The Open University

M814 Block 2 Unit 7

M814 Software engineering

Unit 7: Software processes

This publication forms part of the Open University module M814 *Software engineering*. Details of this and other Open University modules can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes MK7 6BJ, United Kingdom (tel. +44 (0)845 300 60 90; email general-enquiries@open.ac.uk).

Alternatively, you may visit the Open University website at www.open.ac.uk where you can learn more about the wide range of modules and packs offered at all levels by The Open University.

To purchase a selection of Open University materials visit www.ouw.co.uk, or contact Open University Worldwide, Walton Hall, Milton Keynes MK7 6AA, United Kingdom for a catalogue (tel. +44 (0)1908 858779; fax +44 (0)1908 858787; email ouw-customer-services@open.ac.uk).

The Open University,
Walton Hall, Milton Keynes
MK7 6AA

First published 2014.

Copyright © 2014 The Open University

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher or a licence from the Copyright Licensing Agency Ltd. Details of such licences (for reprographic reproduction) may be obtained from the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS (website www.cla.co.uk).

Open University materials may also be made available in electronic formats for use by students of the University. All rights, including copyright and related rights and database rights, in electronic materials and their contents are owned by or licensed to The Open University, or otherwise used by The Open University as permitted by applicable law.

In using electronic materials and their contents you agree that your use will be solely for the purposes of following an Open University course of study or otherwise as licensed by The Open University or its assigns.

Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of The Open University or in accordance with the Copyright, Designs and Patents Act 1988.

Edited and designed by The Open University.

WEB038433

1.1

Contents

Unit 7: Software processes

The sequential development process used as an example in Unit 6 has been widely followed as the ‘waterfall model’. This has been adopted and adapted to create many alternative software development processes. However, in spite of being highly prescriptive and thus hopefully predictable, these methods actually prove difficult to use in building software that is effective, useful and completed within budget and timescale.

In this unit we look at a range of software development approaches that have arisen as alternatives and that aim to overcome the acknowledged shortfalls of sequential methods. Having completed your study of this unit, you will be able to understand and explain:

- alternative sequential and incremental methods
- iterative, prototyping and evolutionary methods that explore possible solutions, aiming to converge on an acceptable solution
- participative methods that involve the intended users and beneficiaries
- timeboxing methods that fix the cost and timescale, and vary the functionality delivered
- design-driven methods that build on known solutions to solve new problems
- agile methods that focus on the software developers and the code they produce
- open-source methods that involve many developers in a loosely controlled, yet effective process.

Each of the methods listed has particular strengths and weaknesses and it is up to the software development team to adopt and, if needed, adapt the most appropriate methods for the project. In this unit we consider the particular strengths of each process model and discuss what new problems each creates in turn.

7.1 Introduction

In managing software activities in an organisation, you may only occasionally be involved in overseeing the development of completely

new software, but you'll almost certainly be involved in procuring software via one or more of the routes described in Unit 6, and then integrating it with other software and the organisation's processes. To do all of this, you need to know how software is usually developed so that you can appreciate claims being made by software suppliers, and can manage integration of your software once acquired. This unit aims to give you that understanding.

In Unit 6 you encountered a range of activities that typically need to be undertaken when developing software or acquiring and adapting software for operational use. We looked at these activities in the context of developing a simple hypothetical library management system for a school, progressing through each activity in sequence. Let's start by looking at those typical activities, focusing on their essence in order to generalise them. Then in the rest of this unit, we'll consider alternative sequences in which these activities could be undertaken in the process of developing successful software. The activities were as follows.

- **Requirements elicitation**, to understand what the stakeholders' problems were that might need a software solution. While this must always be done, it could be that true understanding of what's needed only emerges once the system is in operation.
- **Requirements specification**, to give a precise description of the stakeholders' problems, ready for software developers to start building the new system. These activities will be described in more detail in Block 4.
- **Estimation** of the amount of effort needed to undertake the work required. You saw some illustrations of what was needed in Unit 6, and all this will be described in more detail in Unit 8.
- **Architectural design** to identify the major elements of the system and how they interact. Such design needs to be undertaken for any system that will be composed of more than one element – the parts may constitute human procedures as well as software, and may be purpose built or bought in from outside; and once the parts have been acquired they will need to be integrated.
- **Project planning and control** to establish a sequence in which the work will be done and to ensure that it does get done. You'll learn more about this in Unit 8.
- **Detailed design**, since any component that needs to be developed,

whether software or procedural, needs to be designed – that is, thought about abstractly in terms of its constituent parts and how they interact. The design may be documented explicitly, or may be implicit in the program code and as tacit knowledge on the developers' part.

- **Module implementation or coding**, involving writing the program in some executable code. This includes designing the module initially and testing it afterwards.
- **System integration**, since the various parts need to be made to work together. This is always necessary, and how easy it is depends on the software architecture and the extent to which the components actually do what was specified by the architecture. Inevitably, one aspect of this activity is testing that the components connect and work together – hence the alternative name, 'integration testing'.
- **System testing**, involving the technical testing of a system or major component with respect to its specification, if any. Even if no specification is available, standard sources of failure can be explored.
- **Acceptance testing**, any contract − for work to be done or goods to be supplied (as described in Unit 4) − will conclude with an activity which formally accepts the goods or services as complete and discharges the contract. For software this is usually some trial use that represents operational use.

The activities of architectural design, detailed design, module implementation and system integration are covered in greater detail in the companion module M813 *Software development*.

7.1.1 Software process models

In section 7.2 you'll encounter a number of established methods that advocate the simple linear sequential execution of these activities; you'll also see how this linear sequence creates problems. You'll then look at a number of these problems and, from these, motivate other sequences. The sequence in which the activities are conducted, and the rationale underlying that sequence, is known as a **life-cycle model** or **process model**.

Most software development processes consist of a number of activities being done in some prescribed sequence.

Each activity produces a number of work products. These might be given formal status as deliverables, and the production of each becomes a milestone that marks progress through the development process. You'll see more of this in Unit 8. Each deliverable will be subject to some review or inspection that confirms whether the deliverable has been produced to an acceptable quality. This may lead to changes and quality improvements before final, formal acceptance.

The testing and review activities form part of a larger set of activities known as **verification** and **validation** (V&V). Verification activities focus on the technical correctness of the software, while validation activities focus on ensuring that the system being built achieves the purpose intended. You'll see that validation concerns are often what motivate our different process models. Verification and validation in turn are often treated as elements of a larger quality management system – covered in depth in Unit 10.

The work products may be documents, software or other items. Documents should conform to a standard which defines what should be covered and what notations should be used. The notations help to document the aspect of the software being described at that point – we commonly refer to this description as a model.

The models and notations will also conform to standards that cover all the models to be developed during the software development process. We've been following a simple convention in this module, using data-flow and data-structure diagrams conforming to the 'standard' set out in Appendix A: Modelling notations. Usually the set of notations used is much more comprehensive – examples of full data-flow and data-modelling notations are included in **SSADM** (Structured Systems Analysis and Design Method). The development of SSADM was supported by the UK government and for some time was mandatory for UK government IT projects, along with the project management method PRINCE. This was developed during the 1980s and stabilised in its final 4.2 version in 1995, and is now known as 'Business System Development'. During the 1990s a number of proprietary methods arose

to support the development of object-oriented software, and in the late 1990s the **Object Management Group (OMG)** adopted Unified Modelling Language (UML), which standardises the modelling notations. UML does not prescribe any particular way of using the models it standardises – this is done in the Rational Unified Process and to a lesser extent in OMG's Model-Driven Architecture, both of which are described later in this unit.

Models using these notations are very likely to be produced using a software tool which also stores them in a repository. These tools are commonly known as **Integrated Development Environments (IDEs)**, although in the past they have been known by the rather more all-embracing **Integrated Project Support Environments (IPSEs)**. These IDEs and IPSEs are very likely to not just support modelling using these notations, but also to impose some process model concerned with the sequence in which the particular models are produced.

> All work products should conform to some standard and include models using some standardised notations.

7.2 Classic process models: sequential and incremental

The simplest life-cycle process models view the activities described above as taking place in their natural order, followed by a process of evolution or maintenance.

When drawn in a process diagram, as in Figure 7.1, this process model is known as the waterfall life-cycle. This model encapsulates the standard practices that arose in the early days of software development from the 1950s to the 1970s, matching the views of engineering industries that were increasingly involved in software as an important component of their engineering. This waterfall life-cycle is often attributed to Royce (1970), although some dispute this because this sequential model had long been part of tacit knowledge among engineers.

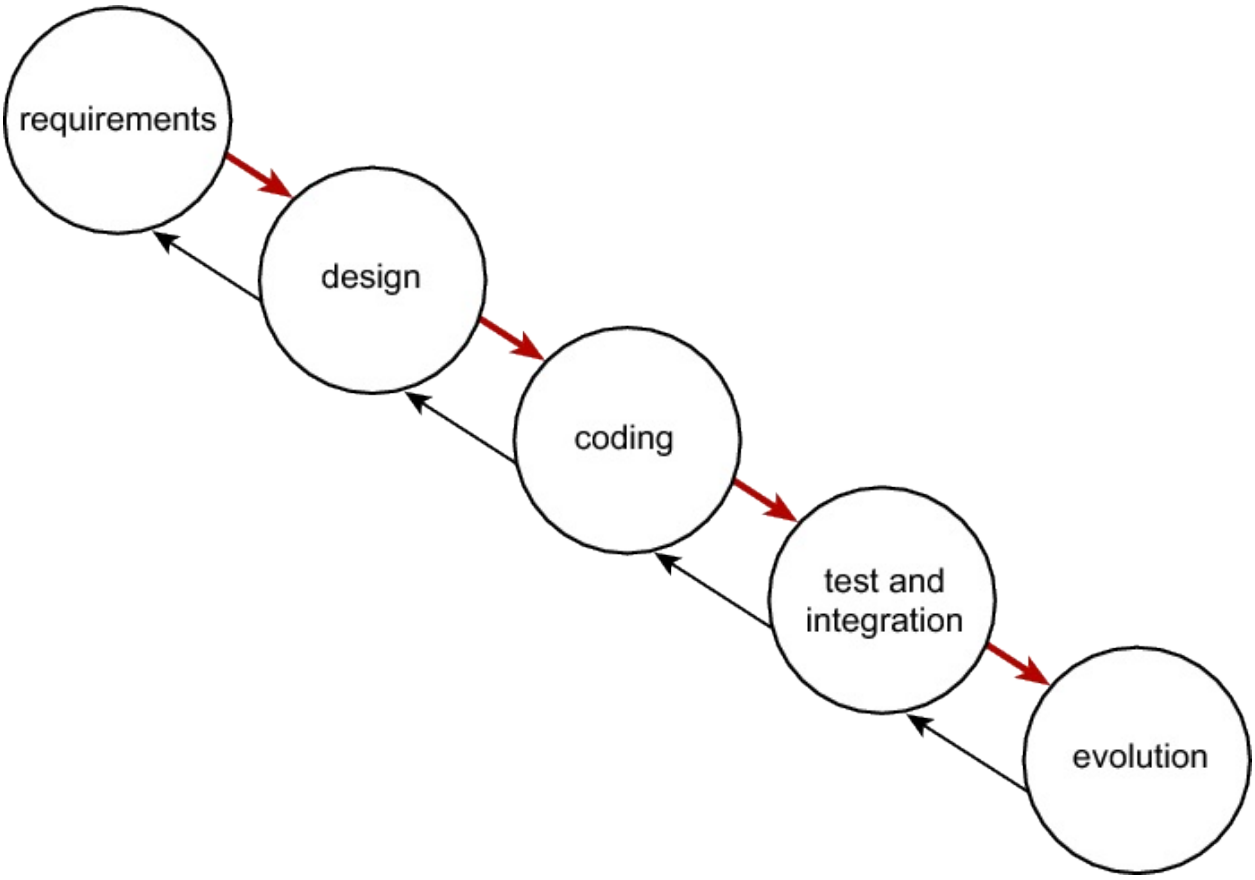


Figure 7.1 The waterfall life-cycle

View description

The main flow (waterfall) of activity is from top left to bottom right, shown using the heavy red arrows. The deliverables flow along these arrows between activities. However, we must recognise that in any process like this there will be some iteration or feedback: as deliverables get reviewed and then used, some reference back to the earlier activity and some reworking will be necessary. This is shown by the thinner black arrows.

An early variant on this waterfall life-cycle was the recognition of parallelism in the development process – particularly the parallel development of database and functional elements, as shown in Figure 7.2. The importance of the user interface was not initially appreciated, but as soon as it was, the parallel development of such interfaces was included.

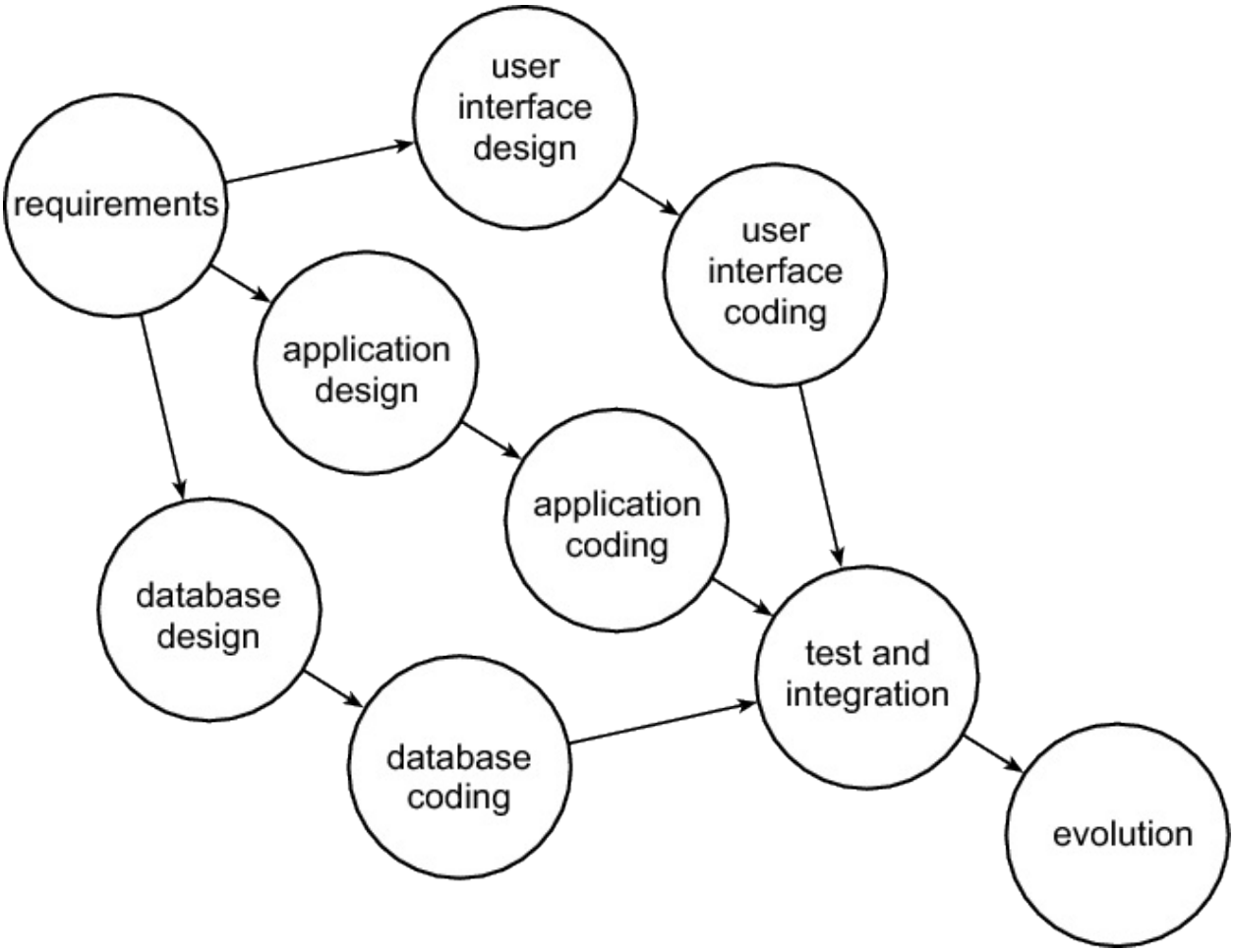


Figure 7.2 The parallel development of database and function and the user interface

View description

Figure 7.2 simply shows the forward flow of the work products. As before there will be some backward flow as reworking occurs, and also some lateral flow between activities when the design or coding of a major component requires agreement with the designers and coders of another component. Note that a small amount of architectural design will have occurred during the requirements activity, sufficient at least to have identified the three architectural components for the application: its functions, its user interface and its database.

These sequential methods clearly do work for relatively simple systems, such as the library system used as the case study in Unit 6. However, as systems have grown in size and complexity, and have been aimed at supporting less routine aspects of an enterprise’s activities, difficulties have arisen.

Sequential process models work well for small problems.

7.2.1 A sequential model: the Rational Unified Process

Case study 7.1 focuses on a very popular method, the **Rational Unified Process (RUP)**. The RUP is an example of an **object-oriented method**. When object-oriented programming became popular in the 1980s through programming languages like Smalltalk, C++ and, later, Java, there were several attempts to modify earlier methods for use with OO programming. What characterised many of these early OO methods was their shift of focus away from any consideration of the total process, within which a part would be automated, towards a focus primarily on the software system and its interaction with the external world. The RUP was developed by the Rational Software Corporation, now part of IBM, but the Object Management Group has also developed its own approach to using UML in the initial development of OO systems: **Model-Driven Architecture (MDA)**.

Case study 7.1: The Rational Unified Process

Object-oriented methods rose to prominence during the 1980s, with many leading consultants advocating their own processes for doing these. The Rational Software Corporation was founded in 1981, and over the next four years recruited three prominent experts of the time: James Rumbaugh, Ivar Jacobson and Grady Booch. Together these ‘three amigos’ developed first the Unified Modeling Language (UML) for the representation of object-oriented software designs, then the Rational Unified Process (RUP) to prescribe how software should be developed using UML. The RUP has since its inception been articulated as a framework which can be adapted to a particular project, in which iteration is an important feature (iterative development is described later in this unit).

7.2.2 The limits of sequential methodologies

A number of process-model developments were essentially variations on the basic waterfall model. The next step was the recognition that in practice, not every aspect of the system needs to be delivered at the same time. It had become quite common to deliver a succession of releases of the system, with each release containing more functions, while also correcting or modifying the functions of earlier releases. This was formalised in the incremental process model shown in Figure 7.3. Activity 7.1 explores some of the limitations of sequential process models.

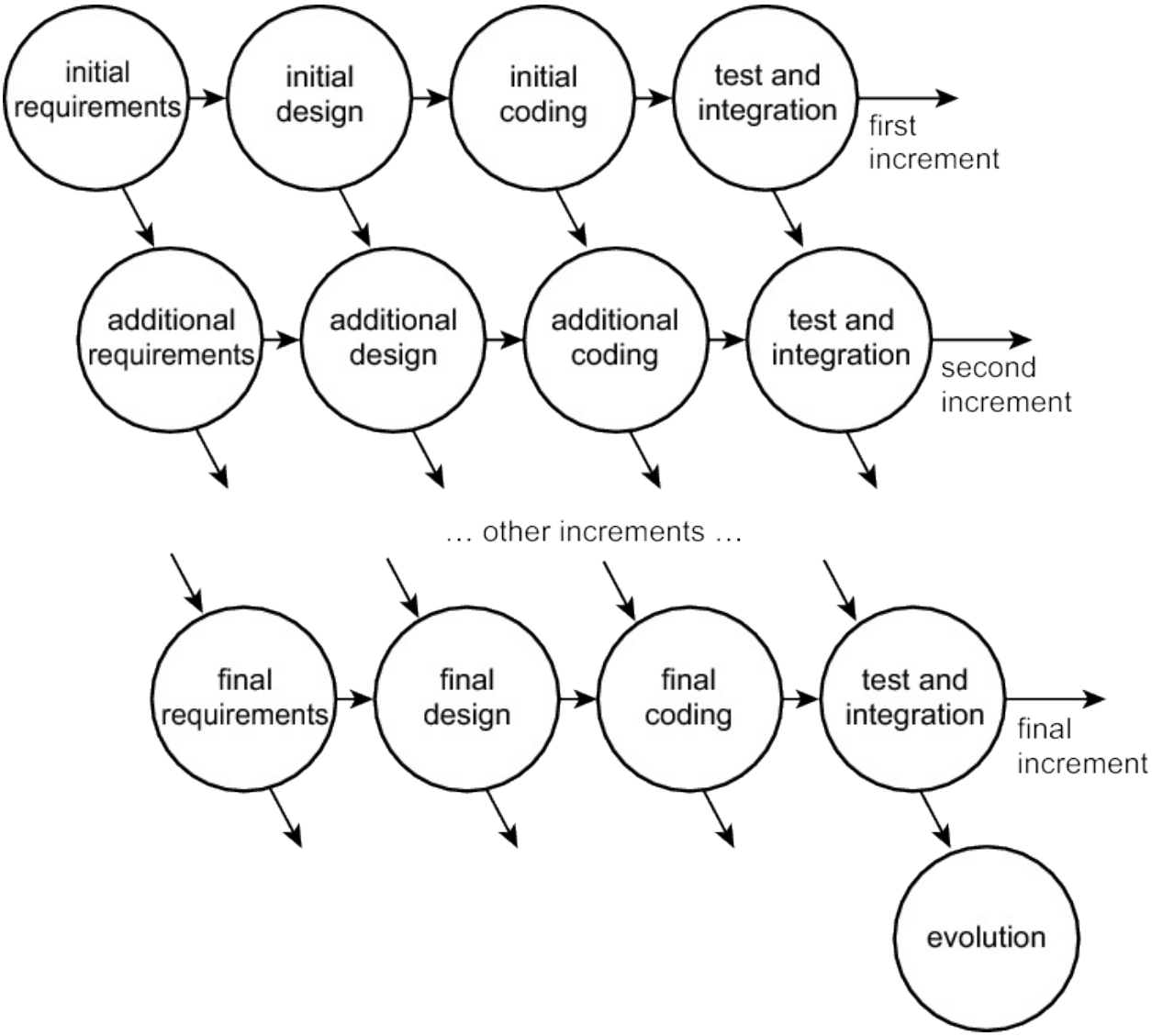


Figure 7.3 An incremental process model

View description

Activity 7.1: The limits of sequential methodologies

The methodologies of the 1970s and 1980s were characterised by large prescriptions of required practices: what activities should be done and when, and what should be produced, including the notations to be used. Large multivolume methodology manuals were produced, and expensive tools and training courses were purchased in the hope that, through following the prescriptions, good quality software would be produced at affordable cost and to a predictable timescale. It was hoped that, through the sheer power of rational planning and systematic work practices, the difficulties encountered in projects could be overcome.

As the problems being tackled get ever larger and more complex, what would you expect to happen with the sequential methodologies described above? Relate your answer to your own experience of the methodologies, as well as to the discussion on the limits of rationality in Units 1 and 2 (throughout this unit the terms ‘method’ and ‘methodology’ are used as synonyms).

View discussion

Sequential process models and the methodologies built upon these have reached the limits of rationality, and are not able to handle large and complex projects properly.

7.3 Resolving uncertainties

In sequential approaches to software development it is assumed that it is possible to identify system requirements in full beforehand, and then to use this identification as the foundation for further development, i.e. the design, coding and testing. However, things may not actually work out like that – it may not be clear at the start what the needs of the

organisation are, such needs may change with the passage of time, or new technologies may offer new opportunities. Furthermore, what the organisation really needs may not be clear until after the system has been built: the use of the new system may suggest new possibilities. We need an approach that is more exploratory, that does not demand that immutable requirements are laid down at the start. Iterative and **evolutionary** methods are a response to this need. Larman and Basili (2003) provide a history of iterative and incremental development, which provides the basis for discussion of some of the key methods in this unit.

> Most people find it difficult to envisage the way a system will work in practice, simply on the basis of descriptions of that system; the new system needs to be available and in use before the full impact can be assessed.

An early idea was **prototyping**. In order to resolve problems to do with requirements, the idea is to build a prototype of the system, or a part of the system, to act as a focus for discussion and agreement with stakeholders. The prototype may itself be very simple – hardly more than a few screens that might typically be encountered when using the system – but it could be working through incomplete software. As you saw from the overview of the requirements process in Unit 6, such prototypes might be built using a variety of techniques, including simply sketching screens on a whiteboard or on adhesive notes (MRP, Chapter 2). Based on reviews of the initial prototype, further prototypes might be built, until a clear understanding of what is needed and how it would fit into the stakeholders' work has been reached. At the end of the process the prototype may form the 'agreement', or a formal requirements specification may be written – and then development would continue sequentially as before. All this is illustrated in Figure 7.4.

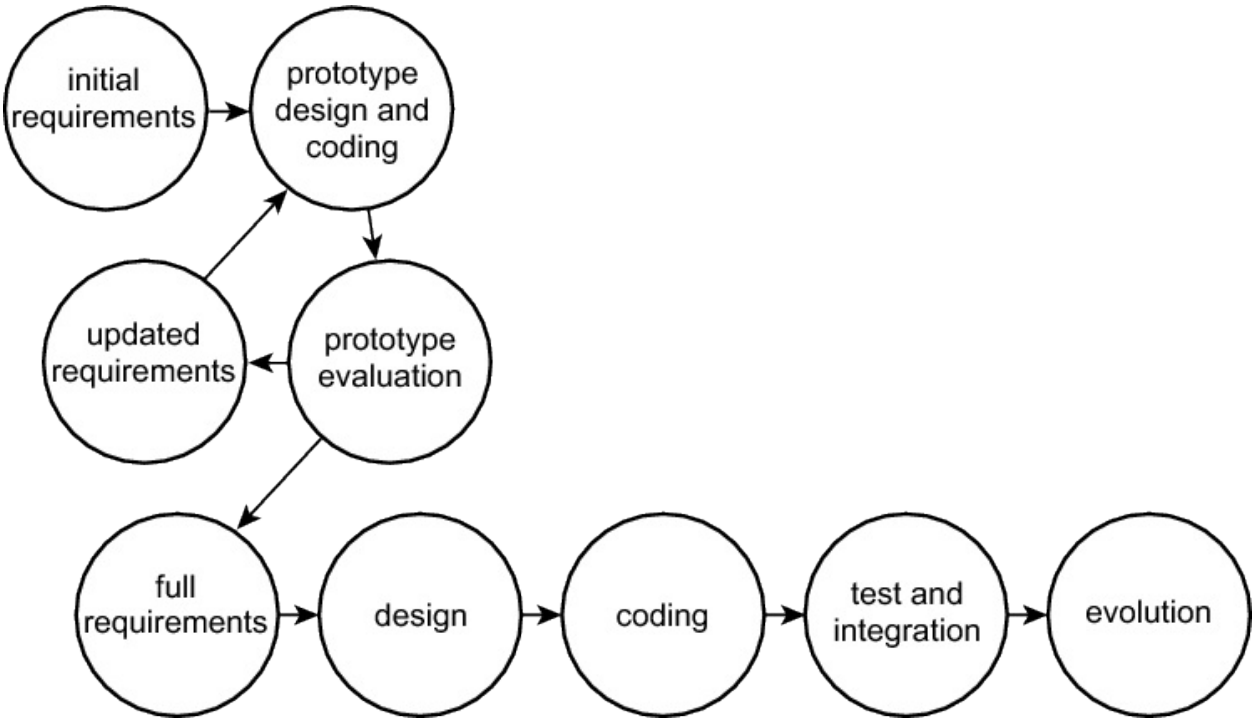


Figure 7.4 The prototyping process model

View description

7.3.1 Prototyping, evolution, participation

Prototyping, used to clarify requirements in this way, still leaves a sequential process at the end. It is also prone to the delays introduced by such a process, as well as the real risk that needs may have changed by the time the real system is delivered. So why not simply use the prototype? Well, this may be difficult – the prototype may not include all the functions, and it certainly will not be fully engineered with the levels of robustness and security required for the final system. But if the basic idea is right, why not let the prototype evolve into the full operational system? This leads us to the **evolutionary process model**, set out in Figure 7.5.

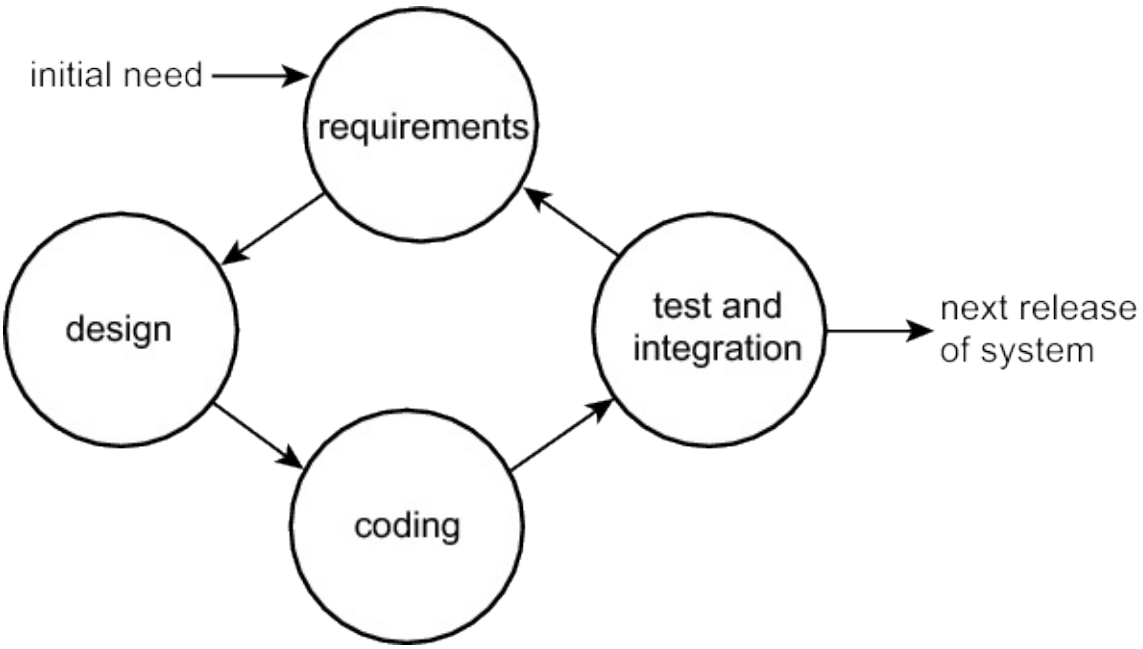


Figure 7.5 The evolutionary or iterative process model

View description

A common practice in prototyping and evolutionary processes, apart from the iteration, is the involvement of users and stakeholders throughout the development process. Initially this was done as part of the test and evaluation activity, but with the increasing adoption of agile methods, involvement occurs in the design and development phases as well. This is clearly important in order to address the risk of misunderstood requirements, and it recognises both that software systems being built are for people, and that software is but part of a socio-technical system. In the wider movement towards the development of **socio-technical systems** – mostly associated with researchers and practitioners in Scandinavia – there has been a marked emphasis on worker participation in the design and development of systems intended to support their work. Not only would stakeholders be involved in testing and reviewing, but they would also be involved in other aspects of development – joining the team and bringing an essential user perspective. This approach was very popular during the 1980s, and has come to be known as **participative design** or **participative development (PD)**.

Now complete Activity 7.2.

Activity 7.2: Motivation in prototyping and participation

Drawing on the knowledge of human motivation you gained in Unit 3, describe how you would expect software engineers to react to prototype development processes and the participation of users in those processes. What have you observed, or what do you think would happen, in practice?

View discussion

Involve users in a participative process to reduce requirements risks.

The evolutionary or iterative process model illustrated in Figure 7.5 can be viewed as a quasi-sequential model, in which all the work products are passed from one activity to the next in sequence. This means that if during coding, for example, a change in requirements or design is proposed, this must be passed on to the next iterative cycle. But it's tempting to simply change the code – and then abstract from the code the design and requirement changes that were implicit in the code. This is **reverse engineering**. Modern software development tools can support some aspects of this, facilitating development to move forwards and backwards so as to give a flexible, totally iterative process known as **round-trip software engineering**.

7.3.2 Mitigating risk

When we became concerned about our requirements – that we might not fully capture them at the start of a sequential development process – we were focusing on risk. What if we got the requirements wrong, and built the wrong system? We iterated through the requirements in order to mitigate that risk. But what about other risks – such as those attendant on adopting a new technology, or building a system much larger than any previous one? We also need to explore these. This problem prompted Barry Boehm to propose the **spiral model** of software development (Boehm, 1988). This model has captured the imagination of many software engineers. Rather than yielding a set of prescribed practices,

Boehm’s model generates insight into how software is produced. It’s described in Case study 7.2.

Case study 7.2: Boehm’s spiral model

Figure 7.6 shows the spiral in its original form. Some users may now draw it differently to serve particular purposes, but the essence remains: iteration. Boehm comments in his original paper:

> The major distinguishing feature of the spiral model is that it creates a risk-driven approach to the software process rather than a primarily document-driven or code-driven process.
>
> Boehm (1988, p. 61)

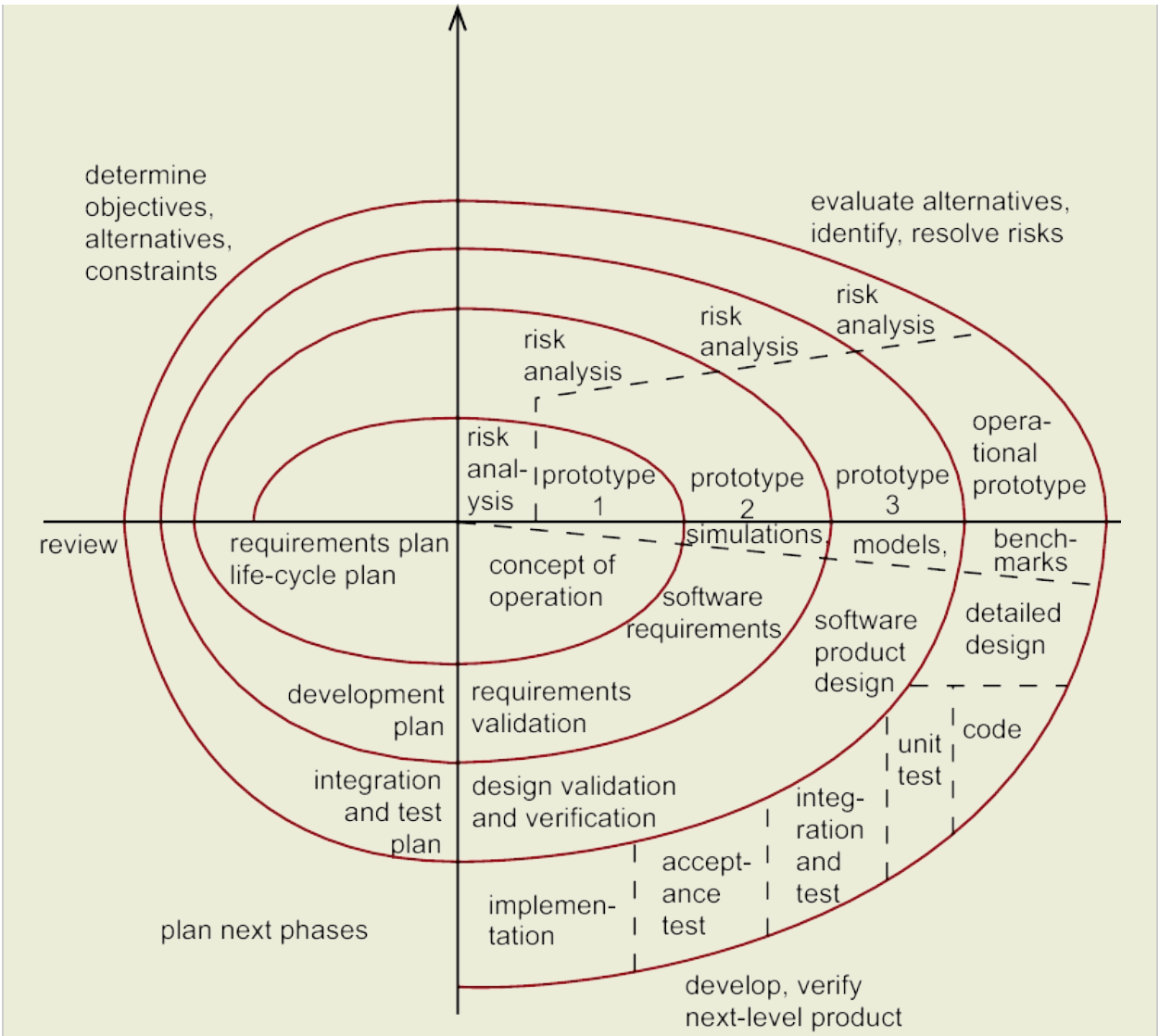


Figure 7.6 Boehm's original spiral model

View description

Boehm calls the approach 'risk-driven' because what the spiral model aims to do is deliver the software's higher-risk components in the earlier iterations. Following Figure 7.6, the project starts in the spiral's centre and progresses clockwise. The radial dimension from the centre outwards represents accumulated cost to date, so each successive cycle of the spiral moves outwards as it adds to the total cost so far.

A typical cycle consists of the following sequence of steps, with roughly one step per quadrant of the cycle:

1. identify the objectives of this cycle, and alternative solutions and any constraints
2. evaluate alternatives, identifying areas of uncertainty and resolving these by appropriate means, and following up any new uncertainties revealed and that need exploration
3. follow normal waterfall-type activities, as seen in Figure 7.1 (in section 7.2 above)
4. end the cycle with a review of everything done in that cycle, aiming to get the commitment of all stakeholders to the next cycle – or to terminating the project.

The next cycle may partition the work to be done, and may have many parallel spirals.

Towards the end of his paper Boehm claims: ‘All of the projects fully using the system have increased their productivity at least 50 per cent’ (Boehm, 1988, p. 69). These days a claim like that would be greeted with scepticism – but at that time, in moving from a chaotic regime to one in which management did think about process and control, making such savings did actually prove possible.

Iterate to mitigate risk, focusing on the highest risks in early iterations.

7.4 Flexibility about functions

If you plan to develop a system iteratively you know you can always defer functions from the current iteration to some later iteration. This means that, for the current iteration, you can focus on the immediate important functions and postpone dealing with those that are less important. In doing this you might be able to fix the delivery schedule, timescales and even the costs, and vary the functions to be delivered. All this can be explained using triangles with vertices of functions, cost and timescale, as shown in Figure 7.7.

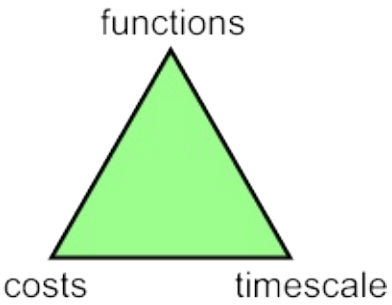


Figure 7.7 The function–cost–timescale triangle

View description

Any software development process must deliver a stated set of functions or requirements at some specified cost and within some specified timescale. However in certain circumstances – for instance, where you have poorly defined projects or chaotic software processes or you lack experience with similar projects – you may not be able to predict accurately either the cost of or the timescale for delivering a system with a given set of functions. In such situations the typical outcome is for the delivery date to be moved on as costs increase, in order to deliver agreed functions. However, focusing on the function–cost–timescale triangle, you may realise that things need not be this way. You could simply have any two of the vertices as fixed, and vary the third vertex to fit. So why not vary the functions and fix the costs and timescales – in other words, deliver what you can by the date agreed and at the costs agreed?

7.4.1 Timeboxing and rapid application development

This leads to the idea of **timeboxing**. Here, you keep elapsed time-to-delivery constant, as well as resources or costs, and vary only what is delivered. Varying the functionality is justified on the basis of the ‘80/20 rule’: it’s said that typically 80 per cent of the benefit from an activity can be gained with 20 per cent of the development effort. Requirements are prioritised into ‘essential’, ‘important’, ‘desirable’ and ‘deferred’. All essential functions must be delivered at the end of the timebox, along with as many important and desirable functions as possible. Timeboxes are relatively short – each just four to six weeks long – and a complete development process model then comprises a series of timeboxes in sequence.

If we lay out each iterative cycle of the iterative process model seen in Figure 7.5, and place a timebox round each one, we get **rapid application development (RAD)**. This has had a significant impact on software development practice, and takes a variety of forms. One significant form is the Dynamic Systems Development Method (DSDM) described in Case study 7.3. The activities that follow provide an opportunity to explore DSDM and RAD in more detail.

Case study 7.3: Dynamic Systems Development Method

The Dynamic Systems Development Method is owned by the DSDM Consortium, which was established in 1994 as a not-for-profit organisation with the aim of developing and promoting an independent rapid-application development framework. DSDM is now described as an agile method (see later in this unit), as emphasised on the DSDM consortium website.

7.4.2 DSDM, timeboxing and the spiral model

Next, Activities 7.3 and 7.4 give you an opportunity to explore the DSDM framework in more depth. To complete these activities you'll need to access the DSDM consortium website.

First, complete Activity 7.3.

Activity 7.3: DSDM principles and techniques

Locate a description of the current DSDM framework, then write a brief summary of the principles and techniques used in DSDM.

Provide your answer...

View discussion

Now complete Activity 7.4.

Activity 7.4: Timeboxing versus the spiral model

The timeboxing and RAD approaches to software development advocate doing the most important features of a system first − invoking the 80/20 rule − whereas the spiral approach advocates doing the highest-risk parts of the system first. How can these two approaches be reconciled?

Provide your answer...

View discussion

If it seems very difficult to produce software to an agreed specification, timescale and cost, you can adjust the set of functions delivered, using timeboxing to facilitate this.

7.5 Design-driven processes

As experience with software has grown, it has become apparent that while most software is unique, it is also similar to other software that's previously been developed and proven in use. How can experience gained with similar systems in the past be built upon when developing a new system? Early attempts at tackling this focused on software reuse and components – the economic and quality advantages of this were considered in Unit 6 – as a motivation for acquiring software by buying part or all of it, rather than building it bespoke. Approaches that focused on general-purpose components have proved unsuccessful – but the more general idea of building on knowledge and experience gained in solving earlier problems has proved fruitful.

It was realised as early as the 1950s that in programming computers you constantly come across the same kinds of problem which you solve in the same kinds of way. Such solutions were so stereotyped that computer

manufacturers would build some of them into the computer hardware itself, while other solutions found their way into early programming languages. However, such an approach only works for simple, repetitive and common situations.

7.5.1 Patterns: model–view–controller

Often what a programmer wants is not exact reuse, but merely reuse of design ideas adapted to a particular situation. Standard ideas in software design are usually passed on as tacit knowledge from expert to less experienced programmers. During the early 1990s people began to realise that such experience could be set down in writing and passed on explicitly, using the ‘patterns’ idea deriving from the work of the architect Christopher Alexander (1977). Patterns, which arose in the object-oriented programming movement, are usually represented using object-oriented notations, usually UML – as in Larman (2002) – which are highly suitable. The model–view–controller (MVC) pattern described in Case study 7.4 arose in the earliest object-oriented programming environment, Smalltalk.

Case study 7.4: The model–view–controller pattern

Suppose you want to build a personal budgeting system that lets users plan how they’ll use their money. This should let the user set a number of budget headings, then under each, record the sum of money he or she is allocating – either the absolute amount or as a percentage of the user’s income. The allocations should be displayable as a bar chart, pie chart or spreadsheet. These requirements are illustrated in Figure 7.8.

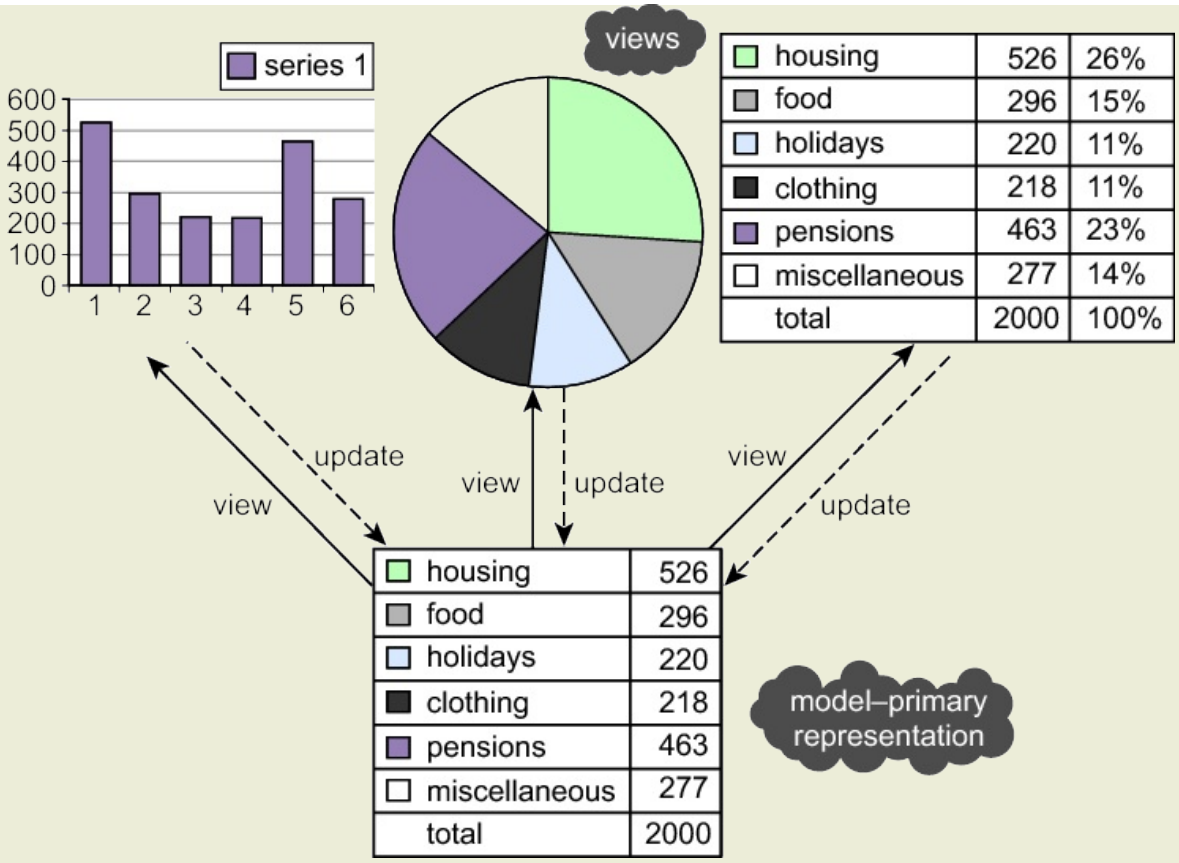


Figure 7.8 Multiple views of a personal budgeting system

View description

The primary representation of the data at the bottom of Figure 7.8 contains the budget headings and the absolute figures in a certain currency (here unspecified). The percentages and various displays are generated from this primary data. This primary representation forms the model, while the other representations form the views. The model can be changed through any view – for example, by changing figures in the spreadsheet view or by dragging lines in the pie- or bar-chart views. This capability is the input or controller aspect of the pattern.

Here we have a commonly recurring problem: that of maintaining several representations or views of data in step with each other, while permitting updates in any representation which must then be propagated to the other representations. We divide the system into three parts, as in Figure 7.9.

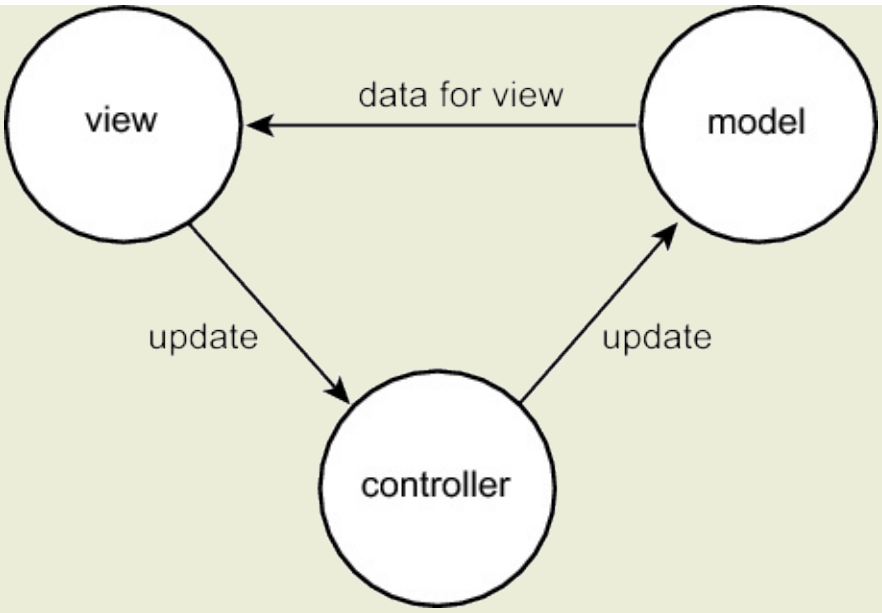


Figure 7.9 The MVC pattern

View description

The basic idea here is simple: views are generated from the data held in the model. Whenever some input − through a controller − changes the primary data for the model, the model then informs all views so that each in turn can update its representation of the data.

We could use this pattern to cater for future developments of the system – for example, to enable figures to be generated in other currencies, and to allow more sophisticated interactions to occur (such as keeping some figures fixed while others vary). Porting the software to various digital personal organisers could also be envisaged, as could sharing the system over a network. The pattern gives an abstract model of the software that enables all this to be done relatively easily.

Patterns have given us a powerful way to pass on good practice in software design. Experienced designers and programmers have proved very open to documenting and publishing their experience as patterns. Patterns can be used in any of the software development processes discussed in this unit.

One attraction of software design patterns is that they aren’t prescriptive: you can take them or leave them, choosing to use them only if they

benefit your design or some other activity. A variety of software design patterns, along with their applications to design and development, are presented in the companion module M813 *Software development*.

7.5.2 Patterns: application frameworks

With most patterns the aim is to facilitate low-level design of programs – but some patterns take an overall architectural view of the software, in the way that the model–view–controller does. Another well-known example of a protocol pattern is **representational state transfer (REST)**, used in many web applications. Model–view–controller and REST are general-purpose, but often these architectural patterns become application-specific, and in this form are commonly called **application frameworks** or simply **framework**s. Because of this association with applications, frameworks are often seen as commercially sensitive – capturing the know-how of the enterprise – though some have been published in sufficient detail for them to be useful for others producing similar applications.

7.5.3 Patterns: product lines

It's often the case that a company wants to produce several similar software systems. For example, today a car engine will be controlled by an on-board computer using quite sophisticated software. Different engines and cars require slightly different engine-control software, but all engine controllers are more or less similar for cars manufactured within the same period. A series of controllers can be built for different customers, using the same major components. The engine-control software manufacturer would only need to change the parts particular to a specific customer. Using the jargon of engineering manufacture, this is known as a **product line** – a line of products that differ only in detail (see Bosch, 2000). The way to develop product lines is to build on the idea of frameworks. Several steps need to be undertaken.

1. **Product-line initiation and domain analysis**. Before product-line development begins, there will have been some recognition that there is a class of commonly-recurring software systems where solutions differ minimally from each other. This leads on to domain analysis – a form of requirements elicitation where existing systems

in the domain are examined and domain experts are consulted. The area in which the product line will operate is very thoroughly analysed to work out general customer needs and the terminology that is normally used, leading on to general models of data and process: the **domain model**.

2. **Architecture specification**. Further analysis then leads to the architecture for the product line. Several such **reference architectures** or frameworks may be produced. These are further elaborations of the domain model to add detail, particularly about the prospective implementation, and with flexibility in mind. Each reference architecture will include any encoding necessary to make the architecture work − once components for deployment in the architecture have been decided on.
3. **Component collection**. As with a framework, each reference architecture will be accompanied by a repository of candidate components that could be used in the architecture. Nevertheless, it must be recognised that complete coverage is not possible or desirable, and that some new components and glue code may need to be written for a particular use of the architecture. All components in the repository must therefore be appropriately documented.

Once completed, these three steps – each very complex and difficult – leave us in a position to produce software systems for which the product line was created. For this, four further steps are required, each of which must be repeated for each new product in the line.

4. **Specific-requirements capture**. The requirements for a new product are captured, as specialisations and extensions of the domain model.
5. **Architecture specialisation**. A reference architecture is selected based on the specific requirements, and changes are made to incorporate the specialisations and extensions of the domain model corresponding to the new requirements.
6. **Component selection and specialisation**. The product that meets the new requirements is then built, with components selected from the repository where these exist, and adapted as necessary. If there is nothing appropriate in the repository, you may ask for a new component to be built by those managing the repository for this product line. Alternatively, you may develop the new component yourself if it's so specialised that it’s unlikely to be required in other

products.
7. **Integration and release**. Integrating the components into the architecture should be relatively easy – if the architecture has been well designed!

7.5.4 Patterns: product lines and process models

To be able to operate a production line effectively you'll need a lot of detailed understanding about what to do at each stage – OMG's model-driven architecture has much of what is needed here. Activity 7.5 explores the relationship between product line development and the process models you have encountered thus far.

Product lines let us build new systems that are based on similar successful systems.

Activity 7.5: The process model for product lines

Of the process models described in sections 7.2 to 7.4, which best fits the product-line approach to software production? Draw a process model for product-line development.

View discussion

7.6 Open-source methods

You learned in preceding sections that the received wisdom has been that software should be developed following sound scientific and engineering principles. You saw that in describing these approaches there were some problems with 'best engineering practices' – and you saw as early as Unit 2 that such problems were not surprising. Engineers who follow accepted good practice work methodically through abstract descriptions of the software before eventually writing the code. However, some people have rejected this framework. Such individuals have been

derogatorily referred to as ‘hackers’, who simply develop software directly in code. But the work of such people has not been a failure; what has evolved is a new approach to developing software, now commonly known as ‘open-source software development’.

Open-source software has achieved enormous popular appeal as FOSS – free open-source software – and through the Free Software Foundation founded by Richard Stallman in 1985 and described in Unit 3. There are two sides to open-source software: a use or deployment side that you’ve already encountered in Unit 6, and a development side that you’ll look at here. Open-source development processes have a reputation for delivering high-quality software for only a small additional cost and thus appear very attractive, if only they could be replicated.

7.6.1 Key elements of open-source

The key elements of open-source software development are worth examining. Following Feller and Fitzgerald (2002, chapter 6), open-source development involves:

- open code, with software engineers, other than those who developed the code, able to access it freely and flexibly and able to propose changes at any time
- established and proven architecture that is highly modular – everyone understands what the overall system being developed is and how the parts fit together, so individual engineers can work on components independently of others working on the software
- highly parallel development – having independent modules enables developments and updates to proceed in parallel, unaffected by each other, which enables very rapid development of the whole system
- lots of duplicate development – many people can be working on the same module, with the best solution being the one chosen
- rapid peer comment and feedback when a new or updated module is proposed for inclusion, with the added advantages of subsequent rounds of discussion from the original contributor and reviewers
- the possibility of acknowledged leaders and experts being party to the review processes, often as the ultimate arbiters on whether a contribution gets accepted
- increased user involvement – because open-source software is often

basic software that developers themselves use for other purposes
- highly talented, motivated developers who are among the best available and who probably use the software themselves, and therefore benefit from high-quality development
- rapid cycle of releases – the complete system may be rebuilt and released every day, or even more often, to ensure that the latest improvements are available to all, and that new contributions are given the ultimate test of full use in context
- a very loose ‘management’ (though this may be very varied, depending upon the particular lead person and other people involved)
- support tools (for instance, see SourceForge) that enable communication and manage software versions.

7.6.2 Further aspects of open-source

Open-source development is often characterised as being done using free voluntary labour – but clearly this is neither essential nor always the case, since major companies may pay their engineers to work full-time on open-source projects, subject to the controls of open-source rather than the controls of the company. Rather, it seems that a key motivation is to be working on exciting technology with extremely able peers, combined with a non-authoritarian management regime.

Seen as a software development process to be managed, open-source may seem very risky – you may not be able to lay out the work in a plan to ensure that development reaches predefined stages at predictable times.

> If you have no proprietary interest in the software you wish to develop, then consider making it open-source – inviting additions and enhancements as widely as possible.

Now complete Activity 7.6.

Activity 7.6: What lies behind open-source?

Discuss the open-source development process using all the concepts you learned about and understandings you gained in Block 1. Contrast this with the use and deployment of open-source software as discussed in Unit 6.

View discussion

7.7 Agile processes

The previous process models rely significantly on process and documentation, and are not able to manage change well. However, in 2001 a group of individuals met and agreed on an alternative approach to developing software. These individuals advocated lightweight approaches that embrace change, and their agreed approach is captured in the **Agile Manifesto**.

> 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.
>
> Agile Alliance (2001)

Underlying the manifesto is an equally important set of principles. The manifesto itself is often misquoted, leaving out the text before and after the bullet points above. Without this text the implication is that, in an agile project, no process, documentation, contracts or plans are needed – but that is emphatically not the case. **Agile processes** require discipline to be successful.

There are various **agile methods**. Most of them were represented at the 2001 meeting (often called ‘the Snowbird meeting’ after its venue, the

Snowbird ski resort in Utah, USA). The different methods provide support and guidance for different phases of development. For example, **XP** focuses on engineering (implementation), **Scrum** focuses on managing the engineering activity, and **DSDM** provides a framework for an entire project, including feasibility, foundations and engineering. Because of these varying emphasis, agile methods are often combined. For example, using XP with Scrum is quite popular, and this combination can also be used during the engineering phase of DSDM.

7.7.1 Rapid development

Whichever method is used, the central goal of agile development is the rapid development and deployment of code that provides value to the business.

Now complete Activity 7.7.

Activity 7.7: The Agile Manifesto and rapid development

How does the agile manifesto support the goal of rapid development?

View discussion

7.7.2 The cost–time–scope trade-off

When discussing or explaining agile working, it is often the manifesto and principles that are focused upon. However, there is another crucial idea underlying agile approaches that impacts significantly on how agile projects are run and organised. This is the cost–time–scope trade-off. Waterfall/plan-based processes assume that scope is fixed and both cost and time can be accurately determined. However, this assumption has in the past led to too many projects being late and over budget – so the reality has been that cost and time become flexible. In agile processes scope is flexible and the other two are fixed (see Figure 7.11).

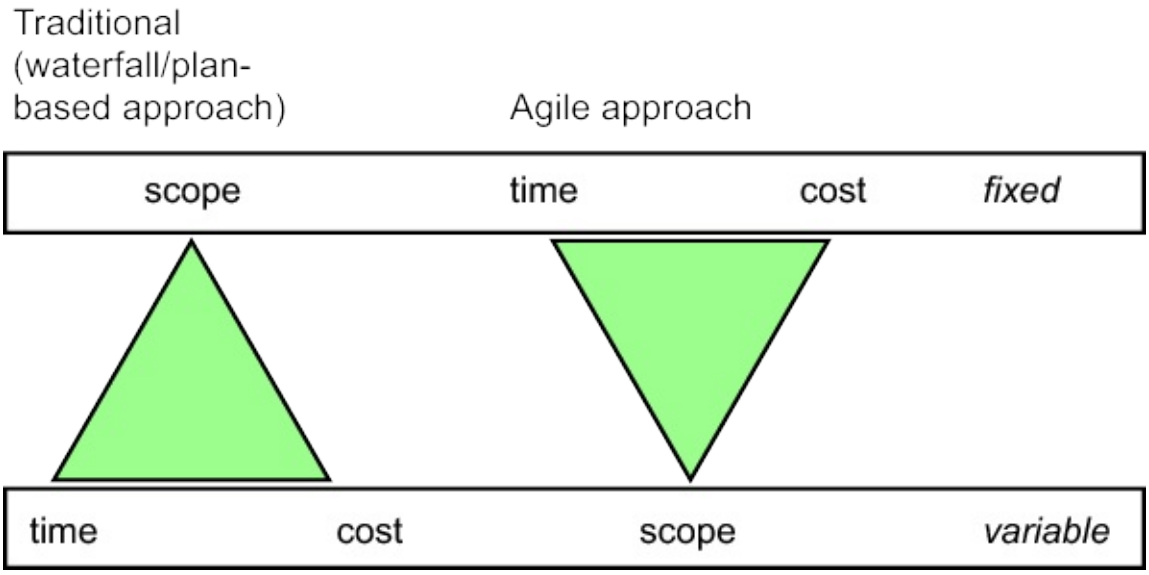


Figure 7.11 Comparison of cost–time–scope trade-off approaches

View description

This trade-off requires software engineering, both at the project management level and on a day-to-day basis to be organised differently, and a different mindset on the part of all stakeholders in order for it to work. Developers need to accept that they will interact more than previously with each other and with the owner of the requirements (i.e. the originator of the requirement). Customers and users need to understand that the project will be driven by their prioritisation of requirements and to accept that not all their original requirements will necessarily be delivered within the timescale set. However, they will receive operational software that meets their prioritised requirements much faster than they would have done using a different model. This can cause difficulties when the development is part of a programme in which customers are unable to prioritise functionality effectively because all the functionality is required, or when an agile project sits within a waterfall/plan-based governance environment.

An additional variable that generally is taken for granted, but suffers if the software team is under pressure, is code quality. By varying the scope but accepting fixed time and cost, code quality will be maintained: and provided the prioritisation of requirements is done well, software that provides value to the business can be released at regular intervals.

7.7.3 Timeboxing and Kanban

Keeping time constant leads to the idea of timeboxing. Elapsed time to delivery, costs, and resources are all kept constant, while the scope of what is delivered varies. Requirements are prioritised to decide which will be addressed during the current timebox, and a deliverable product that provides business value will be available at the end of each timebox, with increments delivered to the business after several timeboxes. This leads to an evolutionary development, and the agile process is often characterised as both iterative and incremental.

Different approaches use different terms for this short timebox. XP calls such items ‘iterations’, Scrum ‘sprints’ and DSDM simply ‘timeboxes’. Organising software development around timeboxes is not the only way to allow scope to be flexible while keeping the other elements fixed. For example, the process known as Kanban does not have fixed timeboxes; instead, it works on the principle of ‘limited work in progress’, or ‘limited WIP’. This controls development by specifying that at any one time only a certain amount of work can be ‘in progress’, and that once a requirement has been completed, resource is freed up to pull another prioritised requirement into development. In the rest of this module we will assume a timebox organisation.

Now complete Activity 7.8.

Activity 7.8: Kanban time–cost–scope trade-off

Review the *Kanban Blog* entry What is Kanban? Then consider: if timeboxes are not used in Kanban, how does this approach still conform to the time–cost–scope trade-off described above?

View discussion

7.7.4 Stories, story cards and the backlog

The basis of software development in an agile environment is stories. Stories are the closest that an agile team comes to a requirements

document, since user stories form the mechanism used to communicate user requirements to the development team. Such stories represent 'units of customer-visible functionality'.

A classic structure for a story is 'As a <user role> I want <functionality> so that <business benefit>'. For example, 'As a module team chair I want to be able to see the list of tutors working on my module so that I know who to contact with errata for the TMA tutor notes'.

7.7.5 Writing a user story

Now complete Activity 7.9.

Activity 7.9: User stories

Write a user story for the OU's eTMA system, from your perspective as an OU student.

View discussion

7.7.6 The agile implementation cycle

Once a set of stories for a product has been produced, these are ordered into a 'product backlog' – a prioritised list of stories that have not yet been implemented. This list can be re-prioritised during implementation as business goals and the environment change. Prioritisation is usually conducted by representatives of the business in order to maintain business value. There are different prioritisation schemes, but one common approach is the 'MoSCoW' approach, in which stories are categorised according to the following criteria.

- **M**ust: a requirement that must be satisfied in the final system for it to be considered a success.
- **S**hould: a high-priority item that should be included in the system if possible.
- **C**ould: a requirement that is considered desirable but not necessary.
- **W**on't (or **W**ould): a requirement that will not be implemented in the given release, but may be considered for the future.

All the cases of '**M**ust' will be at the top of the backlog; next come the cases of '**S**hould' and then those of '**C**ould'.

At the beginning of a timebox a planning meeting is held to determine which stories will be developed in that timescale. Some of these will be taken from the product backlog and some may have been carried over from the previous timebox. The planning game is a collaborative affair involving all team members. Customers are asked to prioritise stories for this timebox, with developers responsible for estimating how long each story will take. Together, the team can thus determine how many and which stories will be accomplished within the timebox. The stories chosen for the timebox are then pulled together to form a backlog for the timebox – often referred to as the 'sprint backlog'. Figure 7.13 illustrates the relationship between the stories, the product backlog and sprint backlog, and implementation cycles.

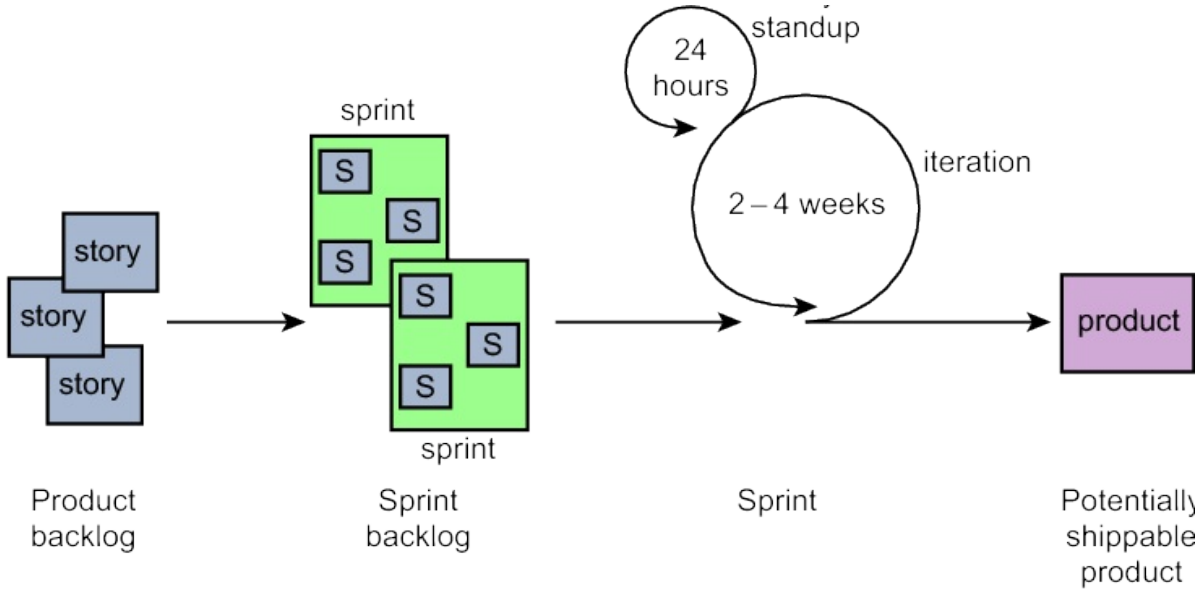


Figure 7.13 An overview of the agile implementation cycle

View description

7.7.7 The wall or progress board

Agile working promotes visibility: of difficulties, of successes, of customers, of users and of progress (among other things). One of the key

agile practices that supports this goal of visibility is the informative workspace: this means that the teams' environment displays information about the project, including the team's progress. Through the use of different colours, story cards provide one element of this visibility, and the wall or progress board provides a complementary element to support visibility of project progress. The wall is a vertical space on which the active story cards for this timebox are kept and are manipulated, annotated and moved according to their status. A key aspect of the wall is that its structure, and where cards are placed on this space, communicate the status of and other information about the story.

One simple structure for the wall places all the story cards yet to be implemented at the top of the space, and as stories are completed, cards are moved from the top to the bottom of the space. Figure 7.14 shows that all the cards for the timebox at the bottom of the wall have been implemented, while the cards on the right-hand side in the middle of the wall will be carried over to the next timebox.

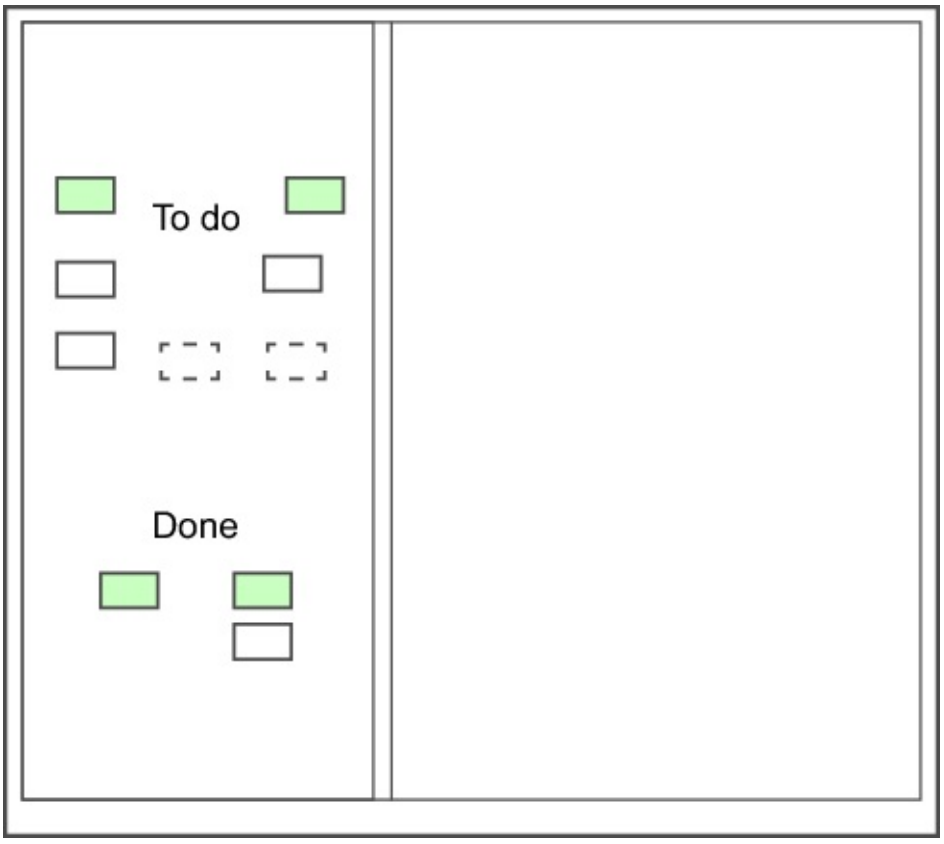


Figure 7.14 Example wall structure. Note that the green cards are stories and the white ones are tasks. The broken outline represent cards that are being worked on. Team members draw a 'ghost' of the card on the glass wall whenever they take a card away

View description

7.7.8 The wall: some examples

Teams can be very imaginative when it comes to finding a surface to use as the ‘wall’ – as suggested by Figures 7.15, 7.16, 7.17 and and 7.18.

Fig 7.15 In this case the company has made use of a glass partition

View description

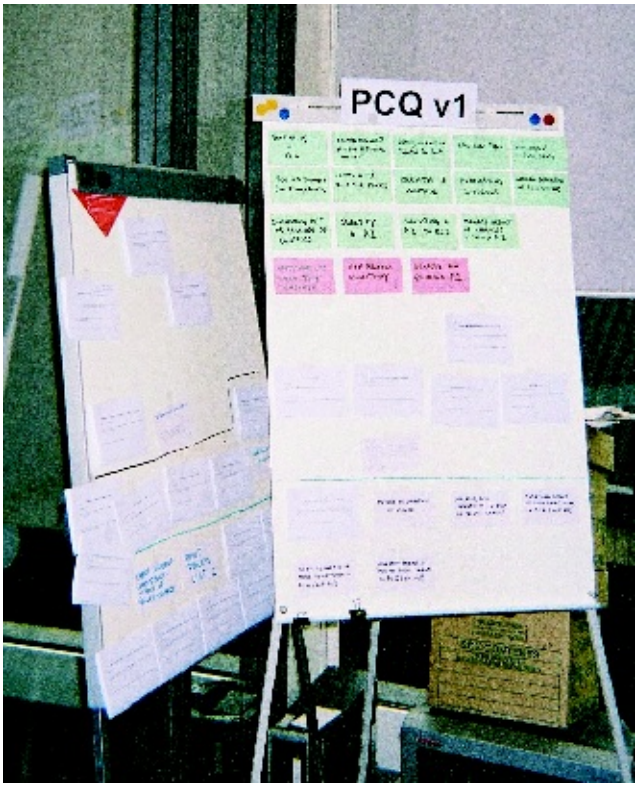


Figure 7.16 This team has used a flipchart easel as the ‘wall’

View description


Figure 7.17 Here a sliding door for a shelving area has been used

View description

Figure 7.18 And in these instances teams have pressed filing cabinets into service

View description

7.7.9 Working in a distributed environment

Now complete Activity 7.10.

Activity 7.10: Agile story cards

In a distributed environment, using story cards is less practical because team members can't share physical cards. Various support tools and digital equivalents have been devised, including agile-specific tools and spreadsheets. Suggest three disadvantages of the physical card and physical wall, and three disadvantages of the digital story and wall display. You don't need to investigate any specific tool, but do consider the differences in general.

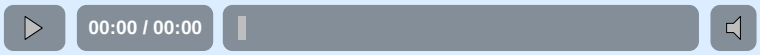
View discussion

7.7.10 Agile teams

Now complete Activity 7.11.

Activity 7.11: Agile teams

Listen to the interview with Helen Sharp on agile working and cross-functional teams and make some notes on the key points covered. Compare your notes with the description of iterative requirements roles presented in Chapter 14 of MRP. Summarise how techniques for agile working in cross-functional teams would apply to iterative requirements engineering.



Audio 7.1 Helen Sharp on agile working and cross-functional teams

View transcript

View discussion

7.8 Summary

We've based this unit on the set of typical activities identified in Unit 6. Based on these we proposed that:

- most software development processes comprise a number of typical activities, done in some prescribed order
- all work products conform to some standard and include models using some standardised notations.

In this unit you encountered a number of software development processes. Important points are summarised below.

1. Sequential process models work well for small problems.
2. Sequential process models and the methodologies built upon these have reached the limits of rationality, and are not able to handle large and complex projects properly.
3. Iterate to control risk, focusing on the highest risks in early iterations.

4. If it seems very difficult to produce software to an agreed specification, timescale and cost, you can adjust the set of functions delivered using timeboxing to facilitate this.
5. Product lines enable us to build new systems that are based on similar successful systems.
6. If you have no proprietary interest in the software you wish to develop, then consider making it open-source – inviting additions and enhancements as widely as possible.
7. Liberate your software engineers' energy and creativity by cutting bureaucracy to the minimum, making the technical work exciting, and engaging the best engineers you can.
8. If you have a highly prescriptive, documentation heavy process that is proving ineffective, consider taking up agile practices.

Agile Alliance (2001), *The Agile Manifesto* [online]. Available at http://agilemanifesto.org (accessed 19 February 2014).

Alexander, C. (1977), *A Pattern Language: Towns/Buildings/Construction*, New York, Oxford University Press.

Boehm, B. (1988), 'A spiral model of software development and enhancement', *Computer*, May, pp. 61–72.

Bosch, J. (2000), *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Reading, MA, Addison-Wesley.

Cohn, M. (2004), *User Stories Applied: For Agile Software Development*, Boston, MA, Addison-Wesley Professional.

Cohn. M. (2011), 'A sample format for a spreadsheet-based product backlog' *Mountain Goat Software* [online]. Available at www.mountaingoatsoftware.com/blog/a-sample-format-for-a-spreadsheet-based-product-backlog (accessed 19 February 2014).

Davies, R. and Sharp, H. (2006), 'Early and often: elaborating agile requirements', *Cutter IT Journal*, vol. 19, no. 7, p. 6.

DSDM Consortium (2008) *DSDM Atern Handbook* [online], available at www.dsdm.org/content/2-fundamentals (accessed 22 May 2014).

Feller, J. and Fitzgerald, B. (2002), *Understanding Open-Source Software Development*, Boston, MA, Addison-Wesley.

Hofstede, G. (2001), *Culture's Consequences: Comparing Values,*

Behaviours, Institutions and Organisations across Nations (2nd edn), London, Sage.

Jeffries (2013) .

Larman, C. (2002), *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, NJ, Prentice-Hall.

Larman, C. and Basili, V. (2003), ‘Iterative and incremental development: a brief history’, *IEEE Computer*, June, pp. 47–56.

Royce, W. (1970), ‘Managing the development of large software systems: Concepts and techniques’, in *Technical Papers of Western Electronic Show and Convention (WesCon)*, Los Alamitos, CA, IEEE Computer Society Press, pp. 328–38.

Sharp, H. and Robinson, H. (2008), ‘Collaboration and co-ordination in mature eXtreme programming teams’, *International Journal of Human–Computer Studies*, vol. 66, no. 7, pp. 506–18.

Trompenaars, F. (1993), *Managing Across Cultures*, London, Business Books.

Activity 7.1: The limits of sequential methodologies

Discussion

The emphasis on rationality and prescription may well alienate software developers, who may feel they're being treated like machines. In addition, the challenges that arise naturally in software development, and which make software development exciting, are not handled by the methodology, which then hinders rather than helps the software engineers. Room must be left for human insight and problem solving – processes that cannot be automated or captured in prescribed procedures.

As a result, it is all too common for methodology manuals to sit on their shelves gathering dust, and for the software tools bought to support them to go largely unused. The materials become 'shelfware', because the difficulties being tackled are not those anticipated by the methodology.

Back

Activity 7.2: Motivation in prototyping and participation

Discussion

Prototype development offers three advantages for the software engineer:

1. interaction with people and their problems
2. reduction in uncertainty over the software's usefulness
3. short delay between producing code and seeing something working.

You saw the importance of meeting social needs in the discussion about general motivation and Maslow's hierarchy of needs in Unit 3. Making software production into a social process is important for motivation.

You also saw in our discussion of cross-cultural factors, following Hofstede (2001) and Trompenaars (1993), that some people can handle uncertainty and others can't. Variation in how well one can handle uncertainty may be culturally determined, with members of some cultures more able to handle it – though clearly everyone has some difficulty with uncertainty and would like to avoid it.

Additionally, you saw long- and short-term orientation as a factor that also varies between cultures, but is equally appropriate when considering individuals. Everyone, to a greater or lesser extent, wants to see the results of their labour, and prototyping offers this – as indeed do open-source and agile methods (covered later).

Back

Activity 7.3: DSDM principles and techniques

Discussion

The current DSDM framework, called Atern, is described as an ‘agile project delivery framework that delivers the right solution at the right time’ (DSDM Consortium, 2008). The approach aims to deliver fixed quality, cost and duration in the early stages of the project, and discards lower-priority features in order to deliver at least a minimally functional product, on time and within budget.

Back

Activity 7.4: Timeboxing versus the spiral model

Discussion

Superficially, they do seem completely contradictory – until you realise that with timeboxing you recognise that a major risk for a project is that it doesn’t get delivered on time and within budget. Thus you can view each cycle of the spiral as a timebox aimed at managing the risk of slippage and cost escalation – delivering the most important functions at the end of each cycle, before starting on another to produce the next most important set of functions.

Back

Activity 7.5: The process model for product lines

Discussion

Figure 7.3 (in subsection 7.2.2 above), representing an incremental process model, provides the best fit for a process model for product-line software development. However, there are some differences – as indicated in Figure 7.10.

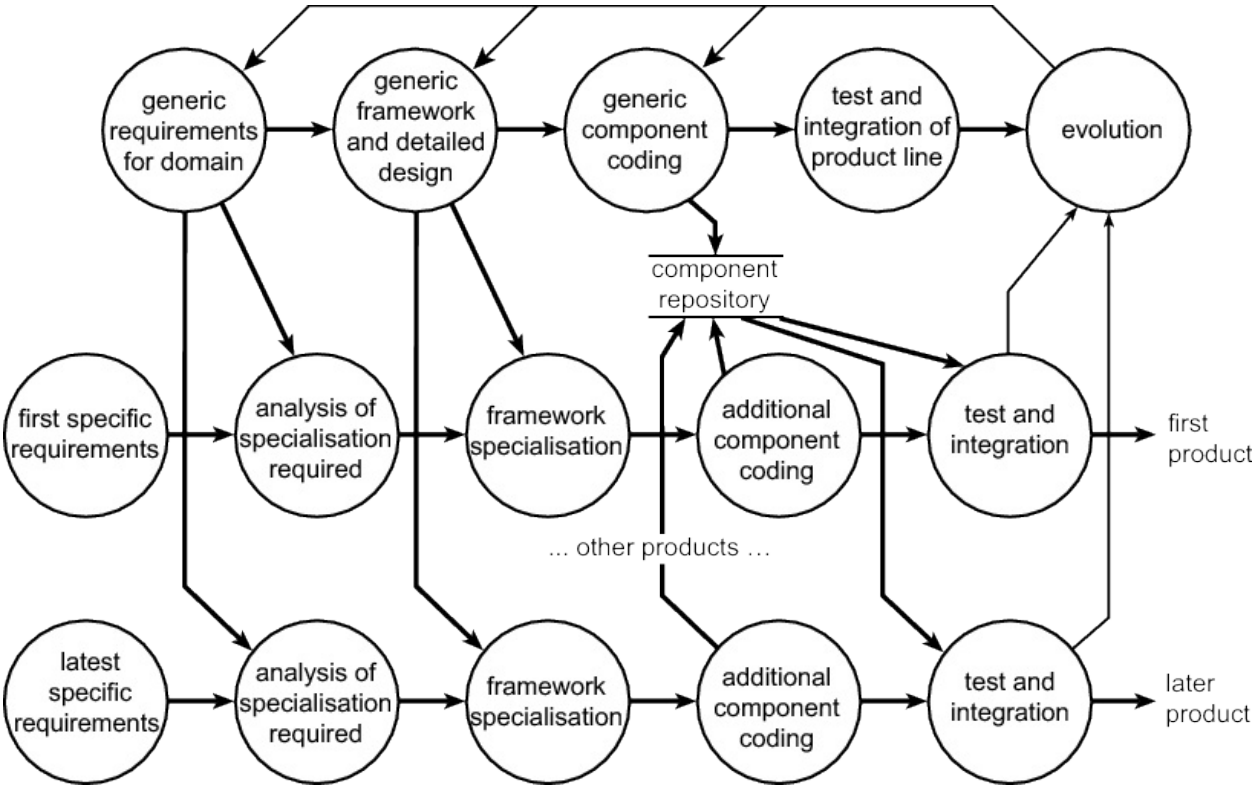


Figure 7.10 The product-line process

View description

An initial software development process along the top of the figure establishes the product line with a generic description of requirements produced by domain analysis, leading to an architectural framework and detailed design, and then a library of useful components. Some trial integration and testing is carried out to validate the framework and its

components. Thereafter the framework and components are evolved, with major feedback from use of the product line.

When a specific requirement comes through, the differences between this and the generic requirement lead to specialisation in the framework and identification of new components that need development, and from which the specific system can be built. This can be viewed as an increment in the overall process. The process continues with successive products in the line.

Back

Activity 7.6: What lies behind open-source?

Discussion

The key attribute of open-source software is that all the source code is made publicly available on a website, free for anybody to copy, modify and use, subject to the licence agreements attached to that source. The licence agreements effectively make the source code into public goods, as described in Unit 3.

All this means that if you have a use for the software, you can take it and use it. If you want to make changes and adaptations for your own use, then you are also free to do that. This is the way open-source was discussed in Unit 6 – as a source of software for someone seeking to acquire it. As you saw in Unit 6, there is no such thing as cost-free software: you have to do some work and spend some money, even when using open-source. But the overall costs can be considerably less than for proprietary software.

A further consequence of making software open-source is that it becomes open to improvement by anybody with an interest in improving it. Users of the software who are also expert enough to fix defects or add new functions can do so, subject to any controls enforced by the guardians of the software. This is the way open-source has been described in this unit − as a means of developing software. The effort required to improve or add to the software is given without payment, as part of the ‘gratis economy’ – though the person contributing the effort may benefit in other ways, such as having the improved software, experiencing personal development, learning from other very able people, gaining recognition from peers, and perhaps being able to sell expertise relating to the software.

Back

Activity 7.7: The Agile Manifesto and rapid development

Discussion

The manifesto is designed to shift emphasis from process and documentation to people and code delivery, relying more on the skill and professionalism of the developers and less on the strength of a process model and notation. For example, rather than spend time honing a requirements specification, agile team members will capture the essence of the requirement (in a user story, see below) and work out the details through conversations between the coders and whoever owns the requirement.

Back

Activity 7.8: Kanban time–cost–scope trade-off

Discussion

Good question. First, scope is still variable because the limited WIP cycle is driven by prioritised requirements. Second, time and cost are still fixed, but at a different level of abstraction – at the individual-requirements level.

Back

Activity 7.9: User stories

Discussion

We came up with the following (though your own output may be different): ‘As a student I want to be able to change my submitted TMA file so that I can correct any mistakes I have made before it is marked’.

Although stories are usually referred to as ‘user stories’, an agile team also uses the story to capture developer-initiated functionality (concerned with technical matters) as well as customer-initiated requests (concerned with customer-visible functionality). In addition, the story is often broken down into smaller units, called ‘tasks’.

The details on how to develop and maintain stories won’t be discussed here – for more, see Davies and Sharp (2006) and Cohn (2004) – but to remember what’s needed, Jeffries (2013) suggests a memorable ‘three Cs’: the Card, the Conversation and the Confirmation.

1. **The Card:** Stories and/or tasks are usually written on 3in × 5in index cards. Cards are small, physically independent entities. The size of each card limits how much information can be written on it, while the independent nature of cards means they can be annotated and manipulated during meetings or discussions.
2. **The Conversation:** Because the card can hold only a limited amount of information, the development team has to talk to others in order to explore the detail of the story and to refine their understanding in order to implement it.
3. **The Confirmation:** Testable and measurable user acceptance tests are agreed between the customer and the development team, so that everyone concerned understands when a story has been implemented successfully.

The physical card entry is usually structured in a particular way and annotated according to progress – often using different coloured pens and stickers. The exact structure varies from team to team and will have been developed according to the team’s own style. An example story card and its idealised structure are shown in Figure 7.12. Colours are important: they’re used to make progress clearly visible. Also, the fact that the example card is itself green indicates that this is a story and not a

task – task cards are white.

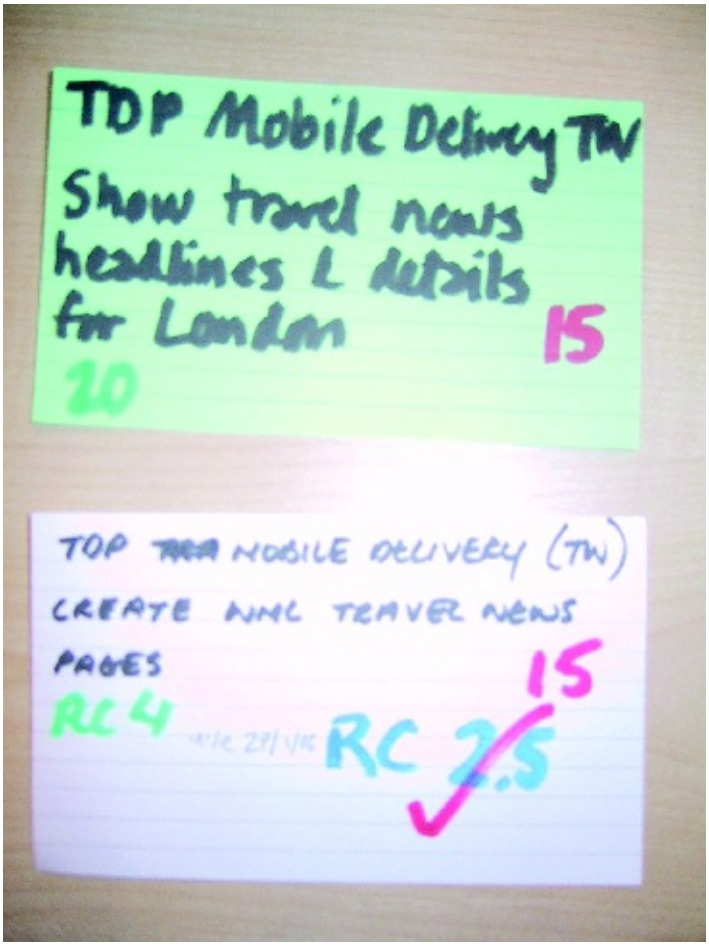


Figure 7.12 (a) A story card annotated with progress days

View description

Figure 7.12 (b) The idealised structure of the card. Note that a red tick means that the story is finished (Sharp and Robinson, 2008)

View description

For teams that are in an environment where physical cards are not the main representation of story cards (for example, in a distributed setting or where project governance expects electronic updates), a spreadsheet is a popular alternative. This representation has its advantages and disadvantages but the same ‘As a <user role> I want <functionality> so that <business benefit>’ format can be used. Cohn (2011) provides a good discussion of this.

Back

Activity 7.10: Agile story cards

Discussion

Disadvantages of the physical system that are often cited are:

1. The physical cards could get lost or damaged.
2. The physical artefacts can’t easily be shared with any colleagues who aren’t co-located.
3. Different versions of the physical artefacts can’t easily be saved to show progress over time. Teams using physical environments sometimes take photographs of the wall and post them on a wiki or in another online communication environment to share with distributed team members.

Disadvantages of the digital system are less easy to identify, but here are three that have been observed:

1. Activity around a digital board is hard to observe because everyone can (probably) access it from their own workstation. This reduces the visibility of team activity and progress.
2. Digital story cards cannot be annotated as creatively.
3. Use of any software system comes with a learning curve, however gentle, while the use of pens, stickers and adhesive tack does not rely on any special knowledge or learning.

Back

Activity 7.11: Agile teams

Discussion

Many of the challenges facing cross-functional teams that use agile methods as identified by Sharp are also applicable to the inclusion of business analysis in the iterative development process. MRP agrees with Sharp's view that shared knowledge and understanding is crucial to the success of teams using iterative, agile development methods.

MRP recognises that it can be a challenge to find individuals who have all the skills needed to answer business questions but who also understand the technical issues. However, MRP does not suggest any of the techniques for getting the technical team to engage with the business analysts, as mentioned by Sharp, such as the 'design studio'.

Back

Description

Figure 7.1 is a simple dataflow diagram with five processes and no terminators. At top left there is a process labelled ‘requirements’ with data flows to and from the next process down and to the right, labelled ‘design’. This then leads through data flows in both directions to the ‘coding’ process, and thus on to the ‘test and integration’ process, and finally through data flows in both directions to the ‘evolution’ process at bottom right.

If you were to add terminators there would be one at the top left feeding information into the requirements process, and one at the end exchanging information with the evolution process. Both these terminators would be labelled ‘stakeholder’ or similar.

Back

Description

Figure 7.2 is an elaboration of Figure 7.1, with nine process. The ‘requirements’ process at top left now has three flows out of it, leading to the three parallel strands of development which converge again at the process ‘test and integration’, which in turn leads in a single flow into the process ‘evolution’. The top of the parallel flows from ‘requirements’ starts with the process ‘user interface design’, which then leads on to ‘user interface coding’ before its flow converges on ‘test and integration’. The second parallel flow goes from ‘requirements’ to the ‘application design’ process and thence to the ‘application coding’ process, before converging on ‘test and integration’. The third parallel flow goes from ‘requirements’ to the ‘database design’ process and thence to the ‘database coding’ process before converging on ‘test and integration’.

Back

Description

This data flow diagram consists of three horizontal sequences each of four processes, labelled successively ‘requirements’, ‘design’, ‘coding’ and ‘test and integration’. The top row has the process labels prefaced by ‘initial’ and the second row by ‘additional’, and then after a gap the third-row labels are prefaced by ‘final’. Flows also go diagonally down the page, so that ‘initial requirements’ pass on information to ‘additional requirements’, and so also for ‘design’ and ‘test and integration’ but not for ‘coding’.

Between the second and third rows there is a gap labelled ‘other increments’ with arrows leading downwards out of the second row and arrows leading downwards into the third row to show that there could be many other rows in between.

Finally, at bottom right a flow goes from process ‘test and integration’ in the third and ‘final’ row to the process ‘evolution’.

Back

Description

This dataflow diagram has nine processes arranged with a cycle of three processes at top left and a sequence of five processes along the bottom.

At top left the process ‘system requirements’ has a flow into the process ‘prototype design and coding’, which is the first of the cycle of three processes. A flow from this process goes to ‘prototype evaluation’, from which two flows emerge.

One flow goes to the process ‘updated requirements’ and then back to ‘prototype design and coding’ to complete the cycle.

The other flow, from ‘prototype evaluation’, goes to the start of the simple sequence at process ‘full requirements’, with a flow to process ‘design’ and so to process ‘coding’ and the process ‘test and integration’ and finally to process ‘evolution’.

Back

Description

Figure 7.5 shows a cycle of four processes with flows in an anti-clockwise direction. The flow diagram starts with a flow labelled ‘initial need’ at top left, flowing into a process ‘requirements’. From there a flow leads to the process ‘design’, and then to process ‘coding’, and so to the process ‘test and integration’. From ‘test and integration’ one flow goes back to the process ‘requirements’ to close the loop, while an alternative flow labelled ‘next release of system’ goes out of the diagram.

Back

Description

This is a reproduction of the original diagram from Barry Boehm and is not a data flow diagram. It consists of a clockwise spiral moving outwards where the radial dimension roughly corresponds to accumulated cost to date, shown at the top of the diagram. The area is divided by horizontal and vertical axes into four quadrants. We will follow round the spiral noting what is recorded as happening at that point.

The spiral starts in the top left quadrant; all the sections of spiral in this quadrant are labelled ‘Determine objectives, alternatives, constraints’.

The innermost spiral then goes on in the second quadrant at top right to do risk analysis followed by prototype 1. In fact, all turns of the spiral have risk analysis followed by prototyping in the quadrant, with the general overall activities for this quadrant described as ‘Evaluate alternatives, identify resolve risks’.

All turns of the spiral start in quadrant 4 with ‘Simulations, models, benchmarks’. The innermost spiral then goes on to ‘Concept of operation’ before considering the ‘requirements plan, lifecycle plan’ in the fourth quadrant.

The second turn of the spiral then determines objectives etc in the first quadrant, risk analysis and prototype 2 in the second quadrant, and so round to ‘Software requirements’ and then ‘Requirements validation’ in the third quadrant, and ‘development plan’ in the fourth quadrant.

And so round again through determining objectives, then risk analysis and prototype 3, and so to ‘Software Project design’ and ‘design validation and verification’ in the third quadrant, and into integration and test plan in the fourth quadrant.

The final cycle of the spiral shown on the diagram then goes round through determining objectives for this round followed by risk analysis and then the operational prototype in quadrant 2. This then leads into usual sequence of software development activities in quadrant 3 – detailed design, coding, unit test, integration and test, acceptance test, and implementation.

This then ends the spiral with the note in quadrant 4, ‘plan next phases’.

Back

Description

This is a very simple diagram, a triangle with vertices marked ‘functions’, ‘costs’ and ‘timescale’.

Back

Description

Figure 7.8 is not a data flow diagram. It shows at the bottom a table with two columns with text entries on the left and numbers on the right, with entries such ‘holidays’ and ‘220’. The exact values of the entries are not important. To the right of this table is the comment ‘model – primary representation’.

Above this are three separate small figures, collectively described as ‘views’ in a comment. Of these, the leftmost shows a bar chart, the middle a pie chart, and the rightmost a table. Arrows lead from the table at the bottom labelled ‘model’ to each of these three views at the top.

Back

Description

Figure 7.9 is a three-process data flow diagram. On the right at the top is a process labelled ‘model’. To the left at the top is a second process labelled ‘view’, with a flow from ‘model’ to ‘view’ labelled ‘data for view’. In the centre at the bottom is the process labelled ‘controller’. The view sends an ‘update’ to the controller, which the controller then passes on to the model.

Back

Description

This data flow model is similar to one you saw previously in Figure 7.3, for incremental software development. This data flow diagram consists of three layers each of sequences of processes. The top layer consists of a sequence of five processes starting with process ‘generic requirements for domain’ at top left, with a data flow to ‘generic framework and detailed design’, then ‘generic component coding’, to ‘test and integration of product line’ and so to ‘evolution’. A feedback pat, shown with a lighter line, goes back from ‘evolution’ to each of the four preceding processes. There is a data store ‘component repository’ just below the process ‘generic component coding’, with a flow from that process to the store.

The second layer starts at the left with the process ‘first specific requirements’ and then data flows from there to ‘analysis of specialisation required’ with a flow from the ‘generic requirements for the domain’. From there the main flow goes to ‘framework specialisation’ with a flow in form ‘generic framework and detailed design’. Then to ‘additional components coding’ with a flow from there to the store ‘component repository’. And so finally to ‘test and integration’ on the right with input from the ‘component repository’ leading to major output flow ‘first product’ and a minor flow to ‘evolution’ to feed back into the generic product.

The lowest layer consists of five processes starting with ‘latest specific requirements’ and so on through the same processes but only for this latest product from the line. As with the first product sequence, there are flows down from the generic sequence to this latest sequence.

In between the first and latest sequences the lines are broken and the comment inserted ‘other products’, to indicate that there may be many products created from the line between that first and this latest product.

Back

Description

The diagram consists of two main sections. At top left is a label ‘Traditional (waterfall/plan-based approach)’. Underneath this there is a green equilateral triangle with one corner pointing upwards. This top corner is labelled ‘scope’ and the bottom two corners are labelled ‘time’ and ‘cost’ from left to right. The label ‘scope’ is enclosed by a rectangle labelled ‘fixed’. The two labels below the triangle are enclosed by another rectangle labelled ‘variable’. To the right of the label starting with ‘Traditional …’ is another label ‘Agile Approach’. Beneath this is a green triangle that is inverted with respect to the first one, with two corners at the top and a single corner pointing downwards. Beneath this corner is the label ‘scope’, which is enclosed in the rectangle labelled ‘variable’ that was mentioned earlier. Above the triangle are the labels ‘time’ and ‘cost’ which are enclosed in the rectangle labelled ‘fixed’ that was mentioned earlier.

Back

Description

This is a colour photograph of two story cards, written on index cards and placed on a table. The cards are positioned one above the other and have been photographed from above. The top story card is written on a light-green index card and the bottom one on a white index card.

The top card has the handwritten words ‘TDP Mobile Delivery TW, Show travel news headlines & details for London’ in black ink. At bottom left of this card the number ‘20’ is written in dark green ink. At bottom left of this card the number ‘15’ is written in red ink.

The bottom card has the handwritten words ‘TDP Mobile Deliver (TW), Create WML Travel News Pages’ in black ink. Underneath this the following items have been written from left to right: ‘RC 4’ (in green ink), some illegible numbers, and ‘RC 2.5’ (in light blue ink). The label ‘RC 2.5’ has a red tick-mark drawn over it with the number ‘15’ written just above it, also in red ink.

Back

Description

This is a drawing of a story card depicted using a white rectangle. Inside the rectangle the words ‘Project title’ and ‘project plan row number’ are written at top left and top right respectively. Underneath this is the word ‘Story’ followed by a series of grey horizontal bars denoting the text of the story. At the bottom of the rectangle are the words ‘Initials’, ‘Estimate’, ‘Date’, ‘Initials’ and ‘Actual’. A large red tick mark has been drawn in between the words ‘Initials’ and ‘Actual’.

Back

Description

At the left-hand side of Figure 7.13 there are three overlapping blue rectangles, each labelled ‘story’. Underneath these rectangles is the label ‘Product backlog’. There is an arrow pointing from these rectangles to a set of slightly larger green overlapping rectangles. Inside each green rectangle is a small blue rectangle, each labelled ‘S’. Underneath these green rectangles is the label ‘Sprint backlog’. There is an arrow pointing from these green rectangles to a large circular arrow that loops around anti-clockwise and has the label ‘Sprint’ underneath it. Inside the looped arrow is the label ‘2–4 weeks’ and just outside it on the right is the label ‘iteration’. There is a smaller looped arrow at top left of the first one, with the label ‘24 hours’ inside it and the label ‘daily standup’ just outside and to the right of it. Another arrow points to the right, to a red rectangle labelled ‘product’; underneath this is the label ‘potentially shippable product’.

Back

Description

This figure shows a large rectangle with a double lined border. The rectangle is split into two vertical sections by another double line. In the left-hand section the top half has the label ‘To do’ and is surrounded by seven smaller rectangles (depicting story cards). Of these story cards the top two are green, the next three down are white with a solid border and the final two are white with a dotted-line border. In the bottom half of this section of the rectangle there is a label ‘Done’, with three smaller rectangles underneath it. Two of these rectangles are green and one is white.

Back

Description

This is a colour photograph showing how a glass partition between two rooms in an office building has been used to display story cards.

Back

Description

This is a colour photograph showing two flipchart easels being used to display story cards

Back

Description

This is a colour photograph showing a whiteboard used to display story cards

Back

Description

This is a colour photograph showing a set of metal filing cabinets being used to display story cards

Back

Transcript

Interviewer So to start off with, I wanted to ask about why cross-functional teams are so important in agile work?

Helen Well, there's several characteristics of agile teams and agile working that means that it's quite an advantage to have cross-functional teams. Some agilists would say that cross-functional teams are crucial and you can't do it satisfactorily if you don't have them. So things like rapid feedback; the fact that you're iterating very often over short periods of time to get some functionality out; increased communication, so agile working downplays the importance of documentation over face-to-face communications. So if you've actually got cross-functional teams who are working together with the many different specialisms, then you can talk to people and find out what the issues are very easily. One of the other characteristics is fail-fast. So the idea is that you are working with the codes, you're trying to produce the system, whatever the software is that you're writing. And if you can find out very quickly what's going to work and what won't work that allows you to learn from that, and then you can improve the product.

So if you're going to fail-fast, again, you have to be able to talk to the specialists that you need to find out whether something will work or not.

Interviewer What are some of the challenges for cross-functional teams in an agile context?

Helen

You can think of them probably in about three different kinds of areas, roughly, just to give an idea. The first one is organisational. Many organisations have limited resources so they may not actually employ the different specialisms you need for that particular team or particular piece of software you're developing. In which case, they need to either bring in consultants or they're going to be working with another organisation. And that immediately sets up some kind of barrier because you've got a different kind of relationship going on there. Even if they have some that they've employed themselves they may have limited resource, so actually you can't have one dedicated UX designer or database designer, or whatever it happens to be. You can't have one dedicated person on that agile team. So they have their time split. So the notion of them being one team, of being a fully-operational cross-functional team, it just isn't going to work.

And there are some organisations that genuinely believe that disciplines work better if they sit with their own kind, so to speak. So instead of having a cross-functional team where you have developers, testers, database designers, UX designers, or whatever, all in one place, they actually believe it's better to have all the UX designers in one place, all the database designers in one place, and so on. And so sometimes organisations themselves set up those barriers. So that's organisational.

And then you've got what you might call physical kinds of challenges, that there may

also be very good reasons why people are distributed. So not only are they separated because of disciplinary reasons or something, but you may need a specialism that lives, works in a different country or in a different part of the same country. So again, you've got this issue of distributors and that sets up barriers for cross-functional teams.

And the last one, really, I've mentioned discipline several times, people who come from different disciplines are trained in different ways, they have different kinds of backgrounds, they have different goals. There's a range of different things which means that they come into the team, even if they are one co-located cross-functional team, there are barriers even then that cause some challenges.

Interviewer What are some of the factors that affect how team members from different disciplines work together in agile teams?

Helen In order for a cross-functional team to be effective, each individual needs to understand the concerns of the other members of the team. Now, if it's a developer then one would hope that there'd be some kind of understanding there. But if you've got people from different disciplines, like a UX designer and a developer, they may not be fully aware of what each other is doing and what their main goals are. Now, that may sound a bit daft but it is true that that very often happens – people work in one particular area and they're not really sure what another area of the discipline does. So, for example, disciplines often have different perspectives and different

goals. And one of the things, again, if I can go back to the UX designer example, when UX designers produce their designs, for them that's their deliverable. They've done the design and it's documented in some form which could be a prototype, it could be all sorts of different kinds of things. But of course, to the developers that's a consumable.

And so there are different perspectives on what's going on. So in an agile context, if developers then come back and say about the design, 'Well, I want to change it, I need to tweak it,' for some particular reasons, or, 'That bit doesn't work,' then that's quite strange for a UX designer. Because it's their deliverable and they've finished. So you get things like that, different perspectives and different goals.

There's also the matter of different disciplines have different processes about the way they design things or produce whatever they're going to produce. And designers tend to develop complete designs. So they want to hand over something which is sometimes called pixel-perfect when it's UX design, so they've really worked through the detail and it's a complete coherent design concept which they hand over. Which, as I'm sure your students will be aware, in agile development that's not the way it works. You're trying to do rapid iterations. So the notion, again, of having this complete upfront design, sometimes again called Big Design Upfront, that's something agilists try to avoid.

So you've got the different processes. And there are different skills and knowledge. Developers tend to have a lot of technical knowledge; database designers may have very detailed knowledge about how to design a database or different kinds of databases you can work with. UX designers will understand about users and how to involve users, and maybe some aesthetics as well. So there'll be a whole range of different kinds of skills that go hand in hand. The cross-functional team doesn't need everybody to understand everything, but it certainly needs an awareness of what's going on and who to go to to ask things and how to work with each other, really. And you need to understand those things for it to be successful.

And finally, there are different commercial pressures that often arise when you've got particularly UX designers and developers. When you're running agile teams as efficiently as possible, you try to avoid blockages and duplication, and so it's tempting to run in a staggered, parallel kind of fashion so that each team can focus on its areas of specialisation. When actually, what you're trying to do really, in a cross-functional team, is to mix people up far more, and mix up the activities far more. So there are commercial pressures there as well, which make a difference in the different disciplines.

Interviewer

We've talked a lot about the interactions between user experience designers and the development team in an agile context. I was wondering if you could elaborate a little more about some specific examples of how

these two disciplines integrate in this environment?

Helen

This is an area that practitioners have been very keen on, of course, they want to improve their practice all the time. Again, that's something agile working asks for. It's also been studied a lot by academics. And I'd like to think of them in terms of two broad categories of how would you go about bringing together UX designers and developers.

The first category is basically talking about bringing people together, so I've talked a bit about some of the challenges and people being separated. But if you think in terms of bringing people together in a cross-functional team, that again is something that agilists think is really important, to have co-location. That's great and that's important and it means people will talk more to each other, but just bringing people together may not be enough. So that's one of the things, just bringing people together in a co-located fashion can work. But in addition to that, I think, there's been some research on what happens when you do that – does it work? Can you just put people together and all the problems are solved? And of course, the answer is no.

And there's actually quite a lot of detailed day-to-day working that has to go on, where people talk to each other a lot but they're also trying to understand the different perspectives. The kinds of things that I've been talking about, different goals, different perspectives. And that is an overhead which, sometimes, managers don't realise,

and that can be frustrating for people if they don’t appreciate that that might happen. So although you bring people together, you’ve got to manage somehow this notion of regular meetings, talking to each other formally, informally, a range of different things like that.

However you bring people together with a specific purpose of sharing some of these things I’ve talked about, then that can be beneficial. So there’s a thing called a design studio where every member of the cross-functional team comes, and they come to this meeting, it can be a day, it can be a half-day, whatever fits with the team schedule. And the idea behind that is that these design studios produce designs. So they’re actually producing designs for the product, whatever it is, but it’s not the designers doing it, it’s the developers or the database designers or the testers, or whoever else. So they have a chance to bring their own creativity to the product, to understand some of the issues maybe the designers have been dealing with that they haven’t quite realised, haven’t quite come to terms with. So that’s quite a good way of doing it. And maybe the designs will be used in the final product and maybe they won’t, but at least it gives people a chance to explore ideas and exchange different kinds of understanding about the goals and skills that people bring.

So bringing people together is one kind of way that people approach this. Another is to align developer and UX design work practices, so taking techniques, for example, from UX design and bringing them

into the development. A very popular one is something called a ‘persona’, which is basically a description of a typical user. And usually that’s the preserve of UX designers; they produce the personas and they own them in some kind of sense. But I’ve seen several teams where they don’t just have a persona as a basis for design, they hand them to the team. So very often they’re blown up as big posters on the walls and they have names, they’re called Mary or Tim or Tony, or whatever it happens to be. And developers will be talking about, ‘Well, how would Tony react to that bit? If we do this and we design the system this way, or we try and have an interaction like that, will Tony like it? What about Mary? Mary’s different than Tony, how will she cope with it? So they do actually work quite well with these, it can inform what people are doing’.

So personas is one of the things that is quite often incorporated. Other things that have been included is something called ‘discount usability’, which is a way of doing evaluations of something, but it’s very, very short. It doesn’t take very long. So again, it fits in well with the iterations and the rapid development of functionality that agile is trying to achieve.

Scenarios also, how will this product be used? What will the user do when they first come to the system or the product, and how will they benefit from using it? So some of these techniques have been used alongside agile working, and they do quite well, actually. But that’s sort of taking things out of UX design and putting in a development context. Several people I know as UX

designers become the Scrum Masters, so they will actually take on board the education so that they understand better what it is, what Scrum is about and what Scrum Masters are doing. And I picked that one in particular because Scrum is quite popular at the moment and there's a lot of people becoming Scrum Masters. And that helps them to know which techniques to bring in and how to bring them in.

Another very popular thing to do is to have what's called 'train tracks' development, where you have two parallel streams – one of them is UX design and the other one is development. And basically, what happens is UX design works one sprint ahead of the development sprint. So the UX designers will do some work and their deliverable at the end of the sprint will feed forward into the next sprint of the developers. Some teams work two sprints ahead, so I've seen that, depending on the nature of the problem, the nature of the product and what they're trying to achieve.

Also along the lines of trying to align work practices, there are several sets of recommendations which have been developed mostly by practitioners who are in the field and trying to share experience. And there's quite a good set of 12 recommendations from Jeff Patton for a successful UX designer within an agile context. And one example is to research, model and design upfront, but only just enough. And of course, the difficulty is in what is just enough, and that's something that people have to work out for themselves, really. Teams need to

understand what is enough – so how much do the UX designers need to do before developers can actually start working? But it's quite nice to encapsulate it that way; people can think about what that means for them.

Another one which I also quite like is that you should buy design time with complex technical stories. And what that basically means is, I've talked about designers working one or two sprints ahead, well if you actually have a very complicated technical story, so it doesn't require any UX design, the developers can be working on that in parallel. And that gives the UX designers time to do some particular area of design where they need to do maybe more evaluation, or more user research, or whatever it happens to be. So if you can align those two then it allows them to get more design time in while the developers are busy. Because one of the key things is that you don't want developers to be sitting around twiddling their thumbs, waiting for UX designs.

Interviewer So some of these approaches for addressing the challenge of user interaction design and agile practices seem to be, it could work with other disciplines as well, like database designers and testers. Is my intuition right?

Helen Yes, I mean, to a degree. Obviously, where I've talked about specific techniques like personas, that is something particular to UX design. But you certainly have the same challenges where some organisations want their testers to all be together, or they want

their database designers to all be together. And there are differences in terminology, differences in goals and perspectives again. So certainly, if the challenges are there, and some of the things I've talked about getting people to work together and understanding each other's concerns, that certainly is something which can help to overcome the challenges, yes.

Interviewer If we take a step back and look at the big picture of agile working, could you say something about where you see it going, both in terms of challenges and new practices for development teams? But also, the wider organisations in which software is produced?

Helen More and more organisations are taking agile on board in some form or another. Very often, of course, that starts with software, so you've got software teams who are developing agile. But it does actually require the organisation to become more agile anyway. Or clients, if you're working in a software house then your clients need to understand how you're working. If you're in a larger organisation then the organisation needs to understand. And I'm aware of several places where they've actually tried to take the agile principles out into the wider organisation. So instead of just doing your software this way, how can you run human resources this way? How can you run the accounts department this way? And they all may sound fairly scary kinds of ideas, but there are a lot of benefits. With failing-fast and thinking in terms of shorter prioritised goals, what is it we need to achieve? Obviously, it doesn't work for everything but

there are more examples of where the agile approach to things is being taken out into the wider organisations. And of course, agile is learning from that. There's feedback from, 'What does it mean for accounts to be agile?' that comes back into the software way of working.

In addition to that, there are, as I said, many more organisations taking agile on board. So whereas in the past people might have said things like, 'Safety critical systems we can't do in an agile way, there's been resistance within the public sector on agile working, we just don't work that way,' some of those opinions are changing, I think. And there are more and more people adopting us.

In terms of where will agile go, I think practitioners are very pragmatic, of course, and the whole point of agile is that you're trying to inspect and improve your process and your working all the time. And so agile is evolving, it is changing, different things are being adopted at different times in different places, and they're being adapted by people. And that will continue. As I said, practitioners are pragmatic, so it will continue to evolve and improve.

Interviewer That's really great, very interesting. So thank you very much, Helen, for your time.

Helen That's okay. Nice to meet you. Thank you.

Back