the storage format, the millennium bug would not have been an issue of concern.

Cohesion is a way of describing how closely the activities within a single module are related to each other. Cohesion is a general concept – for example, a department in an organisation might have a cohesive set of responsibilities (accounts, say), or not (miscellaneous services). In software systems, a highly cohesive module performs one task or achieves a single objective – 'do one thing and do it well' is a useful motto to apply. A module should implement a single logical task or a single logical entity.

Low coupling and high cohesion are competing goals. If every module does only one thing at a low level of abstraction, we might need a complex edifice of highly coupled modules to perform an activity at higher levels of abstraction. A developer should try to achieve the best balance between the levels of coupling and cohesion for a software system. For example, hotels generate income by letting out their rooms to guests. The concept of room is likely to be represented somewhere in the software system for reservations for a hotel. It may be convenient to use a module or class representing the concept of room to collect and store data about the income generated by letting rooms. However, a better solution is to have a separate bill or payment module, because it is more cohesive, especially when a hotel generates income in other ways, for example, from serving meals to people who are not resident guests.

Activity 5 Divide and conquer

Allow approximately 20 minutes.

(a) Why might you consider splitting up a large project into smaller chunks?

(b) How does the complexity of a software system affect the maintenance task?

(c) What is a module?

(d) Why does it help to have low coupling in a software system?

(e) Give examples of the kinds of information that would be valuable when considering a change to a given module.

(f) What are the context dependencies of a module? How do they relate to a module's interface?

(g) What are the benefits of using modules with defined interfaces?

(h) Why does it help to have high cohesion in the modules of a software system?

(i) What characteristics should a module display that will help to ensure that it is easy and cheap to develop and maintain, and that errors are kept to a minimum?

(j) Why is it important to achieve a balance between coupling and cohesion?

Answer

(a) There is a limit to how much one person can understand at any one time. So there is a limit to the size of a software system that any one person can deal with. By splitting a large project into smaller chunks, it is possible to identify a number of more manageable tasks for those involved.

(b) It is essential to be able to make a change to a software system without having to know all about that system. Each change becomes difficult when the flow of control and dependencies within programs are complex. The greater the number and nature of the dependencies, the harder it is to maintain a software system.

(c) A module is any identifiable part of a software system that is considered separately. For example, modules may be subroutines (in a procedural language equivalent to methods), classes (in an object-oriented language), library functions or other constructs that may be treated independently.

(d) With low coupling, there are few dependencies between modules. Therefore changes made to one part (one or more modules) of a software system are less likely to propagate throughout the whole system. (A clear record of the dependencies between modules helps you to predict the impact of a proposed change to a software system.)

(e) There are two kinds of information that contribute to the analysis of a proposed change:

- Which modules are clients of the module in question? This information indicates how far a change may propagate through the software system.
- What assumptions have been made in client modules of the module in question? An understanding of the expected services of a module will help assess the risks associated with a particular change.

(f) The context dependencies for a module are the services of other modules that the module needs in order to work correctly. You can express the context dependencies for a module in terms of other interfaces. In effect, you can express the responsibilities of a module in terms of its interface and context dependencies. If the context provides the services that the module needs and clients meet any conditions specified in the interface, the module can guarantee the provision of the services described in its interface.

(g) The benefits are as follows:

- Developers will need to know only about the module's interface (its syntax and what it requires and achieves – its semantics), not how it provides those services. Consequently developers can be more productive.
- Developers can understand aspects of the software system more thoroughly, so fewer bugs will be introduced.
- It should be easier to find bugs, as irrelevant modules are avoided.
- The possibility of module reuse is increased once it is known what that module provides and requires.

(h) With high cohesion, a module carries out a sensible set of operations or activities. Ideally high cohesion implies just one major abstraction per module. The interface abstracts away from what a developer must know in order to use a module. This makes it easier for developers to understand the purpose of the module and how to use it. In addition high cohesion tends to make a module more reusable in other applications, because it provides a set of operations that sit naturally together.

(i) A module should have low coupling and high cohesion, represent a good abstraction, and have a well-defined interface that is an encapsulated abstraction of a well-understood concept.

(j) In constructing a system, you may have a choice between a smaller set of loosely coupled, less cohesive modules, or a larger set of tightly coupled, more cohesive modules. In the former case each module may be difficult to understand, while in the latter case the relationships between them may be over-complex. You need to strike an appropriate balance.

1.6 Architecture

In software development the term **architecture** is associated with the overall structure of a software system or, at a higher level, a family of software systems. For this course, we will adopt the following definition, taken from Bass et al (2012).

> The **software architecture** of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

It is important to understand the software architecture because it tells the developer, among other interested parties, about the overall shape of the actual or proposed software system. It explains how the development team can use various technologies to construct or assemble a software system.

Software requirements come in all shapes and sizes, so developers need to consider what the most appropriate process to develop an architecture is for a given context. Choosing a software architecture, or at least part of it, is one of the earliest decisions a development team has to make. There is always something that can be called a software architecture, whatever software system is being developed. Even a basic software system has an architecture.

Architecture serves as a main guide for developers but it is also of importance to other stakeholders in software development, all of whom will have different concerns about the software. For example:

- users may be concerned with how easy it is to use and learn
- the customer is interested in how much it will cost and when it will be delivered
- software maintainers will be thinking about how easy it is to modify and understand the repercussions when a change is required to one of the modules.

The architecture will therefore embody many decisions that will affect how the concerns of the many stakeholders are dealt with. It will also serve as an important means of communication between the different groups of stakeholders. It is the role of the software architect to balance these different concerns and reach compromises among competing concerns.

Activity 6 Architecture

Allow approximately 10 minutes.

Suggest some similarities and differences between software architecture and building architecture.

Answer

They are both abstractions of the structure of a system. They represent decisions that will affect concerns of different stakeholders. They are used as a communication vehicle with different stakeholders.

Changing a building once complete is expensive, whereas changing details or internal decoration is quite cheap but may incur costs in wastage in materials. Software has different properties from building components, in particular its malleability and complexity. Reworking the whole architecture of a software system is also expensive because of the complexity involved – making internal changes does not incur costs in wastage of materials, as software is malleable, but it incurs other costs in terms of time and work invested.

Layers

A software architecture identifies a set of rules for decomposition – the assumptions used to modularise a software system. A major aspect of the architecture is the identification of the different partitions into which you can put the various pieces of software that are going to be used to provide your solution to a problem. Any proposed change to a system will have a different impact, depending on the software architecture chosen. Here we look at three ways of decomposing an architecture, with layers, with components and with services.

Figure 2 illustrates one instance of a **layered architecture** that can be found in distributed computing systems. The top layer concentrates on the presentation aspects concerned with the user interface, which are more prone to change than the rest of a software system. (It is natural to expect a number of requests from users to make a software system more usable.)

The application domain is concerned with support for the way a user performs a given task, such as the processing of a customer's order. A business may redesign the tasks that its employees perform, but perhaps not as often as amendments will be made to the user interface. Unless a business makes a radical shift in its core business concepts and transformations, the business services are less prone to change than the user interface.

The infrastructure contains the system support that usually includes the operating system and the databases, which allows the system to be more easily ported to new platforms.

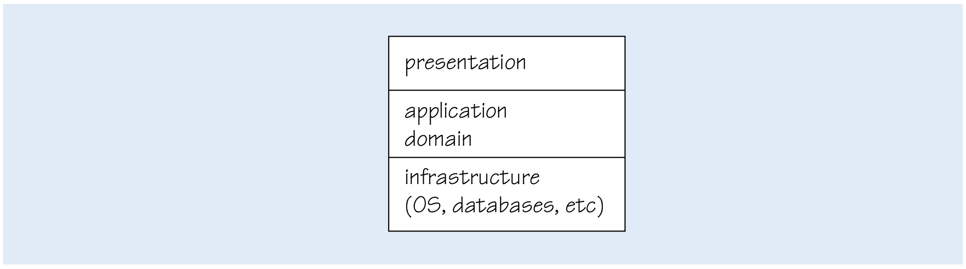


Figure 2 Three-layered architecture

Reuse is one strategy to deal with increased complexity of software and it addresses the concerns of maintainability, minimising costs, short delivery times and quality of software.

Components

We use the term **component** to denote a unit of reuse or replacement in a software system. A component could be a module or class with certain properties that make it reusable or replaceable in a given software architecture and it may depend on other components. Components are well understood, each with its own interface and context dependencies (see Figure 3 for representations of a component). It may be that a popular set of components is adopted as a standard. What is important is to have one, or possibly more than one, standard in mind when you are deciding whether or not a particular module or class is a component. Enterprise Java Beans (EJB), .Net and CORBA are examples of standards for components. The user of a component needs to adopt the same technology as that with which the component was developed.

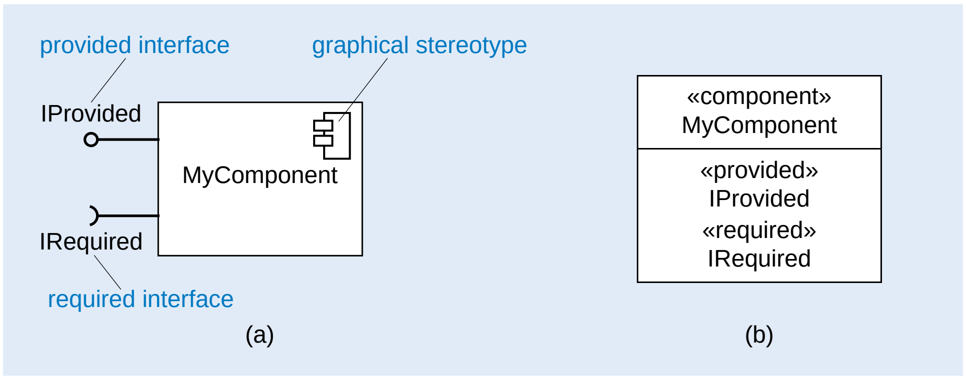


Figure 3 Two graphical representations of a component

Services

A similar concept is that of a **service**. A service is also a unit of reuse corresponding to a piece of functionality, described in a standard language, with published interfaces through which the service execution can be requested. A service, however, is technology-neutral in the sense that it can be invoked using standard communication protocols, while a component is usually technology-dependent as a client needs to use the same technology as the component.

A service is also discoverable, meaning that it can be used by clients independently of where it is located – service repositories can be accessed to locate services according to their definitions. A service-oriented architecture (SOA) structures software as a set of services.

The notion of services being accessed remotely through a web browser – software as a service (SaaS) – is popular now with systems such as those provided by Google (for example, Gmail and Google Docs).

Services may not be owned by the organisation developing the software systems that use them. They come with a service description and reside in a provider server. Programmers (consumers) who use them in their systems do not have access to the code that implements them – they need to find them in a registry of service descriptions and once found they can invoke them (see Figure 4).

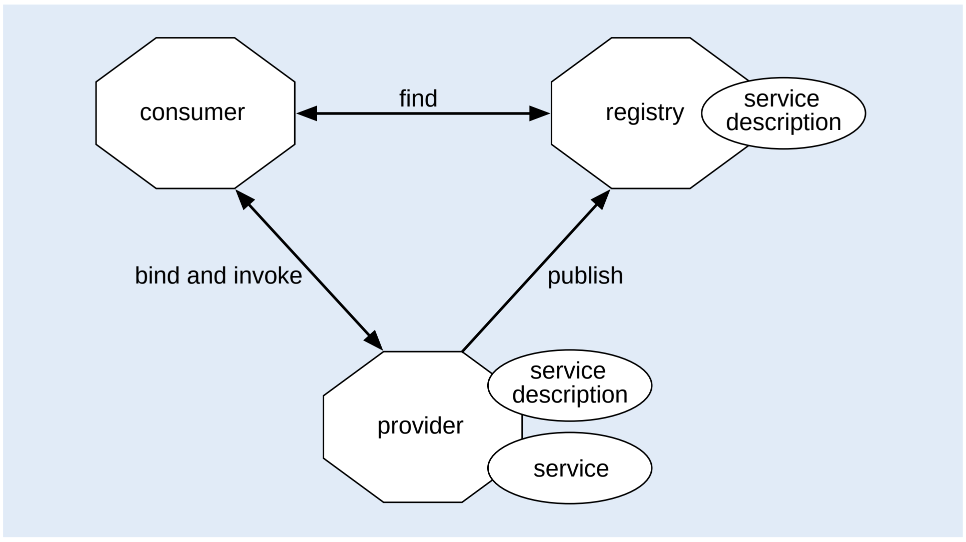


Figure 4 Services

Components tend to relate to entities, while services relate to processes. Components are assembled together through connectors (usually called glue) that are static structures, while services are bound at run-time in a dynamic way when they are discovered (Mašek et al, 2009).

Layers, components and services are different ways of structuring an architecture. They can also be used in conjunction with each other.

Activity 7 Reuse

Allow approximately 20 minutes.

(a) What are the characteristics of a component?
(b) How does the concept of an architecture contribute to component reuse?
(c) Which form of decomposition might be used in a software architecture?
(d) What are the similarities and differences between components and services?

Answer

(a) A component is a module that is considered to be a sufficiently good abstraction for the problem in hand. A component should be capable of being reused in future projects having the same software architecture, or being easily replaced at a later date within the existing software system. As with all modules, a good component has a well-defined interface and is an encapsulated abstraction of a well-understood concept, with strong cohesion and low coupling.

(b) The architecture of a software system embodies high-level decisions about the overall structure of the system, and this architecture may apply to more than one system.

(c) The basic form of decomposition used in a software architecture is partitioning to meet a number of separate concerns, each concern being addressed by a subsystem. For example, you might want to separate the user interface layer from the core business services layer, or you may decide to build or reuse components and/or services for some of the partitions.

(d) There are similarities between a service and a component. They both promote reuse and flexibility. They both use public interfaces to allow requesters to make use of their functionality without relying on their implementation.

There are differences too. A component is usually implemented in a specific object-oriented technology, therefore only clients compliant with that technology can easily communicate and integrate with it. In contrast, a service uses communication standards that allow the interoperation of diverse technologies.

Finally, components tend to be associated with business entities, while services tend to be associated to business processes – they may realise part or the whole of the functions within such a process and may involve several business entities. Be aware that although this is a widely accepted classification, not everyone follows it, and you may see components called services and vice versa.

1.7 Summary of Section 1

This section has briefly examined the nature of software, and identified the desirable characteristics of a software system. You have seen:

- that a good software system is one that meets its users' needs
- examples that illustrated the connections between the usefulness, usability, reliability, flexibility, availability and affordability of a software system
- that a software system can soon be out of date, as users' needs change with time, and that needs can often be missed during requirements capture
- that modularisation is the main method of dealing with the size and complexity of a software system
- the problems that arise with legacy systems
- the significance of maintenance
- the importance of software architecture.

Software systems are becoming all-pervasive in our society, and the demand for new systems is growing rapidly. There are significant risks associated with software systems when they are critical to a business. This all points to the need for software development processes that will deliver software systems that are easy to maintain and reliable, while at the same time ensuring that the systems serve some useful purpose for their users.

Good development processes will produce well-engineered components based on reusable architectures.

2 An introduction to software development

In this section we will introduce the basic activities involved in the development of software. Then, we will consider the general concept of a life cycle for software development and discuss examples of different life cycles. At the same time, we will consider the importance of models as part of software development.

2.1 Software development as an engineering activity

Software development has a great deal in common with the discipline of engineering, from which the term software engineering arose, and is said to be:

> 1 The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
>
> 2 The study of approaches as in (1).
>
> (IEEE, 1990)

Carrying out a systematic, disciplined and quantifiable approach implies management, and assumes that project management is a necessary activity within the discipline of software development.

As discussed in Section 1, words such as reliable, flexible and maintainable (among others) describe characteristics of software systems. They are all aspects of **software quality**. To obtain a high-quality software product requires a well-managed development process.

But the term *engineering* for software development is also associated with a few other factors:

1. Developers are concerned with meeting a set of requirements – there is usually an identifiable problem that they can solve.
2. There is a defined process that can be used to produce a solution and, within that process, there are a number of identifiable phases or activities.
3. There are tasks to be done in each phase that result in one, or possibly more than one, artefact related to the final product (software, hardware or document). Developers undertake different roles to perform such tasks (designer, programmer, tester and so on).
4. The quality of both the products and the processes by which they are made is important: the right product is being built (validation), the product is built in the right way (verification), and the product is behaving as expected (testing).
5. There are tools to improve the effectiveness and efficiency of the tasks performed by developers in their various roles.

6 There is a body of knowledge that developers might use and/or add to. There are ways of working to support the previous five factors (such as standards and rules for decomposition in a given context).

7 There is a recognised professional activity with its own code of practice and legal framework.

You might like to look at some dictionary definitions of the term engineering and see how the above observations fit them.

While there is considerable argument about the term software engineering, there is no doubt that software development does require certain skills and knowledge of certain ideas and principles. This section looks at what is needed to 'engineer software'. What should a developer do in order to produce a software system that meets the needs of its users?

In Section 1, you saw that there are three characteristics of software that affect its development and use – malleability, complexity and size – and that these all contribute to the likelihood of errors. Issues arising from size and complexity are addressed through modularisation. In order to ensure that the introduction of errors is minimised whenever a change occurs, you need to pay special attention to the development process.

Software development is a human activity. Software systems are developed by one group of people for another group of people to use. The success or failure of a project is determined by social as well as technical factors.

Users want a system that does what they need and that they can use. One that is technically superior but does not meet these criteria will fail. Users will resent, ignore or, in the worst case, completely reject it.

Activity 8 Software development

Allow approximately 15 minutes.

Give the characteristics of an engineering approach that support the argument that software development is an engineering discipline.

Answer

Software development follows an engineering approach provided that the following conditions are met:

- it is concerned with meeting a set of requirements that are defined as clearly as possible
- it uses a defined process with clear activities, each of which has at least one identifiable end product
- developers can apply their skills and experience to the tasks demanded of them
- validation and verification are regarded to be as essential as building the software itself
- it makes sensible use of tools and standards
- it follows a code of practice.

2.2 The role of development processes

A **development process** is a set of rules that defines how a software development project should be carried out, and a set of software engineering activities associated with the development of software. Each activity undertakes some clearly defined process, starting with a number of inputs from any preceding activities. On completion of an activity there may be one or more outputs, which are known as **deliverables**.

The order in which these activities are carried out is called a **life cycle** or **process model**, and outlines an overall process for the development of a software system. Project management and quality management activities permeate the life cycle of software development. A complete life cycle takes us from the first ideas about the need for a software system to its final withdrawal.

This approach of following a well-defined process has been promoted as the best way to meet the potential diversity of users' needs, and the primary mechanism to understand and harmonise conflicting demands on the development process. However, this approach has also been contested as too prescriptive, heavy on documentation, and not helping to meet the expectations of customers. In this section, we will start by presenting a well-defined process for software development but will also discuss some lighter-weight alternatives.

In Section 1, we introduced the notion that a good software system is one that meets its users' needs. In general, we will identify a **customer** as someone who pays for a software system, in contrast to the people who use it on a day-to-day basis – the **users** (although a customer may also be one of the intended users). A successful development project will deliver a product that meets, or even exceeds, the customer's expectations.

Software can also be developed without an identified customer – in that case it is successful if it can find someone who will become a customer and buy it.

We will assume that a customer (or someone acting as a customer) has assessed the feasibility of some initial ideas with a potential group of users and has decided that a software system *may* be required.

Typical technical activities for the development of software

There are many and varied methods used to develop software. However each one typically includes activities that can be roughly classified as follows:

- **Domain modelling**. Understanding the environment in which a system may be introduced – the business processes and rules. This is typically an activity that precedes a decision to develop a software system.
- **Requirements** (also known as requirements engineering). A set of steps including *requirements elicitation*, where you identify the problem, and *requirements analysis*, where you categorise, prioritise and model requirements. This defines what the system is to do.
- **Design**. Determining how you will solve the problem.
- **Implementation**. Acting upon the decisions made at the design stage.
- **Testing**. Testing what you have done so that you can determine whether or not you have solved the problem.

Different methods may subdivide the above activities or use different terminology. A specific approach, the Unified Process (UP), will be introduced in Section 3. By themselves, however, these activities are not enough to develop a good software system. Other activities are needed to a greater or lesser extent depending on the context, as you will see in this section.

You are likely to break up most problems into smaller, more manageable chunks, and deal with each one separately. It will then be necessary to bring the chunks together into a unified whole. This process is known as **integration** and is sometimes identified as a separate activity. Sometimes *delivery* of the software system is also identified separately, especially when there are contractual implications such as payment.

A software system is likely to change during its operational lifetime. This is the **maintenance** activity, which allows a software system to evolve in order to:

- correct errors
- adapt to a changing environment
- introduce enhancements required by the customer
- improve the software in anticipation of future changes.

The four activities of analysis (analysis is often used as a generic term for the activities that precede design), design, implementation, and testing are the ones you will see most often in diagrams depicting the process model of software development.

It is important to recognise that they are not the only activities involved in the process of developing a good software system. **Project management** and **quality management** are the two additional activities that hold the process of development together – the all-important glue for software engineering activities. Maintenance will inevitably involve the activities of requirements, design, implementation and testing, and will itself need to be managed, as illustrated in Figure 5.

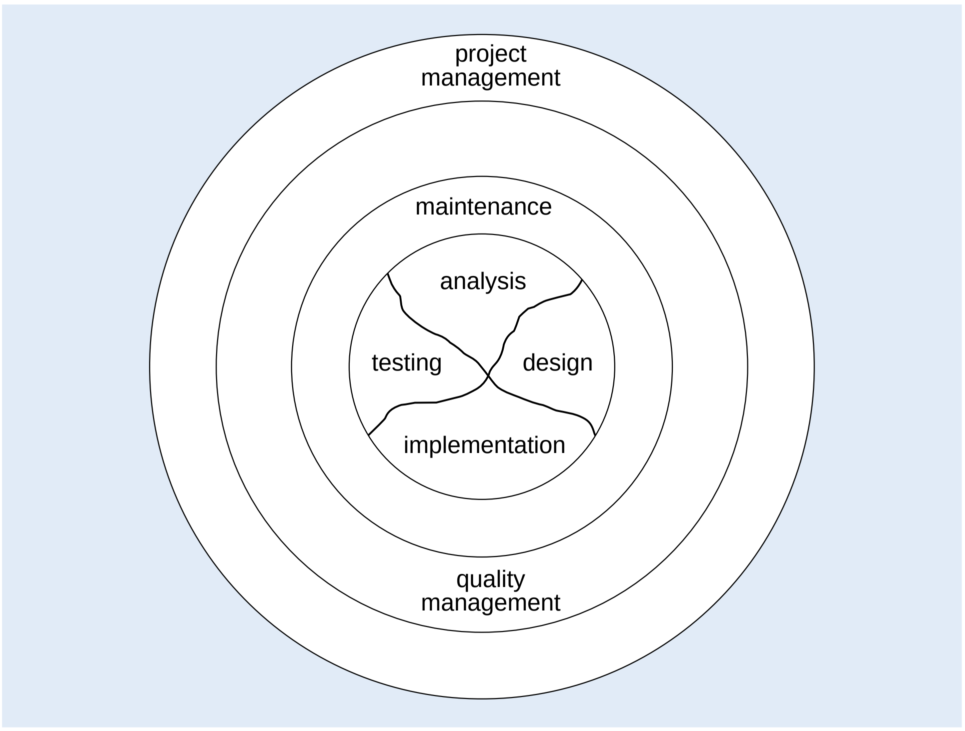


Figure 5 Activities for the development of a good software system

An overall process model

A process model (or life cycle) is a description of all the events and activities in the life of a software system or product and the sequence in which they happen. So you can choose how to connect the activities together to form a process model, which you can then use to elaborate a process for developing software. For example, if you arrange the five activities of requirements, design, implementation, testing and maintenance into a single sequence, you have the classic **waterfall model**.

However, in practice it is not usually possible to complete each activity correctly in one attempt. In addition, as development proceeds, the products of earlier stages become dated, as your understanding of both the software and its environment evolves.

If projects use a strictly sequential process model, a working version of the software system will not be available until late in the testing activity. This will represent a long wait for both customer and users, who won't be able to see a working response to their requirements until the final product is finished. In addition, any errors detected in the working version of the software could be disastrous as it would be too late to correct them. Real projects rarely follow a purely sequential process model. The act of reviewing is an important activity when testing the quality of any development process and its resulting products.

An alternative process model is to iterate around one or more of the activities. **Iteration** allows the developers to improve the outputs from a given set of activities and get feedback before moving on to the next activity. In particular, iteration allows a group of people, usually developers, to perform a review of a sequence of activities, or of an activity and its outputs.

In general, reviewing a proposed solution provides the feedback necessary to modify it and improve the solution. Think what happens when you need to tune a guitar or violin. You pluck a string, a note sounds, and you adjust the tension on that string. You repeat the process until you get the desired pitch.

It is often difficult to identify all requirements and state them explicitly at the outset of a development project. It is a good idea to start with a subset of the requirements and incrementally grow the system with feedback from each iteration. This approach is known as **iterative and incremental development**. As shown in Figure 6, each iteration is a complete, small project, with a short, fixed timeframe (**timeboxed**), consisting of requirements, design, implementation, testing and integration, and resulting in a partially working system. Each of these repeated short iterations adds complexity until the final system is produced.

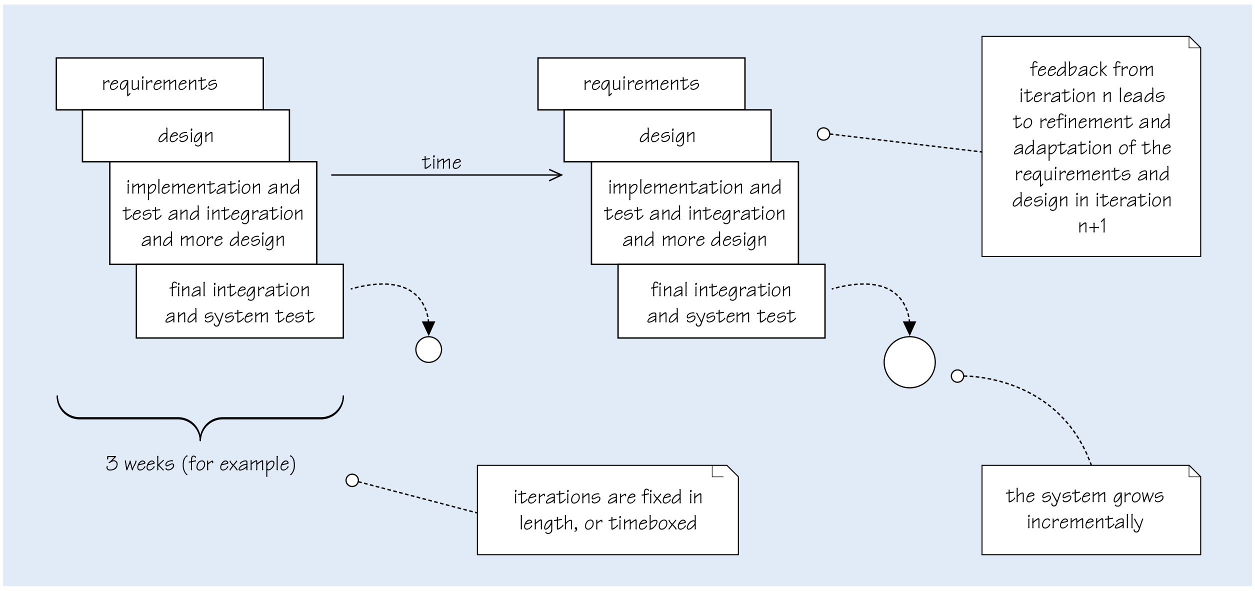

requirements
design
implementation and test and integration and more design
final integration and system test
time
requirements
design
implementation and test and integration and more design
final integration and system test
feedback from iteration n leads to refinement and adaptation of the requirements and design in iteration n+1
3 weeks (for example)
iterations are fixed in length, or timeboxed
the system grows incrementally

Figure 6 Iterative and incremental development process (Larman 2005, p. 20)

In an iterative and incremental development, users obtain useful and usable software quickly. This method also enables the developers to take on board feedback from users as the software develops – an increment may simply be an enhancement of the previous version. Increments can be developed sequentially or in parallel, depending on circumstances. For example, a small team might choose to develop increments sequentially, according to a priority agreed with the users.

Activity 9 Development process

Allow approximately 20 minutes.

(a) What is a development process?

(b) What should the role of project management be with respect to the deliverables of a development project?

(c) What is the difference between a customer and a user?

(d) Suggest a reason why maintenance is a core activity in the development of a good software system.

(e) What additional task is needed when the development of a software system is partitioned into a number of increments?

(f) What are the assumptions on which the waterfall model is based?

Answer

(a) A development process is a set of rules that defines how a software development project should be carried out. It incorporates a number of activities, and a process model (or life cycle) that indicates how these activities are ordered.

(b) A good software system must be affordable and available within an appropriate timeframe. Each deliverable uses resources, such as developers' time, that add to its cost. Project management involves the identification of the appropriate deliverables for a given set of requirements and controlling the cost of producing them. Project management also involves ensuring that deliverables are produced on time and taking steps to cope with any delays.

(c) A customer is the person who pays for a software system, whereas a user is someone who will use that system on a day-to-day basis. A customer will also be a user when the proposed software system is intended to support their job.

(d) Maintenance allows a software system to evolve over its operational lifetime so that it continues to be useful.

(e) A task devoted to the integration of the increments to form the final software system will be needed.

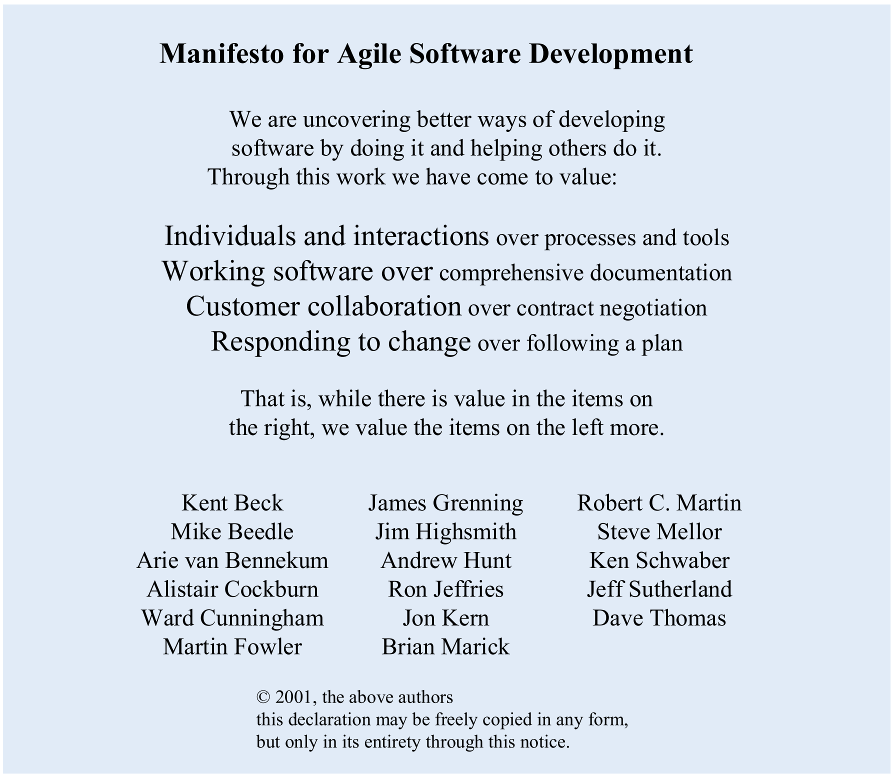
(f) There is an assumption that once a particular activity or phase is finished, it is not re-entered, and that the activities do not overlap but follow each other in a sequential life cycle. There is no need for reviewing or reworking. All this presupposes that an end point for each activity can be identified.

Agile development

Agile development has become, in the last 20 years, a popular approach to software development. It is an umbrella term used to describe a variety of (agile) methods that promote a set of practices that encourage simpler, more light-weight, faster and nimbler software development that can adapt to the inevitable changes in customer requirements. The continual realignment of development goals with the needs and expectations of the customer aims at software that better serves its purpose.

As seen below in the manifesto, agile development is an approach to software development that puts people and working software at the forefront of the development process.

The movement towards a more agile way of developing software has been gathering momentum for a long time. Many of its proponents were associated, throughout the 1990s, with several approaches known as *light-weight* in opposition to more prescriptive approaches to software development. In 2001, leading proponents of the agile approach got together and wrote a manifesto (Figure 7).



Manifesto for Agile Software Development

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

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

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

© 2001, the above authors
this declaration may be freely copied in any form,
but only in its entirety through this notice.

Figure 7 Manifesto for agile software development

Agile practices of simpler, lighter-weight, faster and nimbler software development that can adapt to change are best practices that are not new. Iterative and incremental development, for example, has been promoted for a long time in the short history of software development and adopts some of these practices. However what the agile approach does is to promote the implementation of those practices with well-defined rules. The emphasis on the people involved in the development is also common to all agile approaches and is the basis of the sense of fun, motivation and productivity usually associated with it.

The agile movement also has its deprecators and critiques, one of them being that agile methods do not scale up to larger systems. It is however accepted that some old practices of software development sometimes lead to problems and agile practices, although not a panacea, may bring better ways of working.

Extreme programming (XP) (Beck, 2004) is one of the best-known agile methods. It is a light-weight method, based on intensive testing and incremental development. It defines a series of practices about how individuals and teams should work, how the environment should be set up, and how the work should be carried out. These practices include incremental design, test-first programming, programming in pairs, continuous integration, planning for the week, and so on. Scrum (Schwaber and Sutherland, 2011), which is also a popular agile approach, defines a set of roles, events, artefacts and rules. All events are timeboxed and have well-defined rules – scrum events comprise:

- the sprint – a development phase, no longer than a month, that has as a deliverable a useable working increment
- the sprint planning meeting – lasts no more than eight hours
- the daily scrum – a daily meeting, no longer than 15 minutes, looking at what has been done and planning the work for the next 24 hours.

Many of the agile practices are geared to better communication and collaboration among developers.

Several agile practices have been recognised and incorporated in other more prescriptive development processes, in particular the emphasis on people as opposed to process, short iterations and the acceptance that systems change. The term agile can be found now in many contexts, and it is interesting to see, for example, a report from a UK government agency, the National Audit Office, on *Governance for Agile Delivery* (National Audit Office, 2012).

An agile approach requires experience to be able to pick, choose and adapt the elements of a development process that best suit a real situation. This is often what happens in practice and many development processes get adapted rather than used as mandated. By learning a set of techniques and a process to apply them, and by discussing possible alternatives you will be in a better position to make these decisions. Software development is not an exact science and many factors, such as context, the organisation and the problem will dictate how development proceeds.

We will be using the term agile (with a small 'a') to refer to best practices, as opposed to Agile (with a big 'A') to refer to specific agile methods. We will use the term **plan-driven development** to distinguish traditional, more prescriptive approaches to software development from agile development approaches.

2.3 Choosing an appropriate process

By definition, a good software system must be fit for its intended purpose. It should therefore be evident that because software is needed for such a variety of purposes; there is no single development process that will suit all purposes. Consider the following reasons for building systems:

- to control a series of gates at level crossings
- to control a manufacturing process for chemicals
- to manage an international stock market
- to manage a supermarket
- to manage a public lending library
- to administer the activities of a university
- to help you manage your personal finances
- to control your television and video recorder
- to play a game on a mobile phone
- to manage electronic funds stored on a smartcard.

The people who use these systems will have different views about what it means to have a software system that is useful, usable, reliable, flexible, available and affordable. So it should be no surprise that there are different development processes for different types of system. Indeed software companies often specialise in developing software for specific kinds of business, such as banking or manufacturing.

Choosing the appropriate level of formality

The following two examples illustrate how the amount of formality in a development process varies.

Example 10

Government legislation often requires software systems to be built to support new administrative functions. When developers compete to supply such software systems, they are asked to conform to government-approved standards. One of those standards is known as PRINCE, which stands for 'PRojects IN Controlled Environments' and relates to the management of projects and the quality of the developed software.

Commonly, developers must be able to show that their development process conforms to the regulations set out in PRINCE. In addition, developers may be required to adopt an approved process for their analysis and design activities. The most common of these for use with PRINCE is known as the structured software analysis and design methodology (SSADM).

Example 11

In certain financial areas, being first to offer some new service is the way to make the most profit. In other words, time-to-market is critical because if you miss out your competitor takes the spoils. When there is a need for a software system to support such a service, development is almost a race against time. Every aspect of the development process has to be tuned to meet the time-to-market.

So the aim is to do the simplest things needed that could possibly work when delivered. It also means that communication overheads must be minimised, so there will be few developers (say, two) who work to a minimal set of agreed procedures. Also, if the amount of actual code produced is small, the chance of introducing errors is reduced.

Of course, there must still be testing to ensure that the software performs its intended purpose, but if the complexity of the software is also minimal, there will be fewer tests to do.

Examples 10 and 11 show that there is no single development process that is appropriate to all kinds of software product. The amount of information recorded during development may be different in the two examples.

Example 10 illustrates the need for a formal development process. Although it is a slow and deliberate course of action, the resultant software system must be able to deal with all the nuances of the legislation. The developers must be able to show that each aspect of the legislation has been incorporated into the final software system. Software configuration management is the discipline of managing and controlling change in the evolution of software systems (IEEE, 1990, p. 20).

The control and maintenance of the integrity of a project's deliverables (ensuring that they do not become corrupted or inconsistent) is the responsibility of the **configuration management** activity, which is related to the need for integration.

In contrast, Example 11 relates to software that is required for a specialised purpose with a minimal development time. The developers have the benefit of a narrow scope for the software system, and the short development time means that it is unlikely that the users will request many changes, further reducing the demands on development.

In general, the size of a project influences the choice of development process. As illustrated in Example 11, the amount of formality in the process can be minimal. Small teams of up to 10 people can communicate and respond to changes informally.

In larger projects, such as in Example 10, a well-defined procedure for controlled development is necessary – an informal communication network on its own is not enough. One approach to the solution of the problem of large projects is to split the group into smaller teams according to the responsibilities set out in a given development process. Just as modularisation is the way to deal with the complexity of a software system, developers can be assigned different activities and deliverables within a given project. This partitioning allows developers to specialise and become analysts, designers, programmers and so on.

Agile development encourages collaborative development to reduce the problems of communication that can arise in large projects. XP promotes pair programming where the code is written by pairs of programmers to encourage communication, feedback, encouragement (Beck, 2004), and in Scrum the daily scrum meeting promotes team awareness, with every member of the team knowing what others are doing. In Example 11, developers may opt for an agile approach with pair programming, as in XP, and quickly deliver an initial version that is then updated on short iteration cycles.

At some point, a software system will have outlived its usefulness. The developers should consider the expected lifetime of the prospective software when they assess the risks of producing a successful product. In Example 11, the time-to-market is a significant factor, but the expected lifetime is likely to be short because of the nature of financial markets. So a slow and deliberate development process is inappropriate.

Dealing with risk

A software solution to a problem needs to be considered in the broad context of the domain to allow you to manage the risks associated with a development project. For example, if delays cause the team in Example 11 to miss the market 'window', it does not matter how much software has been developed, because it is no longer of benefit to the customer and users. Assessing risks and taking steps to reduce them are important activities in software development – this is known as **risk management**.

In a typical project, the major risks are around the requirements. Do you understand them? Have you got them all? Are they changing too frequently? Anything that you can do to increase your confidence that the requirements you have are both necessary and sufficient can reduce the risk of project failure. In general, for every decision you make there is a risk that your decision is wrong, or at least inappropriate, and you should consider identifying and reviewing major decisions. If the risks cannot be overcome, the project is unlikely to succeed.

An agile approach takes the view that requirements will change during development, and therefore they should be under continuous review. By involving customers throughout development it is easier to address changes in requirements and reduce the risk of making the wrong decision.

In any project there is likely to be a trade-off between what can be delivered in a given time to a specified budget and the functionality of the software system. Often the number of desirable features of a solution exceeds those that can be delivered for a given price and within a given timescale, so choices have to be made. One way to make such choices is to estimate the risk of not delivering each feature. That is, if a feature were not to appear in the final product, what effect would this have on such things as the usability, usefulness, reliability and flexibility of the product?

Figure 8 illustrates a simple spiral process that deals with risk explicitly and can be used in the development of a software system. (There are many interpretations and variations upon Barry Boehm's original spiral (Boehm, 1986).) There are four steps that are repeated with each iteration of the spiral:

- determine the objectives, the alternatives and the constraints

- evaluate the objectives and identify and resolve the associated risks
- develop and verify a (partial) solution or product
- review that solution, and plan the activities for the next iteration.

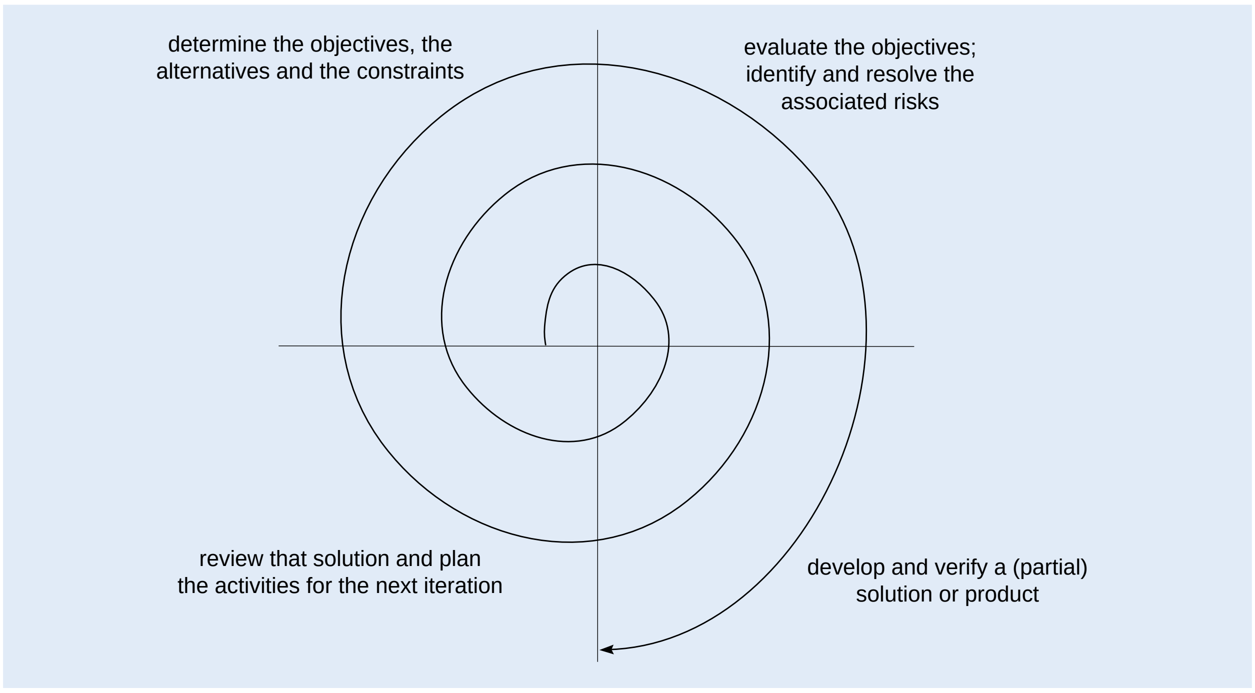

determine the objectives, the alternatives and the constraints
evaluate the objectives; identify and resolve the associated risks
review that solution and plan the activities for the next iteration
develop and verify a (partial) solution or product

Figure 8 Spiral process to deal with risk

The spiral process starts when it is recognised that a particular organisational process can be improved or supported with the aid of a software system. The distance from the origin is intended to show how many resources have been used, the cumulative cost of a project.

There is no need for a complete solution to be produced by the end of the first iteration of the spiral. For example, the first iteration could focus on the question ‘Can we build an acceptable software system with the resources that can be brought to bear?’ After each iteration of the spiral, new risks come to light and plans are made for the next iteration in order to resolve those risks. With successive progressions, you should reach a point where your review indicates that you have an acceptable solution – a software system that meets the needs of its users.

Agile development follows, loosely speaking, a spiral approach and has mitigation of risks as an important concern. There are however some indicators that would distinguish an agile process from the generic spiral model. In agile development the short timeboxed iterations, for example, result in partially working systems, and an iteration would, typically, be no longer than 1 month.

Such a risk-driven model is also helpful when developing large software systems or systems where the developers have little experience of the problem domain. In both cases there is a high risk of failure. For example, if you have never built a real-time system, you need to gain some understanding of the scheduled execution of tasks.

Activity 10 Issues in software development

Allow approximately 20 minutes.

(a) Why might a software development company specialise in a certain kind of customer, such as those in banking or health care?

(b) In which of the activities in Figure 5 would you expect to do your configuration management during a project?

(c) Why are there additional risks when developing large projects?

(d) What is added to a development process with the introduction of risk management?

Answer

(a) Through specialisation, a software development company can foster experience in a given domain, whether it is banking, health care or any other field of interest. The developers in that company would have (or would hope to gain) sufficient knowledge to understand the problems raised by the users and therefore be able to present solutions in a form that can be understood by those users. In addition, the company may develop and use a consistent development process that is appropriate to the set of customers in that domain.

(b) Maintenance deals with change. Configuration management is the discipline of managing and controlling change, and so you would expect maintenance to be where you would perform many of the configuration tasks. However there is a role for configuration management during the development process in, for example, ensuring the consistency of models. Quality management is the activity in which you would perform these tasks.

(c) The chances of failure increase as the size of a software project increases, as more errors are likely to be introduced. Effective communication between the members of a large team also becomes more difficult.

(d) The most important additional aspect is the use of the identification, evaluation and reviewing of risks that are carried out with each iteration of the development. These steps introduce feedback into the process to help ensure that the deliverables at each stage are leading in a timely manner towards the correct product, and risks are controlled.

2.4 Traceability

The need for a software system often comes from a set of potential users, who may not be employed in the same company as the developers of the software system. Sometimes software development has no clearly identified customer other than the developer. Software is developed hoping that someone will buy it.

Initially developers must find out about the users' domain and express their understanding of the problem in a form that is appropriate for the proposed development process.

If a plan-driven process is followed, early in the process developers produce a requirements specification document that identifies what the proposed software system should do and the environment in which it must work. It relates what the users want to what the developers aim to provide. This implies that there is a need to record more than just the requirements. It allows for the tracing of the history of each requirement from its origin in the problem domain, through the various intermediate activities, so that it is possible to reconstruct the significant events that led to the final operational software system.

The ability to trace the history of each requirement is known as **traceability**. Among other things, traceability is essential for dispute resolution (knowing what was agreed during a project), for seeing the effect later in the project of changing or deleting requirements, and for demonstrating that you have dealt with each requirement. The chosen development process will determine which events can be reconstructed and hence determine the level of traceability. It is important to recognise that the amount of documentation should be commensurate with the desired level of traceability – neither too much nor too little.

Agile approaches are sometimes identified with the demise of documentation. The agile manifesto contrasts working software with comprehensive documentation and agility has emerged as a reaction to heavy documentation as required in rigorous management methods (such as the capability maturity model).

However, agile approaches take a pragmatic approach to documentation and traceability (see Example 12). They emphasise the balance between the amount of documentation and the pay-off it brings. Heavy documentation is difficult to keep updated and is sometimes never used. Agile documentation should be simple to understand, have a well-defined purpose and should maximise the pay-off of the effort that is put into it. The involvement of stakeholders in an agile project will help determine the amount of documentation required.

Example 12

An agile project has documentation in the form of user stories and tests. A user story describes some functionality required by a user, while a test is an executable form of a user story and therefore directly related to it. Tests are written before coding and are also directly related to the code produced. Changes over time can therefore be tracked through with the help of a tool that manages source code, and traceability can be preserved.

On a given project, there may be a number of deliverable documents that record your activities. You should treat your documents as part of the explanation of what is done. When you write down what you know about a problem, it helps to clarify your understanding of that problem and helps you to communicate with others and share the same mental model. Furthermore, writing and drawing helps you to explore the problem and the potential solutions.

Project notebook

Remember that the program code is a necessary but not a sufficient deliverable if there is a need to understand decisions taken. Documentation adds to your explanation. However the main purpose of software development is to produce quality software and documentation should serve that purpose. The approach taken to software development, the problem being addressed, contractual obligations and other factors will determine how much to document and what to document.

Keeping a project notebook is a disciplined approach to organising your thoughts and actions as a software developer. Your project notebook is a record of notes, thoughts, drawings, ideas and decisions (and the reasons for taking those decisions) as you work on a project. In its simplest form a project notebook will be paper based, but it could be recorded in files on a personal computer.

For it to be effective, you should keep your project notebook with you at all times, so that you can use it, save it and review it. There is no limit to what you can record in the notebook. For example, you can note down what your users say about their needs, make some preliminary sketches of models or even record telephone numbers and URIs that may be useful while at work.

Whatever you record, you should date-stamp your notes, and review them on a regular basis to ensure that your questions have been addressed and that you have extracted the information that is useful on a given project. You must keep accurate dates and times for the information recorded in your notebook for the following three reasons (all of which relate to traceability):

- your project notebook may be required as evidence in some enquiry or even in a court of law
- it helps you to review what you have done and how long you took to do some task
- it facilitates learning.

Activity 11 Traceability

Allow approximately 30 minutes.

Part A

(a) Why is traceability important to the development of software?

(b) How does documentation contribute to traceability within a development project?

(c) Which activity or activities in the development process (shown in Figure 5) are most affected by poor documentation?

(d) For what kind of software system might you minimise or even avoid any documentation?

(e) How would you characterise agile documentation?

(f) Why is it important to review the contents of your project notebook?

Answer

(a) Traceability is important for the reconstruction of significant events. In software development it should be possible to follow all the activities undertaken in response to a proposed change. In particular, you should be able to trace backwards from an implemented component or components, through their design, to a given requirement.

(b) Within a development project, documentation records the progress from requirements to implementation (and beyond). It is possible to identify each requirement and follow the actions taken to implement a solution to that requirement. You might, quite simply, be required to show that you have implemented each requirement correctly.

(c) All seven activities will be affected by poor documentation. The maintenance and quality management activities will be most affected because they rely on the existence of traceability within the outputs of a development activity. (You saw this problem earlier when we looked at legacy systems.)

(d) If a proposed software system is likely to have a short lifetime and will be discarded after use, it may be acceptable to minimise or avoid the task of documentation.

(e) Agile documentation should be gathered with a purpose, should be easy to use and above all should justify the effort put into gathering it. There is no reason why this shouldn't apply to any kind of documentation but often software development produces heavy documentation that is rarely used.

(f) The act of reviewing your notes helps you to identify what worked and what did not, as long as the notebook records accurately what you did and when. A review helps you to trace the events that led to the decisions you made since your previous review. Regularly reviewing your notes also enables you to check that you have followed up on all the decisions that you made.

Part B

What are the main problems associated with the development of software? You should be able to identify at least six.

Answer

Here is our list of problems that affect software development (you may have identified others):

- difficulty in capturing all the user requirements
- difficulties in managing and developing large and complex systems
- not meeting users' requirements and avoiding dissatisfaction with the completed software
- continual changes in user requirements (often referred to as requirements creep)
- maintaining poorly documented software
- the presence of errors introduced unknowingly during development
- a changing world leading to changes in the needs of the users and decreasing the usefulness of the software
- low productivity of those developing software, and difficulties in measuring that productivity
- ensuring the quality of a software system.

Part C

(a) How do you think a large-scale project differs from a small-scale project in terms of the management of the development team?

(b) What management steps might be taken to tackle the problems that arise in a large-scale project?

Answer

(a) As the size of a project increases, so does the number of people involved, and consequently the complexity of the interactions involved, and the difficulty of managing the team. As with programs, the greater the complexity of interaction, the greater the chance of errors occurring due to poor communication.

(b) One of the functions of software project management is to manage project complexity. One way to achieve this aim is to modularise project teams by splitting them up into a number of groups. Each group performs a specific task and communicates with the other groups through one person (for example, the head of the group). Another way, encouraged by an agile approach to development, is to encourage practices of communication and cooperation in development, such as working in pairs and having daily short meetings.

Part D

Process models sometimes show only the software-creation activities and exclude software maintenance. What problems are associated with the use of such models?

Answer

If maintenance is not mentioned, it is likely that it will not be considered to be an integral part of the overall software process model. This may result in a design that does not take the needs of maintenance into account, and hence a product that is difficult and expensive to change. This is especially likely if time and/or budget are already tight.

Part E

What kinds of software system would be best suited to an iterative and incremental development process?

Answer

Software systems where the problem can grow incrementally or where it falls naturally into partitions, each of which represents a fairly self-contained unit that could be developed and delivered on its own to provide the users with a useful chunk of functionality, would be well suited to this model. An iterative and incremental approach allows several design issues to be tackled simultaneously and enables the system to be implemented in relatively small steps in order to contain the complexity. Of course, this assumes that the design can be partitioned.

Part F

Figure 8 shows a spiral process for software development. Into which quadrant of that figure does each of the seven technical activities of Figure 5 fit? Notice that there is not a one-to-one match.

Answer

The analysis, design, implementation and testing activities all fit into the ‘develop and verify a (partial) solution or product’ segment. The analysis activity also overlaps the evaluation quadrant where objectives and risks are analysed. The testing activity overlaps the ‘review and plan’ quadrant. The maintenance, project management and quality management activities (including configuration management) operate in all four quadrants.

Part G

It has been said that 'the maintenance activity begins with the first deliverable of a development project'. What does this imply about the maintenance activity?

Answer

This refers to the fact that there could be deliverables, such as requirements documents, that exist prior to any software being built, and these also require maintenance. For example, you might detect errors or inconsistencies in a document that lists the important terms in your users' domain, or you might have to respond to changes in the environment, which may be due to new regulations or market forces.

2.5 Summary of Section 2

This section considered how you might approach the development of a good software system as follows.

- Large software projects are prone to problems because of their size and complexity.
- Software is inherently easy to change, and this makes it easy to introduce new errors ('break' it).
- Software development is similar to engineering when it involves a defined process with clear activities, each of which has one or more products that can be tested against the users' requirements.
- A basic process for the development of software is a set of rules that define how a software development project should be carried out. It incorporates a number of activities, and a life cycle (or process model) that indicates how these activities are ordered. It helps to coordinate and control the use of those methods and any tools that support them.
- Process models can be sequential, iterative, incremental or some combination of these.
- A disciplined approach to software development implies that you must make some effort to record your activities from both a personal view and a project view. It means that documentation is an important task for all those who work on a project. In particular, it must be sufficient for the level of traceability required.
- An agile approach to software development puts an emphasis on people rather than on process or documentation, on short iterations and quickly working software, and on the acceptance that systems change. It also encourages practices that promote cooperative work in software development.

3 Modelling in software development

In this section we will consider the importance of models as part of software development, present a popular software development process and introduce one approach to the software development process.

3.1 Importance of modelling

Modelling is a way of thinking about things and ideas in the 'real world'. A model is an abstract representation of a situation in reality, or of a system from a particular point of view. In effect a model is a way of expressing a particular view of an identifiable system of some kind.

In terms of the development of software systems, **models** are:

- a way of understanding the problems involved
- an aid to communication among those involved, especially between the developer and the user, as part of some deliverable
- a component of the processes used in development activities such as analysis and design.

A good model is an abstraction, which allows those involved to concentrate on the essentials of a (complex) problem by keeping out non-essential details. Previously, you saw that there is a limit to how much a person can understand at any one time. So we build models to help us in activities such as the development of software systems. For example, developers build different models at different stages during the development process in order to confirm that the eventual software system will meet the users' requirements.

Models contain a representation of the significant states, objects and events in a real-world situation or within a software system. In one respect models are an idealisation, because they are less complicated than reality, and so they are easier to use for software development. The benefit arises from the fact that only the relevant properties of the outside world are represented. For example, a road map is a model of a particular part of the earth's surface. We do not show things like birds' nests as they are not relevant to the map's purpose. We use a road map to plan our journeys from one place to another, and so the map should contain only those aspects of the real world that serve the purpose of planning journeys.

When we model, we are trying to show or reveal what something is like. Models can do more than this because they can help us explain the past and the present (the problem), and they can help us predict and control the future (the solution – a software system). However remember that the model and the real world are alike in some ways, but different in others. For example, road maps are helpful because they represent distances between (and relative positions of) places, as well as routes between them. They use the relevant properties of the real thing with just a change in scale – one centimetre on the road map, for instance, may be equivalent to one kilometre on the ground. They may however be unhelpful if they show only major roads.

We can also use one property in a model to represent another in the real world. In the case of the road map, we can use different colours to represent different classes or types of road. Such a road map should have a key or legend so that those who read the map can understand what the different coloured lines are intended to represent. On a road map, a blue line might represent a motorway, and a yellow line a narrow track.

Models of a problem situation are only an approximate representation of that situation. The real-world situation is normally complex – so much so that an exact representation is likely to be impossible to achieve. The problem therefore confronting a developer is to find some way of achieving an acceptable balance between completeness and manageability.

In a software development project, there will be a number of practical considerations that result in some compromise in completeness.

If the constructed model is so complex that the developer (or other team members) cannot use it, it is of little or no value. So the model must simplify the reality a good deal. However the developer should make explicit all simplifying assumptions for a given model, as well as their consequences. At some point in a development project, any of these assumptions may need to be justified.

Models are subject to change. At the very least, they require some form of periodic testing so that a model can maintain its correspondence with reality. As towns and cities expand and contract, a road map must be changed to reflect the new situation. In the worst case, a change in scope necessitates a whole new model. For example, if there were a need to reflect the current status of roads and the traffic density on them, a simple road map would be inadequate.

Agile modelling

Agile modelling is Scott Ambler's approach to lighter modelling (Ambler, 2002). It proposes a set of values, principles and practices to help software developers become agile modellers. These are mostly common sense but reinforce the alignment of modelling with an agile approach. Modelling should not be seen as a routine exercise that has to be done independently of how models are going to be used. Models should only be done with a purpose and up to the point where they stop being useful. There is no justification in spending too much time in getting a model right (consistent, complete, accurate) if the only aim is to communicate an overview of the structure that is in the developer's mind. A typical activity of an agile modeller is to stand up with others in front of a whiteboard, sketch a few models, discuss these sketches, and then discard them if they serve no further need.

It requires practice to be able to pick, choose and adapt the modelling techniques that best suit a real situation. But that is often what happens in practice and many software organisations adapt development processes to their needs rather than using them as mandated. However there are also situations when it is not easy to assess how much effort should be put into a development activity, and in that case it is worth carrying modelling through in a systematic way to achieve a more complete view.

There are also tools to help in the creation of models from existing software. They can be used to document a piece of code that has not been modelled throughout development.

A standard notation

In developing a software system, a developer will not just use a single 'catch all' model – a set of related models is more likely. It would be preferable to have a consistent way of representing each of the different models for a given software system. A **modelling language** allows the developer to make useful connections between those different models. In software development, a modelling language is often based on diagrams and their construction, meaning and use. There are two sets of rules within a diagram-based modelling language:

- one that determines what diagrams exist and what symbols are used on each one – its *syntax*
- one that determines what the diagrams and symbols mean – its *semantics*.

What should you look for when choosing a modelling language? On any given development project there are decisions to be made about every model. You should favour a modelling language that:

- allows you to express the many facets of the subject of your model
- helps you to resolve misunderstandings rather than introducing new ones
- is easy to learn and use – you want to make progress quickly
- is supported by tools that allow you to use your modelling skills rather than your drawing skills
- is widely used and is accepted within the industry.

It is advantageous if a modelling language is widely used. If you leave one team of developers and join another that uses the same modelling language, you need only learn about the new problem situation. The advantage is amplified if you move to a new team in a new company. Naturally, the team that you join will expect you to be more productive than you would be if you had to learn a completely new modelling language.

Unified Modeling Language (UML) is now accepted as the standard object-oriented modelling language. It is intended to be a general-purpose language for software development. It is not meant to be a complete language for modelling all aspects of all systems.

Its success is partly due to its separation from any particular method. It is available for anyone to include in their own method for software development. The Object Management Group (OMG), an industry consortium for modelling and integration standards, has adopted UML and is the main body responsible for its development.

UML is predominantly diagrammatic, but does allow developers to add text in appropriate places.

Activity 12 Modelling

Allow approximately 15 minutes.

(a) What is a model?

(b) What is a 'good model'?

(c) What are the two kinds of rule that govern the use of a modelling language?

(d) Does a modelling language need to be associated with a particular development process?

(e) What are the required characteristics of a standard modelling language?

(f) How does a standard modelling language contribute to software development?

Answer

(a) A model, in terms of software development, is an abstract representation of a specification, a design or a system, from a particular point of view. In general, it is a simplification of reality, created in order to understand an aspect or viewpoint of the system in question.

(b) A 'good model' is an abstraction that allows those involved to concentrate on the essentials of a complex problem by excluding non-essential details while capturing those of interest.

(c) A modelling language is normally diagrammatic, although it can be textual. In common with natural language, there are two distinct kinds of rule:

- those that determine whether or not a diagram is legal – the syntax of a diagram
- those that define what a legal diagram means – the semantics of a diagram.

(d) No, a modelling language does not dictate how it should be used and it is up to a development process to define which notations are appropriate and how they should be used. With experience practitioners tend to pick and mix from different modelling languages and use the notations that are most appropriate to the task – most modelling languages do not provide notations for all types of tasks.

(e) When choosing a modelling language, it should be:

- sufficiently expressive
- easy to learn and use
- unambiguous
- widely used
- supported by suitable tools.

(f) A standard modelling language helps when new people join a project – a common modelling language reduces the time needed to enable them to become productive team members. Also, when a modelling language is widely used, it is likely that project components will have been constructed using that language. This makes the software easier and cheaper to maintain.

3.2 Models illustrate points of view

To understand the architecture of a software system, you will need a number of complementary and related views. If the main influence is the users, a view that expresses their requirements is essential. This is quite normal. Every development process you will encounter will encompass one or more views of the software systems within its scope.

During software development, you will be interested in different aspects or *views* of the problem, and its solution at different times. It follows that you will construct different models to suit those aspects. As mentioned above, it is unrealistic to expect to put everything into just one model. Too much detail in a model can only be a distraction.

When it comes to the development of an object-oriented software system, there are two distinct kinds of model:

- **structural (or static) models**, which describe the objects in the real world or in a (software) system and their relationships to other objects
- **behaviour (or dynamic) models**, which describe the behaviour in the real world or of a (software) system over time.

In practice, such a simple partition of a software system is not enough. For example, in a distributed system a developer must also consider the potential location of the modules (or classes). So the overall process of software development takes a number of different views into account. When developing the software architecture you will be identifying the differing views. In practice, the developers are likely to produce a system architecture for the software system during certain project activities such as analysis or design. Each view can be thought of as a model that expresses a particular aspect of the overall system – each one is an abstraction. The views interact with each other.

Activity 13 Models

Allow approximately 15 minutes.

(a) Should you try to capture everything about a design in a single model?

(b) What is the difference between a structural and a behaviour model?

(c) Do the models used in a given development project need to be consistent?

Answer

(a) No, because you will be interested in different aspects of a design at different times, and different models of your design will be built to reflect your interpretations of users' needs.

(b) A structural model describes the elements of the system and their relationships to other elements. A behaviour model describes the behaviour of a system over a period of time.

(c) Yes, the whole set of diagrams should contain the different aspects of a single software system, so they should not contradict one another. For example, there must be some consistency checking between the static and dynamic models. This can be automated by a suitable tool.

3.3 Introducing the Unified Process

The **Unified Process** (UP) (Jacobson et al, 1999) has emerged as a popular iterative and incremental development process for building enterprise systems based on an object-oriented approach, and using UML. It promotes a set of best practices, namely that development should be organised in short timeboxed iterations and that it should be adaptive to accommodate inevitable change. A commercial version of the UP, the Rational Unified Process (RUP) (Krutchen, 1999), is its most well-known implementation although there are many others around. RUP was developed by Rational, which was acquired by IBM in 2003. We will, from here onwards, be talking about the UP but the characteristics of the process that are mentioned are also present in the RUP.

Timeboxing means that a (usually) short fixed period – for example, three to four weeks – is devoted to each iteration. Consequently, at each iteration only a small set of requirements is considered and progressed to the implementation and testing stages. Each iteration results in a working but possibly not yet complete system that normally delivers an increment of functionality on the previous incomplete system. Typically many iterations with progressive integration of increments are required before the product can be delivered.

Being adaptive means that adjustments are allowed at each iteration. The motivation for this is the recognition that requirements may change throughout development, and that such changes should be welcome rather than resisted. By involving customers and users at each iteration, feedback can be gained quickly and the required adjustments made within the next iteration. So each iteration may provide an increment over the previous one, or simply revisit its output.

Other best practices promoted by UP are:

- dealing with high-risk issues in early iterations
- prioritising the user's perspective by involving users in requirements, evaluation and feedback
- building a view of the system's architecture in early iterations.

A UP project is organised into four major phases:

1. **Inception**. The business case is developed, together with an idea of the scope of the system and a rough estimate of the effort required.
2. **Elaboration**. The core of the system is developed in an iterative fashion. In this phase all major risks are addressed and resolved, most of the requirements are identified and dealt with, and a more realistic estimate of the effort required is made.
3. **Construction**. The final product is constructed, including the remaining lower-risk and easier elements of the system, again in an iterative fashion.
4. **Transition**. This includes beta testing and deploying the system.

Within the UP phases, development work is organised within many *disciplines*, the UP term for development activities such as requirements, analysis, design and testing. An example of disciplines and their relationship to UP phases is shown in Figure 9.

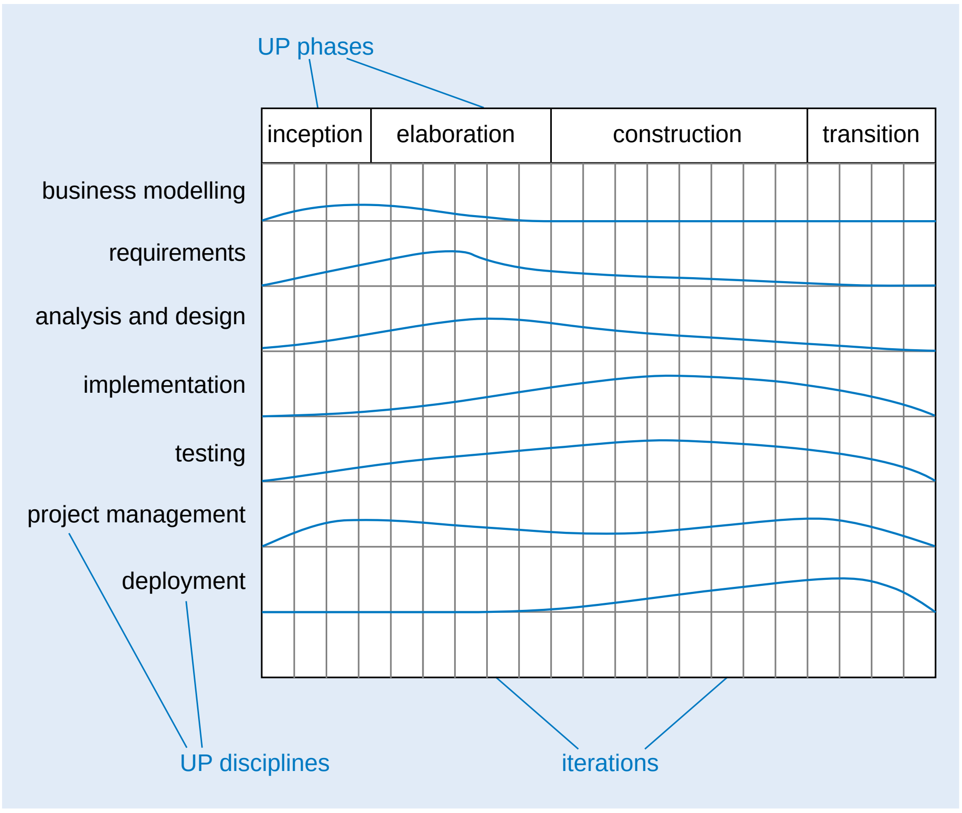


Figure 9 UP phases and disciplines

The figure illustrates what a UP project might look like. The columns represent iterations. For each discipline, the relative effort is represented throughout the UP phases and their iterations. For instance, most of the domain modelling (referred to as **business modelling** in the UP) occurs in the early iterations of the inception and elaboration phases, while most of the implementation occurs within the construction phase.

Views in the UP

The UP was developed by the same people who originally specified UML, and UML is its modelling language. A system's architecture includes models that address five different views:

- *The use case view* contains the basic scenarios that describe the users and the tasks that they need to perform with the aid of a software system.
- *The logical view* is concerned with the functional requirements of the software system. What should the software do for its intended users? Typically this involves the construction models that represent the main elements of a system and how they interact.
- *The implementation view* is concerned with the organisation of the code modules that comprise a software system. Typically it addresses the management of source code, data files and executables.
- *The deployment view* is concerned with the relationship between the various executables (and other run-time components) and their intended computer systems.
- *The process view* is concerned with aspects of concurrency. What are the processes and threads? How do they interact? It deals with such things as response time, deadlock and fault tolerance.

Figure 10 shows that the central use case view relates to the other four views.

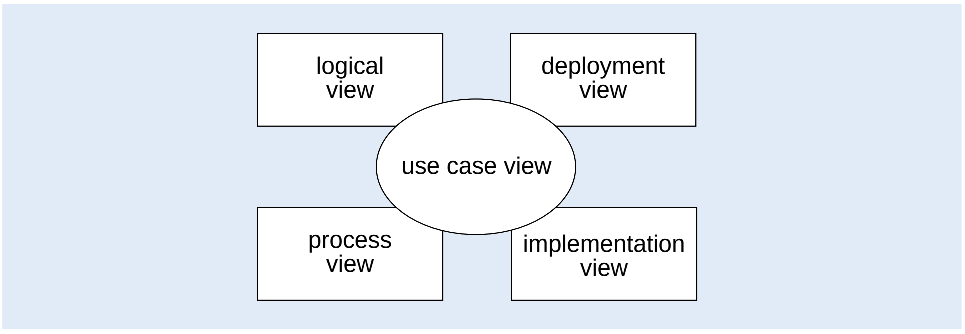


Figure 10 Five views of a software system's architecture

Agile UP

The UP was never intended to be a heavy process and some of its practices are also promoted by agile developers, notably iterative and incremental development, time-boxing, dealing with requirements change and being itself an adaptive process. It is possible to apply the UP in an agile manner and it is often the context in which it is used – the will and experience of developers will dictate whether an agile development takes place.

3.4 Activities and artefacts in the development process

The techniques used in the UP are also in other iterative and incremental processes based on an object-oriented approach to software.

Here, we identify the core activities to develop software. This will give you an initial understanding of how the activities you will learn about contribute to the overall development process. As mentioned earlier, different ways of subdividing the development process and different terminology may be used elsewhere.

Our core activities follow the UP disciplines:

- domain modelling – modelling what already exists in the domain
- requirements – identifying, categorising, prioritising and modelling what the system must do
- analysis – modelling how the structure and behaviour of the system will meet its specification from a user's perspective, moving from the domain to a software solution
- design – deciding on the distribution of responsibilities to fulfil that specification
- implementation – producing code that will meet the user requirements
- testing – ensuring that the software does meet its requirements
- deployment – configuring the code to give a runnable system.

It therefore makes sense to talk of models corresponding to each of these activities. So we have a domain model, a requirements model, an analysis model and so on. However these models generally need to cover a number of views of the system. For example in the previous subsection we talked about structural and behaviour models. A domain model, for example, will usually include both of these. A design model typically consists of a number of more specific models.

There will be other documents that accompany the models, and so we will use the term artefact to refer to both models and other documents. The artefacts from the activities earlier in the list above feed into the activities that follow.

Domain modelling

Our starting point will be to document and model the structure and processes of the organisation's business. Starting from some description of the problem, the initial problem statement, you will learn how to identify elements of the real-world problem and their properties, and build corresponding structural and dynamic models. The emphasis at this stage is on understanding the current domain situation.

The behaviour model provides descriptions of **business processes** and behavioural aspects of the domain. You will meet modelling techniques such as activity diagrams. **Business rules** are used to express constraints on the dynamic model. For example, given a behaviour model describing aspects of a library, there might be a business rule determining the limit on the number of books an individual member can borrow. A structural domain model, also called the **conceptual model**, describes the significant concepts in the domain and how they are related.

A **glossary** of relevant terms and definitions is also produced. This textual document grows throughout the duration of the project.

Some artefacts involved in domain modelling (as well as some used in other phases) are shown in Figure 11.

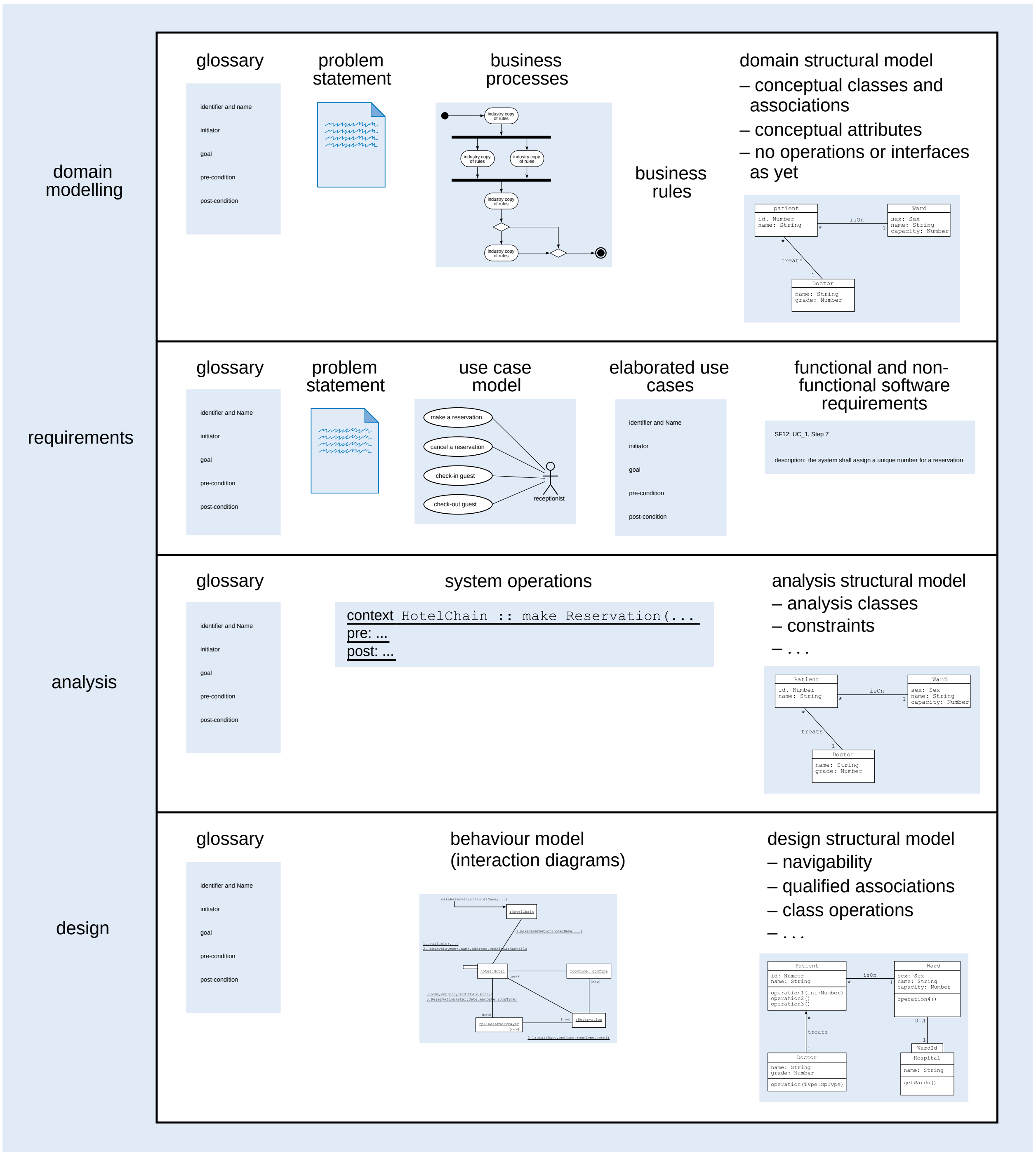


Figure 11 Some development artefacts

Activity 14 Domain modelling

Allow approximately 15 minutes.

(a) What is the purpose of domain modelling?

(b) What is the role of each of the artefacts produced during domain modelling?

Answer

(a) Domain modelling is concerned with gaining an understanding of the environment in which any system that is designed must operate.

(b) During domain modelling, we produce the following artefacts:

- initial problem statement – a description of the problem
- behaviour model – a description of the business processes and behaviour of the domain
- business rules – constraints on the way the behaviour model operates
- glossary – definitions of relevant terms
- structural domain model – an initial structural model representing the concepts relevant to the domain.

Requirements

The next step is to gather and document the requirements for your system and to model what the system is intended to do. Requirements are the expression of the things the system must do or the qualities the system must have in order to meet the stakeholders' needs. Starting from some description of the problem, you will learn how to systematically record requirements information.

Some artefacts involved in requirements are shown in Figure 11.

Activity 15 Requirements

Allow approximately 15 minutes.

What is the purpose of the requirements phase?

Answer

The requirements phase is concerned with establishing and modelling what a software system must do.

Analysis

During analysis, with both use cases and software requirements, you start looking at a system to be implemented, and build both structural and behaviour models. The structural model evolves from that of the domain – it no longer represents concepts from the domain, but rather entities in a software solution. This is usually called the **analysis model**. Use cases and software requirements lead to the specification of what is expected from the system from the user perspective. The **system operations** show how this behaviour can be carried out by the entities chosen.

It is at this stage also that architectural decisions are taken in terms of the overall structure of the system.

Some artefacts involved in analysis are shown in Figure 11.

Design

During design, your goal will be to decide how the expected functionality is to be allocated to each part of the system. You need to make choices about which classes the system operations should be allocated to. These decisions can be explored and documented using further behaviour models. UML uses **interaction (sequence and communication) diagrams** that show a set of classes and the messages between them.

During design both structural and behaviour models are elaborated. In fact you will see that behaviour models are used for both external and internal behaviour of objects.

Some artefacts involved in design are shown in Figure 11.

Activity 16 Analysis and design

Allow approximately 15 minutes.

(a) What is the purpose of analysis?

(b) What is the purpose of design?

(c) What is the role of each of the artefacts produced during design?

Answer

(a) Analysis starts modelling the structure and behaviour of a software solution from a user's perspective.

(b) Design is concerned with making decisions concerning how a system will meet its specification.

(c) During design you produce the following artefacts:

- structural model, an updated version of the one produced during analysis but with its operations specified
- behavioural models, showing how objects in the system will interact and behave internally, and also how functionality will be distributed across the system.

Implementation

The **implementation** model is a description of the assembled components that comprise the working version of the software. It describes how classes are packaged together, and shows some of the relationships between such packages of classes. These relationships are modelled using a component diagram.

Testing

The implementation needs to be tested against its requirements to ensure that it meets them. Tests will be drawn up based on the requirements, and held in a test document. Notice therefore that tests can be drawn up as soon as you know the requirements.

In fact there is a trend towards test-driven design. In this approach, the tests for a system are drawn up before design and automated testing procedures are set up. An initially empty implementation can be run against the tests and the subsequent failure can be used to drive design. As time goes on, this process is repeated as the implementation is built up, until no tests fail. This is the approach followed by some agile approaches such as XP.

Deployment

Many significant computer systems will consist of a variety of software components located on a number of machines communicating via a variety of hardware and software mechanisms. A deployment model records how various components are to be mapped onto different machines and how they will communicate. This can be represented using a deployment diagram.

Activity 17 Modelling languages

Allow approximately 20 minutes.

Suppose you and a colleague used UML on a particular development project. What would be the consequences of introducing symbols or notations that were not defined in UML?

Answer

Such an extension would incur costs as well as offering potential benefits. The costs include the need to communicate the syntax and semantics of your new symbols or notation to other developers. The benefits include the ability to express concepts specific to your project that could be more meaningful in your context.

In the past, there have been many attempts to standardise modelling notations and languages. How might the choice of modelling language affect the maintenance of a software system?

Answer

Every time a new standard notation or language is defined, it allows new software systems to be built using that standard. It can only help with existing software retrospectively. For example, a developer might use the new standard to express a proposed change to the software.

There is no guarantee that an existing software system used any modelling language at all, let alone any agreed standard for one. Even if a standard was used, new staff arriving on the team for the purpose of maintenance might not know about it. Each variety of standard would affect the organisation of a component library. For example, if components using a particular standard were to be put in a library, the descriptions would be likely to require some maintenance as that standard changes over time.

3.5 Summary of Section 3

As you become more familiar with the activity of modelling, it should become apparent that there is considerable flexibility in the construction of models. There is no guarantee that different developers, when confronted by the same problem, will select the same things to model, and even if they do, will produce identical models, although there should be a great deal of similarity in most cases.

This section considered the role of modelling in the development of a good software system:

- A model is an abstract representation of a concept, a specification, a design or a system from a particular point of view of either the user or the developer. In general, it is a simplification that is used to understand an aspect of the system in question.
- A standard modelling language helps those involved in software development projects to communicate effectively. If the standard is widely used, it will reduce the time taken for developers to become familiar with a new project.
- The Unified Modeling Language (UML) is a useful standard because it is easy to use, sufficiently expressive, unambiguous and widely used. There are also a variety of tools that support UML.
- During software development, there may be many models made from a variety of viewpoints. They must not contradict each other, that is, they must be consistent.

The UP is a popular iterative and incremental development process. It defines a series of timeboxed iterations and promotes a set of best practices that are high-risk driven, user centred and architecture focused. It is possible to take an agile approach to the UP as many of its practices are also supported by agile developers.

Conclusion

In this free course, *Approaches to software development*, you have extended your knowledge of the important ideas in software development. We investigated the characteristics of a good software system, and considered what a development process would need to include to build such software. You saw that there is no single development process to suit the variety of users' requirements. However there must be a disciplined approach to software development, especially in the case of large projects.

We introduced the notion that it is good practice to split a project into smaller, more manageable activities. When developing good software systems, you should focus on the users' needs and, wherever possible, make use of replaceable and reusable modules – components. The overall software architecture should be constructed around the users' requirements.

We then introduced the role of modelling in the development of software. In particular, the concepts of object orientation allow us to represent users' requirements in a way that reflects our natural tendency to view the world around us in terms of objects. The way we relate the various activities of software development and associated artefacts (including models) was then described.

This OpenLearn course is an adapted extract from the Open University course TM354 *Software engineering*.

References

Ambler, S. (2002) *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process* [Online], New York, John Wiley & Sons. Available at http://libezproxy.

open.ac.uk/login?url=http://open.eblib.com/patron/FullRecord.aspx?p=131031 (Accessed 6 November 2015).

Bass, D.L., Clements, D.P. and Kazman, D.R. (2012) *Software Architecture in Practice*, 3rd edn, Upper Saddle River, NJ, Addison Wesley [Online]. Available at http://libezproxy.open.ac.uk/login?url=http://proquestcombo.safaribooksonline.com/book/software-engineering-and-development/9780132942799 (Accessed 6 November 2015).

Beck, K. (2004) *Extreme Programming Explained: Embrace Change*, Upper Saddle River, NJ, Addison Wesley [Online]. Available at www.open.ac.uk/libraryservices/resource/ebook:0321278658/provider/ProQuest_Safari_Tech_Books_Online (Accessed 6 November 2015).

Boehm, B. (1988) ‘A spiral model of software development and enhancement’, *Computer*, vol. 21, no. 5, pp. 61–72 [Online]. Available at http://ieeexplore.ieee.org.libezproxy.open.ac.uk/xpls/abs_all.jsp?arnumber=59 (Accessed 6 November 2015).

IEEE (1990) *610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*, IEEE [Online]. Available at http://libezproxy.open.ac.uk/login?url=http://ieeexplore.ieee.org/servlet/opac?punumber=2238 (Accessed 6 November 2015).

Jacobson, I., Booch, G. and Rumbaugh, J. (1999) *The Unified Software Development Process*, Upper Saddle River, NJ, Addison Wesley.

Krutchen, P. (1999) *The Rational Unified Process: An Introduction*, Upper Saddle River, NJ, Addison Wesley.

Larman, C. (2005) *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and iterative Development*, Pearson [Online]. Available at http://proquestcombo.safaribooksonline.com.libezproxy.open.ac.uk/book/software-engineering-and-development/agile-development/0131111558 (Accessed 6 November 2015).

Leveson, N. and Turner, C. (1993) ‘An investigation of the Therac-25 accidents’, *Computer*, July, pp. 18–41 [Online]. Available at http://ieeexplore.ieee.org.libezproxy.open.ac.uk/stamp/stamp.jsp?tp=&arnumber=274940&isnumber=6812 (Accessed 6 November 2015).

Lions, J.L. (1996) *ARIANE 5 Flight 501 Failure, Report by the Inquiry Board* [Online]. Available at www.di.unito.it/~damiani/ariane5rep.html (Accessed 6 November 2015).

Mašek, K., Hnetynka, P. and Bureš, T. (2009) ‘Bridging the component-based and service-oriented worlds’, *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, Patras, August 27–29 2009. IEEE, pp. 47–54 [Online]. Available at http://ieeexplore.ieee.org.libezproxy.open.ac.uk/stamp/stamp.jsp?tp=&arnumber=5349894&isnumber=5349835 (Accessed 6 November 2015).

National Audit Office (2012) *Governance for Agile Delivery* [Online]. Available at www.nao.org.uk/publications/1213/governance_for_agile_delivery.aspx (Accessed 6 November 2015).

South West Thames Regional Health Authority (1993) *Report of the Inquiry into the London Ambulance Service* [Online]. Available at www0.cs.ucl.ac.uk/staff/a.finkelstein/las/lascase0.9.pdf (Accessed 6 November 2015).

Schwaber, K. and Sutherland, J. (2011) *The Scrum Guide* [Online]. Available at http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf (Accessed 24 May 2018).

Acknowledgements

Acknowledgements

This free course was written by Leonor Barroca.

Except for third party materials and otherwise stated (see terms and conditions), this content is made available under a
Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Licence.

The material acknowledged below is Proprietary and used under licence (not subject to Creative Commons Licence). Grateful acknowledgement is made to the following sources for permission to reproduce material in this free course:

Course image: © istock.com/duncan1890

Figure 1 © The Open University

Figure 2 © The Open University

Figure 3 © The Open University

Figure 4 © The Open University

Figure 5 © The Open University

Figure 6 © Larman C., (2005) 'Applying UML and Patterns' 3rd Edition, Prentice Hall Inc.

Figure 7 © Ward Cunningham et al (2001) http://agilemanifesto.org/

Figure 8 © The Open University

Figure 9 © The Open University

Figure 10 © The Open University

Figure 11 © The Open University

Section 2.3 SE-Radio, "Episode 188: Requirements in Agile Projects", 12 September 2012, http://creativecommons.org/licenses/by-nc-nd/2.5/

Every effort has been made to contact copyright owners. If any have been inadvertently overlooked, the publishers will be pleased to make the necessary arrangements at the first opportunity.