UNIVERSITY
OF LONDON

Software engineering project management

D. Murray and N. Sandford

CO3353

2013

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London by:

Dianne Murray

Neil Sandford

Putting People Before Computers, London, UK.

This is one of a series of subject guides published by the University. We regret that due to pressure of work the authors are unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

In this and other publications you may see references to the 'University of London International Programmes', which was the name of the University's flexible and distance learning arm until 2018. It is now known simply as the 'University of London', which better reflects the academic award our students are working towards. The change in name will be incorporated into our materials as they are revised.

University of London
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

london.ac.uk

Published by: University of London

Contents

Preface

About this course and subject guide

This is the subject guide for the Computing **CO3353 Software engineering project management** course. It introduces key concepts addressed by the Essential reading textbooks. For example, we give overviews of several testing techniques. However, the textbooks describe a larger number of techniques and explain them in more detail. The subject guide is intended to support and reinforce personal study, not to replace it. It will not be possible to pass this course by relying only on the subject guide. Do not cite this subject guide in your coursework or examination papers. Instead, go back to the original sources.

Software engineering project management focuses on techniques for managing software engineering (SE) projects, and builds on core software engineering concepts. A pass in Computing **CO2226 Software engineering, algorithm design and analysis**, is therefore a prerequisite, and you will also benefit from some programming experience, ideally as part of a team.

Course aims

This course provides an understanding of both theoretical and methodological issues involved in modern software engineering project management and focuses strongly on practical techniques.

The scale and complexity of the software systems now being developed demands that software engineers work in multi-functional teams and that they adopt scalable and robust methodologies and tools. As a student of business and creative computing, you need to develop the transferable skills in logical analysis, communication and project management necessary for working within team-based, professional environments.

You also need to extend your knowledge and understanding of the software development lifecycle and fundamental software engineering concepts (such as object-oriented programming, the software lifecycle, design for re-use and user-centred design). In addition to knowing and understanding the principles of traditional development methodologies, being able to understand and put to use contemporary approaches (such as '**Agile**' methods of software and project management) will help you to produce more robust processes and designs for delivering successful projects.

You should also learn to utilise that knowledge in a variety of contexts, ranging from embedded systems to the inherently parallel distributed environments of cloud computing.

Essential reading textbooks

Students will need to purchase **either** Ian Sommerville's book *Software engineering* (note: this includes advanced engineering topics beyond the scope of this course; such as reliability and security issues and the special demands of distributed and embedded systems) **or** Roger Pressman's book *Software engineering: a practitioner's approach* (with additional material for students wishing to reinforce their understanding of the fundamentals of software engineering and more than a quarter of the book dedicated to quality control techniques and testing). Both Essential reading textbooks have associated websites and additional online material which will be of benefit. These are the most recent editions of two long-standing texts.

Sommerville, I. *Software engineering*. (Harlow: Pearson Education, 2010) ninth edition [ISBN 9780137035151]; www.cs.st-andrews.ac.uk/~ifs/Books/SE9/, http://catalogue.pearsoned.co.uk/catalog/academic/product/0,1144,0137053460-NTE,00.html

Pressman, R.S. *Software engineering: a practitioner's approach*. (New York: McGraw Hill, 2010) seventh edition [ISBN 9780071267823 (pbk); 9780073375977 (hbk)]; www.rspa.com/sepa7e.html, http://highered.mcgraw-hill.com/sites/0073375977/information_center_view0; www.mhprofessional.com/product.php?isbn=0073523291

Recommended reading

This course is structured in accordance with the methodology known as the Rational Unified Process ('Rational' here being a reference to the company who developed it) and you are advised to consider acquiring Philippe Kruchten's book which extends the explanation of the Rational Unified Process (RUP), which is given in the Essential reading textbooks. There are descriptions of generic project management workflows (such as progress monitoring) as well as specific activities (such as supporting the software development environment) in Kruchten, which are missing from both Essential reading textbooks.

Kruchten, P. *The rational unified process – an introduction*. (Boston: Addison Wesley Professional, 2004) third edition [ISBN 9780321197702]; www.pearsonhighered.com/educator/product/Rational-Unified-Process-The-An-Introduction/9780321197702.page

Cornerstones of SE project management

To complete this course successfully, you will also need to understand and apply some key principles and address issues that are specifically relevant to project managers responsible for SE projects. Some management activities – and certain interactions and interdependencies between activities – are not addressed directly by the recommended textbooks. This subject guide highlights what we call the '**cornerstones**' of SE project management – the parts of a structure that hold everything else together. These cornerstones are listed in an appendix at the end of the guide.

CORNERSTONE 1

Project management is a creative activity. It requires a combination of knowledge, insights and skills in order to deliver the required outcomes for each unique project. If you do not take into account the specific challenges and rely instead on a single standard approach, you are not a project manager, you are a project administrator.

Further reading material

Other reading material is suggested in specific chapters, but the following are more broadly relevant. All textbooks are listed in the Bibliography in the Appendices section of this subject guide and in the *Computing Extended Booklist* for this course, which can be found on the virtual learning environment (VLE). Certain terminology used will be explained later in the subject guide or in the 'List of acronyms' at the end of the preface.

PMI, A guide to the Project Management Body of Knowledge. (*PMBOK® guide*) (Newtown Square, PA: Project Management Institute, 2013) fifth edition [ISBN 9781935589679]; www.pmi.org/en/PMBOK-Guide-and-Standards/Standards-Library-of-PMI-Global-Standards.aspx. This guidebook from a standards organisation contains comprehensive and well-structured descriptions of the relevant management concepts.

Highsmith, J. *Agile project management: creating innovative products*. (Boston: Addison Wesley Professional, 2010) second edition [ISBN 9780321658395]; www.pearsonhighered.com/educator/product/Agile-Project-Management-Creating-Innovative-Products/9780321658395.page This author analyses agility as an approach to project management and this is not just a guide to managing agile projects.

Cockburn, A. *Agile software development: the cooperative game*. (Boston: Addison Wesley Professional, 2006) second edition [ISBN 9780321482754]; www.pearsonhighered.com/educator/product/Agile-Software-Development-The-Cooperative-Game/9780321482754.page Highsmith's collaborator provides further insights into the philosophy and application of agile methods. Be warned that locating issues in Highsmith or Cockburn is sometimes made more difficult by a predilection for colloquial and polemical terminology such as 'argh-minutes' and '…agile teams don't need plans'.

Beyer, H. *User-centred agile methods. (Synthesis lectures on human-centred informatics)*. (Morgan & Claypool Publishers, 2010) [ISBN 9781608453726].

Lund, A. *User experience management: essential skills for leading effective UX teams*. (Morgan Kauffmann, 2011) [ISBN 9780123854964].

Additional resources

1. **Archive material** (papers, videos and adjunct proceedings) of SE-related conferences:

   The ACM SIGCHI User Interface and Software Technology (UIST) archive (http://uist.org/archive), contains material presented at the conferences from 1988–2013, as well as interviews with early UIST pioneers.

   The major SE conferences series have many online resources, along with the conference papers. Look for those such as: Requirements Engineering, International Conference on Software Engineering (ICSE), Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), IFIP – World Computer Congress.

2. **Case studies** (such as those presented by the Essential reading textbook authors):
   - Sommerville: www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/index.html
   - Pressman and Lowe: www.SafeHomeAssured.com
3. **Video material** in repositories (of which there are many), such as:
   - Academic Earth: http://academicearth.org/subjects/computer-science
   - Google tech talks: http://research.google.com/video.html
   - VideoLectures.net: http://videolectures.net/Top/Computer%5FScience

Note: Unless otherwise stated, all websites in this subject guide were accessed in July 2013. We cannot guarantee, however, that they will stay current and you may need to perform an internet search to find the relevant pages.

Detailed reading references in this subject guide refer to the editions of the set textbooks listed above. New editions of one or more of these textbooks may have been published by the time you study this course. You can use a more recent edition of any of the books; use the detailed chapter and section headings and the index to identify relevant readings. Also check the VLE regularly for updated guidance on readings. Most authors provide links and 'Changes tables' or 'Correspondences' to help you migrate from earlier versions to the most recent updated versions of their textbooks. For the latest textbooks, web support is almost always available and, sometimes, there is an accompanying CD.

Diagrams

Note that colour versions of some of the diagrams in the subject guide are available in the electronic version; you may find them easier to read in this format.

Learning objectives

You should, at the end of the course, be able to draw upon appropriate techniques and have sufficient background knowledge to ensure good software engineering

management practices. You should gain the ability to select tools and methodologies that are fit for specific purposes, rather than encouraging a 'one size fits all' approach.

Learning outcomes for the subject guide

Having completed this subject guide, including the Essential reading and activities, you should be able to:

- demonstrate that you have acquired a structured view of the overall process of software development, with greater understanding of its main phases, specific notations and the specification and development tools employed
- show that you have developed a broader understanding of software engineering as a discipline, recognising its relationship and interactions with other computing and design disciplines
- demonstrate the ability to structure a problem into natural components and evaluate critically a piece of analysis and development work
- demonstrate your understanding of the implications of computer and network architectures for system-level design and development
- demonstrate that you understand how to identify, select and apply appropriate methods and tools for the development of solutions to specific real-world problems
- show that you can explain what it means to work effectively as part of a software development project team following a recognised development methodology, including the need to communicate effectively through written, oral and other forms of technical presentation.

Suggested study time

The Student Handbook states the following: 'To be able to gain the most benefit from the programme, and hence do well in the examinations, it is likely that you will have to spend at least 300 hours studying for each full course, though you are likely to benefit from spending up to twice this time'. Note that this subject is a half unit.

It is suggested that a chapter of the subject guide be read in detail, and immediate notes taken, each week. At the same time, the associated readings and web-based material should be reviewed. It is advisable to attempt the practice questions and to access relevant case study or video material when studying a particular topic. Some of these may be design exercises involving paper and pen prototyping, so sufficient time should be set aside, depending on how many are chosen. Revision should take place over a number of weeks before the examination, and practice examination questions undertaken on a timed basis.

Assessment

Important: The information and advice given are based on the examination structure used at the time this guide was written. Please note that subject guides may be used for several years. Because of this we strongly advise you to always check the current Regulations for relevant information about the examination. You should also carefully check the rubric/instructions on the paper you actually sit and follow those instructions.

The course is assessed by an unseen written examination consisting of three (out of a choice of five) questions. Guidance on how to prepare for examination questions is given in the next section.

There are also separate coursework assignment(s). Please note that the coursework assignments will **not** be the same as those that have previously been set for **CO3314 Software engineering management** and will change each year.

Coursework guidance

A detailed statement of what is required for your coursework assignments will be provided, with the marking scheme clearly indicated and suitable references given. Coursework assignments will be based on a prepared case study and will test the understanding and application of the techniques of software engineering project management covered in the syllabus. You should be able to structure a problem into natural components and to critically evaluate your own analysis and development work, presenting your material and arguments in an effective and informative manner.

The following guidance will help you to produce the appropriate standard of coursework:

- You should write in a report or essay format – not in note or bullet-point form – with a defined structure as detailed in the coursework assignment instructions. You do not need to restate the question asked or provide a table of contents, an index or a coversheet, or any extra information or appendices, and you should attempt to present your work both clearly and concisely. However, very short submissions are unacceptable.
- The structure, clarity and organisation of your work will be assessed and some marks may be awarded for it. Your submission must be well-presented in a coherent and logical fashion. It should be spell- and grammar-checked and you should structure it so that you have both a clear introduction and a conclusion. A concluding section is a required part of your coursework submission.
- You should include relevant diagrams, drawings or illustrations as graphics and screenshots. Note that these will have an impact on readability and presentation values.
- You must provide a References section at the end of your work, showing the books, articles and websites that you have referenced and consulted. All books cited, reports referred to and any material used (including all online resources) must be referenced. Students who do not provide references, or cite only the course textbook and subject guide, will be marked down.
- Websites should be referenced by the date of access and a complete and correct URL (generic site names are not acceptable). Other references should be in a standard format (namely, author names (correctly spelled, and with the correct last and first names or initials), year of publication, title, publisher, actual page numbers referenced). For a useful guide to referencing procedure, look, for example, at: http://education.exeter.ac.uk/dll/studyskills/harvard_referencing.htm There are also guidelines for referencing in the **CO3320 Project** guide. The submitted assignment must be your own work and must not duplicate another's work. Copying, plagiarism and unaccredited and wholesale reproduction of material from textbooks or from any online source are unacceptable. Any such work will be severely penalised or rejected. Any text that is not your own words and which is taken from any source must be placed in quotation marks and the source identified correctly in the References section. Failure to do so will incur serious penalties.
- Do try to be selective and critical in your choice of material and be extremely careful about the validity of information on internet sites and web sources. Citing and copying from Wikipedia and other encyclopaedic sources, for example, is not appropriate for a coursework submission. Note that personal or company-sponsored blogs, social networking material (such as Twitter or Facebook) or comments on any other type of posted transitory material cannot be used as references for academic work. Be aware that many information sites are really commercial advertising, or simply reproduce unvalidated and unchecked material from elsewhere.

- Do check the date of all material and do not use out of date sites, sites which list student work or projects, references from commercial publishers to journal paper abstracts only, or those which are simply personal opinions, blogs or comments.

Examination guidance

Failure to take account of the following points means that questions will not be answered as well as they should be.

Take a few minutes to make a plan of each question and to gather your thoughts instead of immediately starting to write. Organise your time in the examination to allow sufficient time to read over what you have written and to make any necessary corrections.

Ensure that you fully understand the topic area of the question (which will be related to the course outline). Check that you can answer every part and subsection of the question. It is generally better to answer three questions as fully as you can, rather than, for example, some in excessive detail and others very briefly.

Ensure that you understand what type of answer is expected. Read the question carefully and answer in the way that is requested: the wording will indicate the expected kind of answer and level of detail.

Ensure that the level of detail of the answer you give corresponds to the marks for that part. Try to achieve the balance reflected in the marks indicated. Do not spend excessive time and effort on a subsection of the question that is worth only a very small percentage of the overall marks. Similarly, do not write cryptic notes or single points for a part of the question that is worth a significant percentage of the available marks.

Ensure that you do provide diagrams or examples where requested since this is part of the marking scheme for that question. All figures and diagrams should be clearly labelled and described. Use tables and lists where appropriate – for example, in a question which asks you to contrast two approaches or itemise the differences between two techniques.

Do not waste time by:

- providing unnecessary diagrams where this is not required
- restating the question in your own words or repeating the question text
- answering one question in great detail at the expense of others
- repeating details from one part of the question in another part.

List of acronyms

Below is a list of acronyms which are relevant to this subject. You may wish to add to this list as you study.

ATAM	Architectural Trade-off Analysis Method
AUP	Agile Unified Process
CASE	Computer-aided Software Engineering
CMMI	Capability Maturity Model Integration
COCOMO	COnstructive COst MOdel
COTS	Commercial Off The Shelf (software) components
CSR	Critical Software Reviews
CUPRIMDSO	Capability, Usability, Performance, Reliability, Install-ability, Maintainability, Documentation, Serviceability and Overall
DFD	Data Flow Diagram

DSDM®	Dynamic Systems Development Method
ERP	Enterprise Resource Planning
FPA	Function Point Analysis
FSAR	Final Software Acceptance Review
FURPS	Functionality, Usability, Reliability, Performance, Supportability
GQM	Goals, Questions, Metrics
IDE	Integrated Design Environment
IPM	Integrated Project Management (CMMI Level 3)
ISO/IEC	International Standards Organisation/International Electrotechnical Commission
ITIL	Information Technology Infrastructure Library
LASCAD	London Ambulance Service Computer-Aided Despatch system
LOC	Lines of Code
MVC	Model, View, Controller architecture
OGC	Office of Government Commerce (UK Government)
OO	Object Oriented
PDCA	'Plan Do Check Act'
PERT	Program Evaluation and Review Technique
PFD	Product Flow Diagram
PID	Project Initiation Documentation
PMBOK®	Project Management Body of Knowledge
PMC	Project Monitoring and Control (CMMI Level 3)
PMI	Project Management Institute
PP	Project Planning (CMMI Level 3)
PRINCE2	PRojects IN Controlled Environments
PSR	Preliminary Software Review
QA	Quality Assurance
QM	Quality Management
QMS	Quality Management Strategy
QPM	Quantitative Project Management (CMMI Level 3)
RBS	Risk Breakdown Structure
REQM	Requirements Management (CMMI Level 3)
RMMM	Risk Mitigation, Monitoring and Management
RSKM	Risk Management (CMMI Level 3)
RUP	Rational Unified Process
SAM	Supplier Agreement Management (CMMI Level 3)
SEI	Software Engineering Institute (Carnegie Mellon University, USA).
SOA	Service-Oriented Architecture
SRS	Software Requirements Specifications
SWOT	Strengths, Weaknesses, Opportunities and Threats

UML	Unified Modeling Language
UP	Unified Process
V & V	Verification and Validation
WBS	Work Breakdown Structure
XP	eXtreme Programming

Introduction: the need for software engineering

References cited testing

Cook, S. 'What the lessons learned from large, complex, technical projects tell us about the art of Systems Engineering', *Proceedings of INCOSE 2000 Annual Symposium, Minneapolis*, pp.723–30.

Fagan, M.E. 'Design and code inspections to reduce errors in program development', *IBM Systems Journal* 15(3) 1970, pp.182–211. (Available from IBM under Reprint Order No. G321-5433.)

Hosier, W.A. 'Pitfalls and safeguards in real-time digital systems with emphasis on programming', *IRE Transactions on Engineering Management* EM-8(2) 1961, pp.99–115.

Liskov, B. and S. Zilles 'Programming with abstract data types', *Proceedings of ACM SIGPLAN Symposium*, 1974, pp.50–59.

Naur, P. and B. Randell (eds) 'Software engineering. Report on a conference held in Garmisch, Oct. 1968, sponsored by NATO'.

Parnas, D.L. 'Abstract types defined as classes of variables', *ACM SIGPLAN Notices II* 2 1976, pp.149–54.

Royce, W.W. 'Managing the development of large software systems: concepts and techniques', *Proceedings of the IEEE* 1970.

Wirth, N. (2008) 'A brief history of software engineering'; www.inf.ethz.ch/personal/wirth/Articles/Miscellaneous/IEEE-Annals.pdf

> 'It is unfortunate that people dealing with computers often have little interest in the history of their subject. As a result, many concepts and ideas are propagated and advertised as being new, which existed decades ago, perhaps under a different terminology.' (Wirth, 2008)

Overview

The term 'Software Engineering' (SE) comes from recognition that formal methods and more sophisticated tools and techniques were needed to address the challenge of '…the construction of systems by large groups of people' (Naur and Randell, 1968), specifically real-time systems.

These 'frequently present unusually strict requirements for speed, capacity, reliability and compatibility, together with the need for a carefully designed stored program' (Hosier, 1961). These features, particularly the last, have implications that are not always foreseen by management.

Stephen Cook treats software engineering as a subset of the broader discipline of systems engineering, and sets the tone for this course when he advises the reader that:

> 'Systems engineering should not be considered merely a set of procedures to be followed. Rather, it is a … way of thinking that encompasses a set of competencies and a set of processes that can be invoked as needed, each of which can be achieved through a range of methods. An important aspect of systems engineering is the selection and tailoring of the processes to suit each project.' (Cook, 2000)

This reflects the need to balance between sometimes incompatible criteria such as functionality and reliability. Martin Rinard wrote:

> 'This combination is challenging because of the inherent tension between these two properties. Increasing the functionality increases the complexity, which in turn

> increases the difficulty of ensuring that the software executes reliably without errors.'
> (See: http://people.csail.mit.edu/rinard/paper/oopsla03.acceptability.pdf)

Software engineering covers the entire product lifecycle, from definition of requirements, through the design and development phase, all-important testing and delivery of long-term stakeholder benefits.

Two essential weaknesses of the early 'top-down' methodologies were the difficulty in obtaining complete, correct and appropriate specifications of what the user actually needs (and here we are talking both of 'business' users and the end-users who are typically employees or customers of the 'business') and the significant effort in development and testing that is inevitable if bespoke solutions are created as part of every project.

In most areas of engineering, a top-down approach reflects the predictable nature of the interactions between modules or sub-systems, whereas many of the interactions between software modules are largely unpredictable and only emerge during final integration. This has led to adoption of a range of iterative development processes that recognise the importance of user-centred design and prototyping as a way of configuring and deploying validated and robust components as part of an overall system.

In his commentary on standards for software lifecycle processes, Raghu Singh of the Federal Aviation Administration recognises that the *ISO/IEC 12207* development activities (see: www.iso.org/iso/catalogue_detail?csnumber=43447):

> '…may be iterated and overlapped, or may be recursed to offset any implied or default Waterfall sequence. All the tasks in an activity need not be completed in the first or any given iteration, but these tasks should have been completed as the final iteration comes to an end. These activities and tasks may be used to construct one or more developmental models (such as, the Waterfall, incremental, evolutionary, the Spiral, or other, or a combination of these) for a project or an organisation.'
> (See: www.abelia.com/docs/12207cpt.pdf)

The critical success factor is the way in which specialists performing the roles defined in the standard are integrated into a team operating within a common framework so that their activities are manageable. SE project management is a distinct specialism within the team and only the smallest of projects will not need a recognised team leader or project manager.

Unsurprisingly, a 2007 survey into project management practices (See: www.pwcprojects.co/Documentos/BRO Boosting Business Performance DC-07-1104-A v8[1].pdf) found that:

- Use of project management methodologies is widespread. Organisations that do not have a project management methodology reported lower-performing projects.
- Use of project management software positively impacts project performance.
- Project management certification and project performance are clearly connected.
- Projects with poor management are more likely to suffer from problems like bad estimates/missed deadlines, scope changes and insufficient resources – the top three reasons for project failure cited by 50 per cent of those interviewed.

CORNERSTONE 2

Project management is about controlling the strengths and weaknesses of the project team (and enterprise) and the opportunities and threats that can arise and lead to issues for the project, the 'internal' and 'external' factors of a SWOT analysis.

Software that is being produced today typically falls into one of three categories:

1. Software as a **component**, such as the fuel management system embedded in your car.

2. Software as a **tool**, such as an '**I**ntegrated **D**evelopment **E**nvironment' (**IDE**) in which other software is developed and maintained.
3. Software as a **process**, no longer simply deployed as a tool to help organisations do the things they do, but a result of rethinking what an organisation does and how it does it.

The significance of the last of these is that the organisation may be gambling its future in a game where it does not understand the rules, let alone have prior experience. This may be the case whether the project is seen as being technology-driven (and managed by someone with little or no experience of the human-resource issues involved) or organisation development-driven (and managed by someone without SE experience).

In his book *The change equation*, Peter Duschinsky argues that projects fail when '… the complexity of the project exceeds the capability of the organisation to cope with the changes needed' (See: http://wdn.ipublishcentral.net/management_books_2000/viewinside/207741229298377), where that capability necessarily includes the ability to manage both social/cultural factors and technical ones. In the subsequent paper with the same title (see: www.irma-international.org/viewtitle/43568) he delivers the message that only a quarter of these change projects succeed fully, while another quarter fail to deliver any benefit whatsoever.

The SE manager has to achieve a balance between the reliability of the end result and its general acceptability for the client (compliance with functional and non-functional requirements), its ease of maintenance and its efficient use of resources. There are three major sources of risk in software development which make the job more difficult:

- Projects are getting larger and more complex and demand more sophisticated tools and methodologies, especially in the context of business change programs.
- Development teams are also getting larger. Roles and responsibilities, especially those requiring people-oriented 'soft skills', need to be well-defined and understood.
- There are several software development lifecycle models and there is no 'one size fits all' solution.

The subject guide and the Rational Unified Process (RUP)

You will see that this subject guide does not simply follow the structure of the Essential reading textbooks. Rather, we have divided it into four parts, each corresponding to one of the phases of IBM's '**R**ational **U**nified **P**rocess' which is goal-driven, not activity-driven or plan-driven. It views each project phase as the journey from one milestone state to another. Each part consists of chapters addressing the key software engineering concepts that have to be applied in order to reach the required exit condition for that phase and each chapter includes discussion of associated project management issues.

Highsmith talks of a framework of project governance consisting of the **Concept phase, Expansion phase**, **Extension phase** and **Deployment phase**. Progression between phases is controlled by milestones called 'phase gates' which segment the phases and, importantly, segment the operational agile cycles of envision, explore and deploy. Cockburn says in his book that 'RUP is not incompatible with the principles' of agile methods. Indeed, there is an 'Agile Unified Process' (AUP) developed by Scott Ambler. (See: www.ambysoft.com/unifiedprocess/agileUP.html)

There is a clear relationship between Highsmith's governance framework and the RUP. RUP and AUP are sufficiently similar for us to be able to refer to '**U**nified **P**rocess' **(UP)** phases, UP milestones and UP disciplines compatible with agile and more traditional approaches.

Software engineering activities and workflows typically span several UP phases and are addressed in the subject guide at the most relevant time. For example, the 'mission' for testing is established in the inception phase but that mission is only achieved through

execution of component and system testing in later phases. Discussion of testing is therefore deferred until the chapter on the **Construction phase**.

Each chapter is grouped as part of the four '**phases**', except for the final chapter, which contains case studies.

Part 1: Inception phase – scoping and justifying a project

By the end of the first part, you should understand the software development process, project planning and the information needed to enable a project to be described in terms of scope and the approach to satisfying requirements.

- **Chapter 1** reviews the fundamental elements of a software development process and the way in which a project manager delivers project results. It also introduces the key features of the Unified Process.
- **Chapter 2** reviews the role of the business case and system requirements as the basis for the project plan.
- **Chapter 3** reviews techniques for project cost estimation and scheduling, with specific reference to planning within iterative and agile approaches.

Part 2: Elaboration phase – defining a response to stakeholder needs

By the end of this part, you should understand how to produce a robust implementation plan based on a stable architecture and appropriate approaches to risk and quality management.

- **Chapter 4** reviews approaches to risk identification and mitigation.
- **Chapter 5** reviews techniques for evaluating architectural options and design and modelling of system architecture.
- **Chapter 6** reviews the project manager's responsibilities for detailed planning, people and risk management and the project development environment, including configuration management.

Part 3: Construction phase – managing the provision of the functionality agreed

By the end of this part, you should understand what is involved in delivering a version of a stable system that can be deployed with real users.

- **Chapter 7** looks at the implementation workflow, including the use of design patterns, design for re-use and management of agile teams.
- **Chapter 8** reviews approaches to Quality Management (QM) and testing.

Part 4: Transition phase – producing a product release

By the end of this part, you should understand how to manage entry into the post-release maintenance cycle and authorise the start of a new development cycle if required.

- **Chapter 9** addresses deployment issues, including evolution, process improvement and management of expectations and change.

Part 5: Review and revision

The final chapter provides a look back on key lessons learned in the form of case studies (**Chapter 10**). There are various appendices to aid revision, in the form of revision topics, and the full list of cornerstones, as well as a Sample examination paper and an outline marking schema (Appendix 6).

Each chapter starts with a set of **Learning outcomes**, detailing the learning objectives for that topic and a brief Overview section. Also shown at the beginning of each chapter are details of the **Essential reading** required for full comprehension of the topic – normally a chapter in the designated textbooks – and any **Additional** or **Further reading**. These additional references have been chosen to enhance knowledge and expand coverage and

are usually books, journal articles or free-access websites. The **Bibliography** (Appendix 1) has the full citable details (and, where available, the online link to the publisher's website page for that book) of all the textbooks and print sources referred to in the subject guide.

There are also **References cited** sections where appropriate. These are usually citation details for quotations or for particular articles or websites referred to. These additional links and citations can be followed up, if wished, but it is not mandatory: they do not necessarily appear in the Bibliography.

Each chapter concludes with a summary of the main points as an *aide-memoir*. **Reminder of learning outcomes** and **Test your knowledge and understanding** sections are given at the end of each chapter to help with examination revision and to test understanding of the concepts covered in that chapter. Local tutors in institutions may also wish to use such suggested topics to lead discussion groups; therefore sample answers are not always provided.

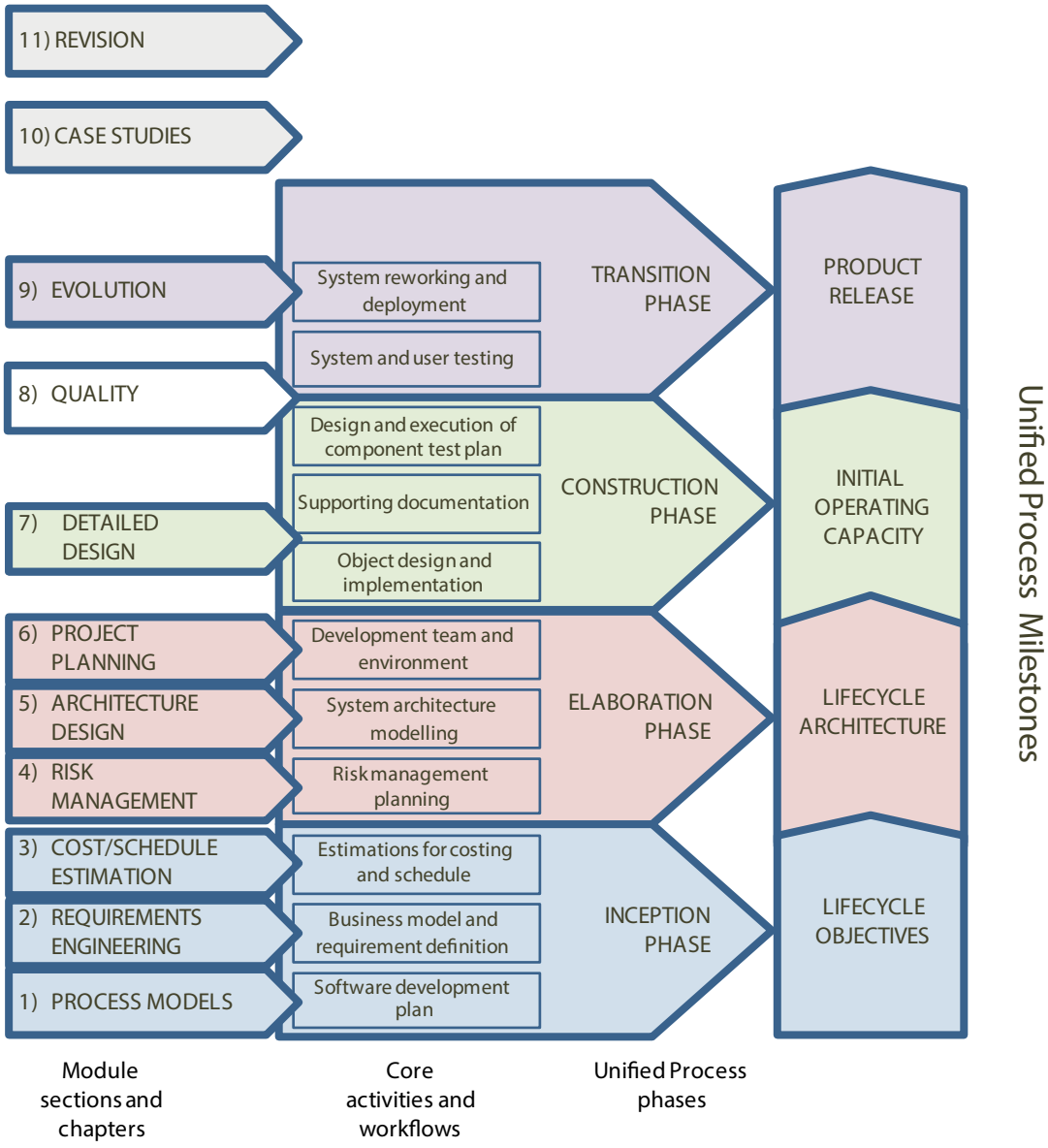


Figure 0.1: Architecture of the SE Project Management module and its relationship to the RUP.

A **Sample examination paper**, together with an outline marking scheme, can be found at the end of the subject guide.

A **List of acronyms**, expanding all the acronyms used in the subject guide, can be found in the Preface to the subject guide.

Summary of Introduction: the need for software engineering

- Software engineering is a profession, an engineering discipline.
- Software engineering covers the entire development process and applies to development of all kinds of software system although tools and techniques are rarely universal.

Test your knowledge and understanding: seven important references

Investigate the following milestones leading to today's software engineering environment.

1. The concept of abstraction developed in the mid-1970s, allowing the details of the implementation of a 'module' to be hidden behind a published 'interface' (Liskoy and Zilles, 1974; Parnas, 1976) leading to object-oriented system design with components (objects) that have unique properties and behaviours.
2. Recognition that the engineering lifecycle concept also applied to software engineering, which led in the mid-1970s to widespread use of the waterfall model (see: Royce, 1970), formal software inspection methods for each step (Fagan, 1976), comparable to those used to validate electrical circuit diagrams, architectural scale plans and the like. The lifecycle approach was formalised as a standard in 1995. (See: ISO/IEC 12207, Information Technology – Software life cycle processes (see www.iso.org/iso/catalogue_detail?csnumber=43447)
3. Identification in 1995 of the major factors that cause software projects to fail and key ingredients that can reduce project failure, in The Standish Group's first 'CHAOS report'. (See: www.spinroot.com/spin/Doc/course/Standish_Survey.htm)
4. The publication of the 'Manifesto for Agile Software Development' (see: http://agilemanifesto.org/principles.html) by the Agile Alliance in 2001.
5. The UK government's green paper on the development of 'e-government' services (HM Treasury, 2003) (see: www.publicnet.co.uk/abstracts/2003/10/20/measuring-the-expected-benefits-of-e-government), which differentiated between four types of 'service re-engineering' from a simple 'electronic customer interface' to 'end-to-end-process transformation'.
6. Growing recognition of strategic priorities that reach beyond 'on time and on budget' such as stakeholder satisfaction, delivery of benefits, quality of the result and return on investment (see: www.pwcprojects.co/Documentos/BRO Boosting Business Performance DC-07-1104-A v8[1].pdf), reported in 2007.
7. Widespread ignorance that information sent via cloud email could be stored by a service provider in another country with 38 per cent of those surveyed in 2011 sending valuable or confidential personal information via cloud email or messaging services (61 per cent of 18–24 year-olds, compared to 24 per cent of the 55+ age group). (See: www.rackspace.co.uk/uploads/involve/user_all/generation_cloud.pdf)

Part 1: Inception phase

Chapter 1: Software processes

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- understand the concept of a process model as a way of describing a software development project and differentiate between common approaches such as waterfall, iterative and agile
- identify the key artefacts produced during an iteration of the software development process
- understand the role of the project manager in delivering those artefacts
- understand the ways in which the development process can affect quality.

Essential reading

Sommerville	Pressman
Chapter 2: Software processes	Chapter 2: Process models

Further reading

Ambler, S. 'A Manager's Introduction to the Rational Unified Process' (2005); www.ambysoft.com/unifiedprocess/rupIntroduction.html

Boehm, B.W. 'A spiral model of software development and enhancement', *IEEE Computer*, 21(5) 1988, pp.61–72.

Royce, W.W. 'Managing the development of large software systems: concepts and techniques', *Proceedings of IEEE WESTCON* (Los Angeles, 1970), pp.1–9; http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf

Scacchi, W. 'Process models in software engineering', in J.J. Marciniak (ed.) *Encyclopedia of software engineering*. (New York: John Wiley and Sons, Inc., 2001) second edition Volume 2 [ISBN 9780471210078].

References cited

Booch, G., J. Rumbaugh and I. Jacobson *Unified modeling language user guide*. (New York: Addison Wesley Professional, 2005) second edition [ISBN 9780321267979].

Highsmith J. *Agile project management*. (Boston: Addison Wesley, 2009) second edition [ISBN 9780321658395].

PMI, *A guide to the project management body of knowledge. (PMBOK® guide)* (Newtown Square, PA: Project Management Institute, 2013) fifth edition [ISBN 9781935589679]; www.pmi.org/en/PMBOK-Guide-and-Standards/Standards-Library-of-PMI-Global-Standards.aspx

Sutherland, J. 'Agile development: lessons learned from the first Scrum' (2004–10); www.scrumalliance.org/resources/35

Overview

The development of a bespoke software system can be thought of as a process with three 'actors':

- A **system user**, whose requirements determine the intended scope and behaviour of the system.
- A **team of system engineers** who translate that specification into a design for the system.
- The **system** created from that design.

From the customer's perspective, the project team should be able to take the stated needs and implement and maintain the system through until the end of its useful life. In an ideal world, the process continues smoothly, perhaps through several iterations (versions) over time.

However, as all software developers know to their cost, customer needs change and the real world is rarely ideal. This leads to mismatches between what was intended and what actually occurred, which impacts on one or more of the three actors. Errors introduced and not trapped locally can have costly repercussions and steps need to be taken to limit the impact.

An error in the software engineer's understanding of requirements will affect the ability to address the system user's needs and result in reworking of the 'specify' activity.

A mistake in the creation of the system designed to meet those needs will result in reworking of the 'design' activity.

A failure on the part of the client to plan and support the deployment of the system will impact on the performance of the system and its users and reworking of the 'deploy' phase.

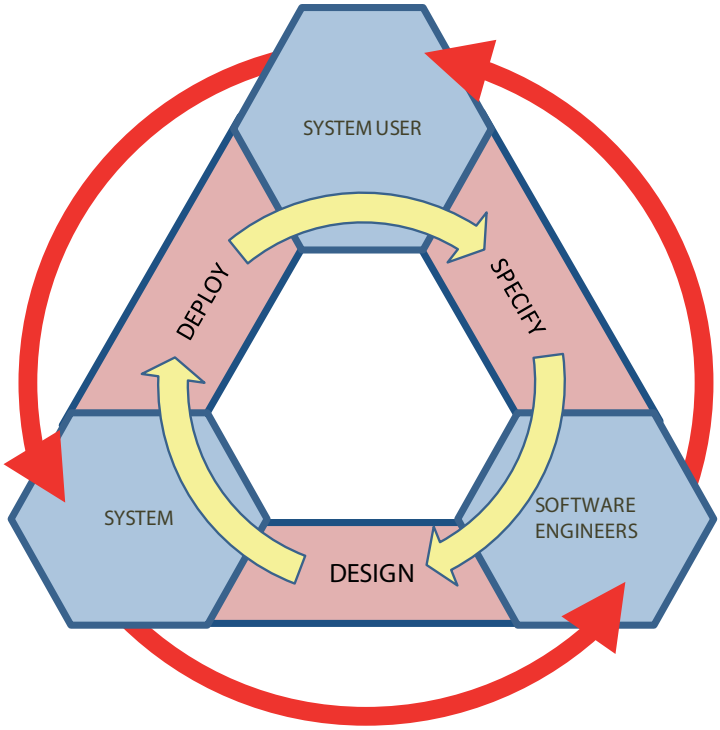


Figure 1.1: Failure in the development process.

Process modelling

Both recommended textbooks review and compare the most common types of software development process models in use today. There are useful process models that rely on iteration, such as incremental development, prototyping and incremental delivery. Process models can be activity-driven (such as the simple waterfall), plan-driven (the German 'V-Modell' and Boehm's 'Spiral' are both predicated on the existence of an overall project plan) or goal-driven (such as the UP's milestones, or the targets that drive the SCRUM technique (see Sutherland, 2004–10)).

CORNERSTONE 3

A process model is a high-level and incomplete description of the way a project is controlled and how it evolves. It introduces interfaces between actors and workflows and transitions between activities. Your choice of model does not fundamentally affect **what** you are going to do; customers still need to know that their requirements are the reference point for design and code still needs to be tested before and after integration. What the model allows you and your team to do is focus on the important transitions during the project.

Key concepts: phases and iteration, milestones and artefacts

Phases

The classical linear process model is the waterfall (Royce, 1970) shown in Figure 1.2. It begins with requirements which are the basis for specification. The expectation is that each activity in the process is visited once and its output is fit for purpose.

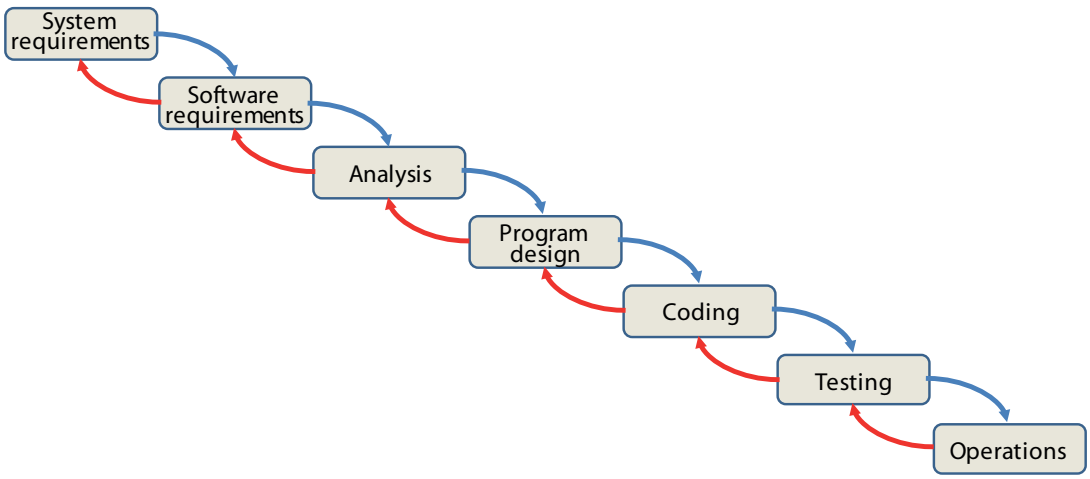


Figure 1.2: The classical waterfall model.

Figure 1.3 takes the waterfall testing phase and sub-divides it in accordance with the V-Modell approach to verification and validation. Localised testing is efficient but errors that remain undetected until acceptance testing still require reworking of everything that is a response to requirements. Quality issues are addressed in Chapter 8 of the subject guide.

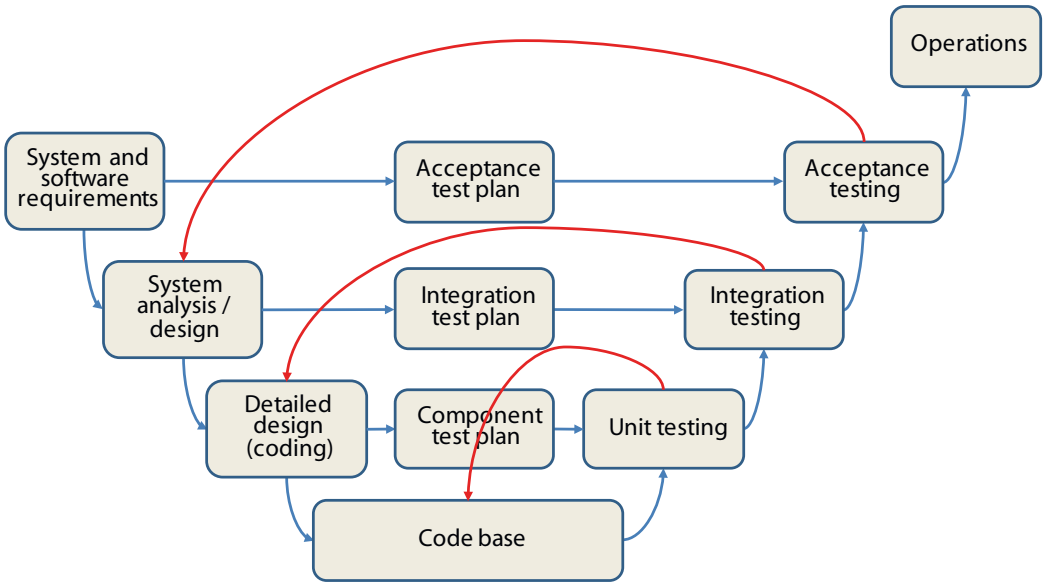


Figure 1.3: Waterfall model modified to V-Modell format.

Milestones

Royce's 1970 paper is best known for its description of the linear process we call the classical waterfall. Royce also proposed formal review-points or milestones and test-plans which can be drawn up once the final software review has validated the program design. He suggested a '**P**reliminary **S**oftware **R**eview' (**PSR**) between supplier and customer before formally moving on to the analysis phase, based on a preliminary program design which acts as a form of prototype to guide design and coding. As work develops, '**C**ritical **S**oftware **R**eviews' (**CSR**s) can be scheduled with the customer, to further inform the coding process. Finally, the transition from testing to operational deployment is controlled by a '**F**inal **S**oftware **A**cceptance **R**eview' (**FSAR**) between the testing and operations phases.

The goal is to minimise the wasted work that results from re-iteration of previous phases. The PSR, CSR and FSAR are not simply reviews of what has happened already, they prevent problems being fed forward into the next phase. In this respect, they are akin to quality gates or milestones. The reviews have a significant effect on the way phases are re-iterated.

Since a CSR validates that the program design is a true expression of the output of the analysis phase, there is no point in looking at the program design for the source of an error found during the coding phase. It must have been present at the end of analysis – and perhaps even further back.

The complete architecture proposed by Royce in 1970 – with milestones, prototypes and code reviews – would look something like the iterated linear process in Figure 1.4.
The topic of project planning is covered in detail in Chapter 6 of the subject guide.

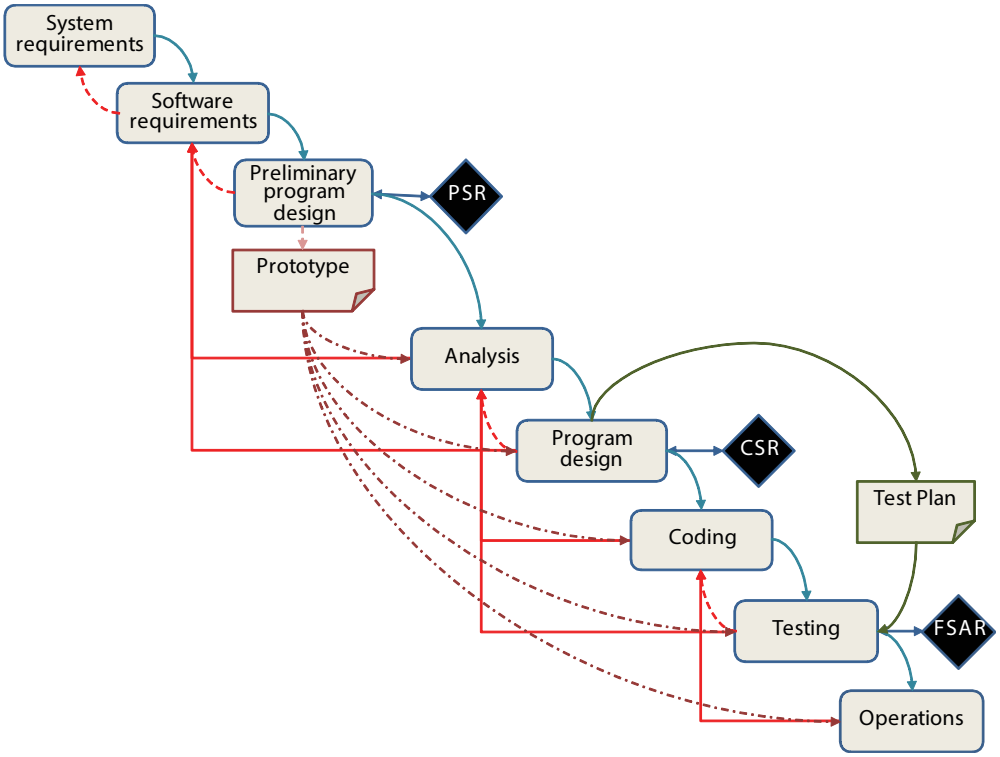


Figure 1.4: Controlling the waterfall, after Royce.

The UP approach

The three diagrams above are all expressions of the way a development project moves through discrete phases. Normally, these phases encapsulate specific workflows such as 'requirements engineering' and 'design'. The UP takes a different approach, separating the lifecycle phases from workflow disciplines. There are four reasons for this:

1. A workflow may extend across several phases. Testing, for example, begins during the elaboration phase, following the V-Modell approach of agreeing a test-plan before software is produced, and continues until the end of acceptance testing in the transition phase and possibly beyond that into planning for the next iteration of the UP lifecycle.
2. Several different workflows or disciplines may be required to fulfil all of the requirements for completing the phase. For example, coding and testing are concurrent or interleaved tasks within the construction phase but require different disciplines.
3. The same (or very similar) tasks may need to be performed several times within a single iteration of a phase. This occurs during incremental development and rapid prototyping.
4. Where the complete process is repeated (for a second release, for example), the planning and justification tasks may take advantage of the outputs from the previous iteration.

The term 'goal-driven' describes the philosophy of the UP very well. Each phase ends with an identifiable moment in time at which a major state change occurs in a project. The inception phase ends when the client is able to authorise the supplier to proceed with the project. That is, the inception phase produces the evidence (including a scoping

statement, budget and resource plan, schedule and analysis of risks) on which the client makes the decision to proceed.

This is a major milestone in the evolution of the project. The others are the stabilisation of project vision and architecture (the goals of the elaboration phase), readiness to roll-out the system developed in the construction phase and the final goal: product acceptance.

Figure 1.5 shows how the linear waterfall process can be broken down into the four UP phases, each with its goal expressed as a milestone. Inception leads to the ability to define system requirements as lifecycle objectives for this iteration. Elaboration results in a lifecycle architecture, or outline design. The end of the construction phase is marked by the availability of tested components, or an initial operating capacity. The cycle repeats if necessary once the product is released. Note how the milestones reduce the need for iteration within the lifecycle compared to Figure 1.4. Unresolved issues detected at those milestone points are either addressed before moving on to the next phase or deferred until the next UP iteration.

The use of the term 'milestone' refers back to the 18th century when roadside stones marked the distance to the destination. Milestones are not a measure of the distance already travelled; they tell you how much further you have to go. In a typical project where activities are planned backwards from an agreed endpoint, a milestone is not the date at which an agreed state is reached – it is the date at which the state was planned to be reached. There is a big difference.

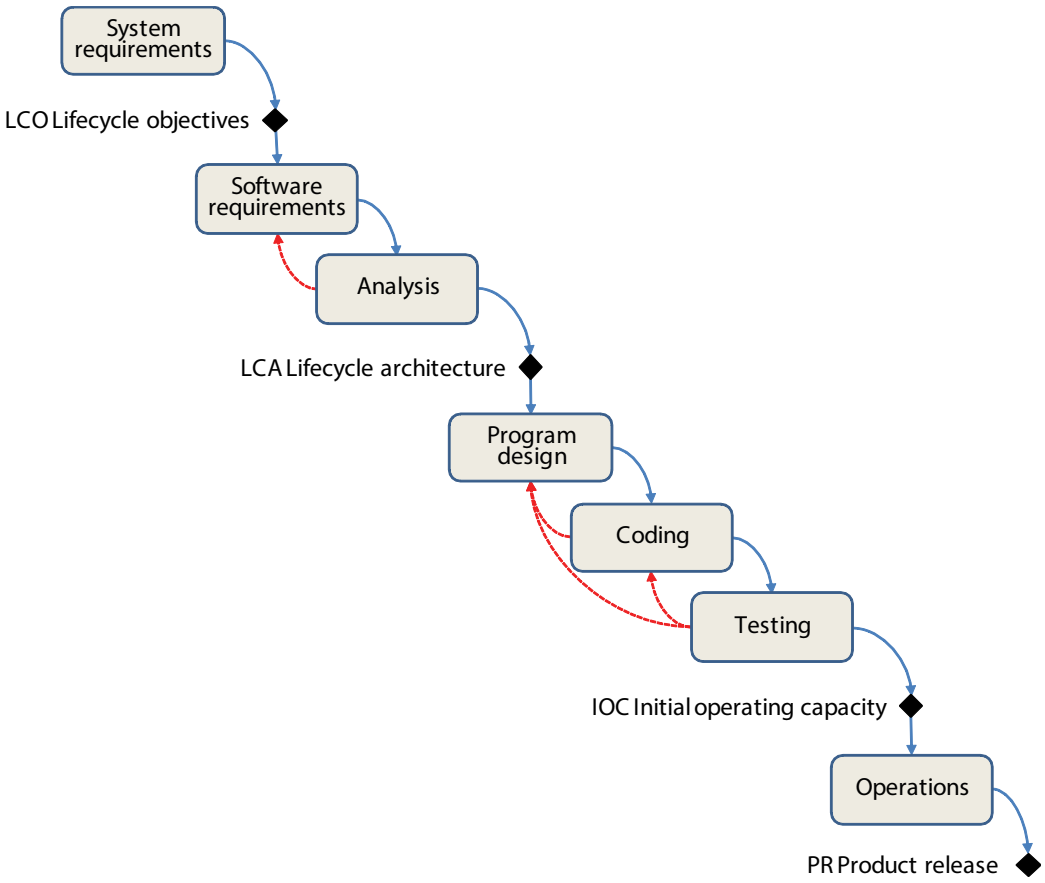


Figure 1.5: Waterfall controlled by UP milestones.

The other difference between a goal-driven and an activity-driven or plan-driven project is that the focus is on achieving results, not on doing the work that should deliver those results. Planning determines the way 'products' are produced, not the way work is organised.

CORNERSTONE 4

The purpose of a software project is to meet a client's need by delivering a software system. That system decomposes into sub-systems and components, and each component requires a number of products to be built, tested and integrated. There are two approaches that a project manager can adopt – focus on the product, or focus on the work required to deliver that product.

Artefacts

An output from the project, or '**product**', is known in the UP as an '**artefact**'. There are many definitions of artefact in different contexts, but the one we will focus on is that it is something that allows another, higher level, product to be constructed.

The artefacts that must be signed off to allow the 'Lifecycle objectives' milestone to be met at the end of the inception phase, for example, should be based on:

- project scope definition
- cost and schedule definitions
- risk identification
- project feasibility
- plans for the project (development) environment.

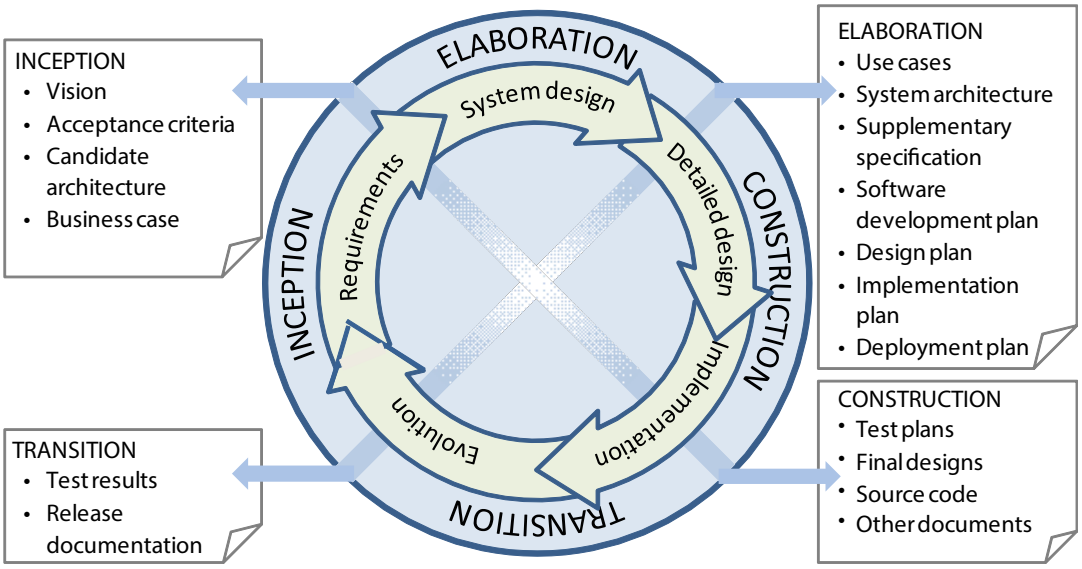


Figure 1.6: Relationships between phases, activities, milestones and artefacts.

Relationship between the UML and UP

There is a further difference between UP and the other methodologies described above and that is the adoption of a standard form of documentation – the 'Unified Modeling Language' (UML (™ Object Management Group, Inc.) (See: Booch et al., 2005).

The UML is the result of efforts to provide a standardised and easy-to-understand way of describing aspects of the structure, interactions and behaviour of a software system. It provides a diagrammatic way of modelling business processes, software requirements and architecture design in UP.

Table 1.1: Use of UML in the RUP.

Scope	UML	Purpose
Business modelling	Activity diagram	Definition of system boundaries and the flow of control between the system and its external context and environment.
	Class diagram	Identification of key business stakeholders or entities.
	Use case diagram	Definition of high-level business requirements.
	Data flow diagram	Description of business processes to give a high-level definition of system scope.
Software requirements	Use case diagram	A statement of a functional requirement (a specific task undertaken on behalf of a specific class of user).
Architectural design	Class diagram	Definition of the classes of object required by the architectural design and the relationships between them.
	Sequence diagram	A more detailed description of the interactions between the actors and the objects or components within a system.
	Component diagram	Definition of the interfaces (services) provided by a component of a system (especially a re-usable component) and the services that have to be provided by other components.
	Activity or sequence diagram	Description of the behaviour of objects in response to a stimulus.
	State diagram	Description of the way objects change state in response to a stimulus.
	Deployment diagram	Description of the deployment of software on hardware connected by middleware at run-time.

All of these models (and six other less-used diagrams) are described by Scott Ambler. (See: www.agilemodeling.com/essays/umlDiagrams.htm)

Agile methods

The principles of agile development are based in a large part on '**lean thinking**', an approach to manufacturing derived from the work of W. Edwards Deming in the 1950s. A 'lean' approach has five underlying principles, which advocates of the agile approach should recognise. We do not need to spend time here discussing the ramifications of the 2001 Agile Manifesto (see: http://agilemanifesto.org) except to characterise similarities and differences between agile and more traditional approaches based on these five principles.

1. Define the activities which add value to a product or service from the customer's perspective and aim to eliminate wasteful activities that do not add value. The agile practitioner would see the value in a brief daily staff meeting looking forward and the waste in a weekly written progress report looking back.
2. Understand how value is delivered to the customer, as an end-to-end process. Identify aspects of value that do not survive that end-to-end process and parts of the process that do not add value. The agile practitioner would see little or no

value in undertaking a comprehensive requirements analysis in areas where those requirements can change as the system evolves.

3. Work out how to remove waste from the process by concentrating on improving the flow of value and challenging ways of working that reduce the flow. The agile practitioner would see the correlation between the size of a task (such as an increment in functionality to be delivered) and the risk that it will contain errors and other forms of wasted work.
4. Learn how to respond to the customer's priorities rather than persisting with fixed ways of working. The agile practitioner would see the benefits of having a flexible and multi-skilled project team that can be configured to reduce the risk of bottlenecks when responding to the customer's requests.
5. The goal of the lean approach in manufacturing is creation of a perfect value chain through principles of continuous improvement and total quality management. Cockburn points out, in discussion of the differences between the agile approach and that of '**C**apability **M**aturity **M**odel **I**ntegration' (**CMMI**) (™ Carnegie Mellon University), that '…CMMI places optimisation of the process at the final level, Level 5. Agile teams start optimising the process immediately.'

One major difference between agile and more conventional methods is in the approach to governance. Governance is about information flows and the way they facilitate decision-making. Ambler (see: www.ambysoft.com/unifiedprocess/agileUP.html) describes delivery of incremental releases over time as 'serial in the large, iterative in the small' where senior management see a linear sequence of milestones and team members see a highly iterative set of envision, explore and deploy activities.

Highsmith (2009) discusses the need to address the concerns of senior management who are looking at a portfolio of projects from a perspective of investment and risk, on the one hand, as well as project and iteration managers at the other. His enterprise-level governance model for agile organisations is essentially based on the UP lifecycle.

The project management perspective

Focus on business case and managing risk

Royce was primarily concerned with managing the technical aspects of software development and most waterfall activities take place during the construction phase of the UP lifecycle. The waterfall starts after the project has been commissioned and ends when the system becomes operational. It does not explicitly include activities such as justification of the project business case or system deployment, evolution and retirement. As a methodology, it does not make risk planning or resource-planning explicit. There is no activity to address the identification or evaluation of options from a risk perspective or for optimising the use of resources by allowing teams to work in parallel on independent sub-goals.

All of these non-technical concerns are, however, within the scope of the UP inception, elaboration and transition phases so that UP provides an overall business context within which the project team can select an appropriate methodology for the construction phase – whether that follows a linear or iterated approach, prototyping, re-use of software objects or '**C**ommercial **O**ff **T**he **S**helf' (COTS) components, incremental development, incremental delivery or a partially automated process based on formal methods.

Some methodologies are more suited to large-scale, mission-critical, projects where risk management is a high priority. Some are appropriate when the client cannot articulate requirements for one reason or another or cannot visualise the end-product. Sometimes the most appropriate methodology depends on constraints associated with the context of the project, such as the need to migrate an operational system or a project that forms just part of an overall program of work.

One other common form of process model must also be considered, and that is Boehm's 'Spiral Model' (Boehm, 1988), described in both recommended textbooks. Here, we need only consider the essential characteristics of the Spiral.

- Since the elements in the model are addressed sequentially, one journey round the axis of the Spiral can represent one iteration of a process model (for example, the UP model based on business case acceptance, development plan acceptance, system acceptance and deployment).
- At a lower level, the journey could represent one phase of such a model (for example, the elaboration phase, which involves capturing and modelling requirements, identifying risks and ways of managing them, developing an architectural framework, identifying resource requirements and producing the development plan).
- Because the elements of the Spiral are organised concentrically round that axis, the journey can be divided into standard sub-goals, such as Boehm's objective-setting, risk assessment and reduction, development and validation followed by planning for the next iteration.
- Within a sub-goal, a standard set of activities may be required. Risk assessment and reduction typically requires analysis of the risks that were identified for this phase, prototyping or simulation to validate the analysis, benchmarking to provide a baseline for verification and so on.

Boehm's development and validation phase can be implemented, like the UP construction phase, using any valid process model. When constructing a web-app based on the '**M**odel, **V**iew, **C**ontroller' (**MVC**) architecture, for example; business rules for the model can be verified by prototyping and implemented incrementally; the view can be built by re-using existing objects and application-specific sub-systems within; the controller could be implemented and integrated using a waterfall approach.

The framework is robust enough to be used in non-prescriptive ways. For example, Boehm says 'prototype' but does not define the nature of the prototype.

Best practices

CORNERSTONE 5

A best practice in common use is the breakdown structure. A Work Breakdown Structure, for example, decomposes the project into phases, decomposes phases into activities and activities into tasks and sub-tasks. You can describe dependencies from one to another through a Work Flow Diagram (for example, a GANTT chart; named after the person who developed it, Henry Gantt). Similarly, you can represent the relationships between artefacts from a project in a Product Flow Diagram, which then defines the order in which they must be completed. You can also use breakdown structures to ensure that an analysis of factors such as risks or competences is comprehensive.

As described by Sommerville, UP embodies three views on the software engineering process.

- A dynamic model of the evolution of a project over time.
- A static description of the activities that form the components of that process model.
- A generalised description of the types of good practice which are appropriate for execution of those activities.

Best practices include suggestions for workflows and the document artefacts that will normally be needed, and their purpose/readership, content and structure.

Table 1.2: Best practice workflows and artefacts.

Workflow	Scope	UML artefacts
Business modelling	Business processes modelled in a form understood by both business and software analysts	Business use case descriptions organised into the business object model.
Requirements	Functional requirements	Use case descriptions, collectively the use case model.
	Non-functional constraints – trade-offs and decisions	Supplementary specifications.
Analysis and design (the '4+1' views, see Figure 5.1)	Static structure (logical view)	Object model (class diagram).
	Dynamic behaviour (process view)	Object model translated into sequence diagram. Objects with complex state-change behaviour may require a state diagram.
	Physical organisation of components (implementation view, optional physical views)	Object model translated into component diagram. Optional deployment diagram showing how components are deployed on distributed platform.
Implementation	Building components and sub-systems	The component model is implemented in code in agreed increments.
Testing	Component testing	Components are verified and validated against the relevant use case(s).
	System testing	The system produced by the implementation workflow is validated against the use case model and SRS.
Deployment	Acceptance testing Roll-out	A product release is created, distributed and installed in accordance with the deployment model.
Support	Configuration and change management	
	Project management	
	Development environment	

There are other sources of good practices, such as the AUP, PMI's '**P**roject **M**anagement **B**ody of **K**nowledge' (PMBOK®; PMI, 2013). The freely-available '**ProjectInABox**' provides document templates suited for use within a variety of project management approaches (see: www.projectinabox.org.uk).

Other practices include generic approaches to standard activities like engaging with users to elicit requirements and how to conduct milestone reviews.

Reminder of learning outcomes

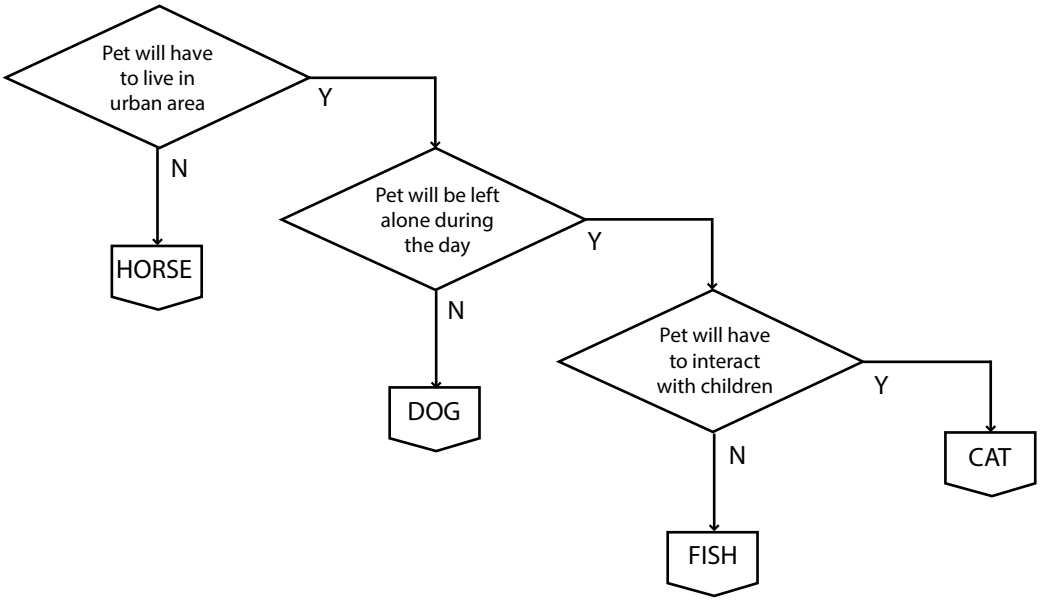
Having completed this chapter, and the Essential reading and activities, you should be able to:

- understand the concept of a process model as a way of describing a software development project and differentiate between common approaches such as waterfall, iterative and agile
- identify the key artefacts produced during an iteration of the software development process
- understand the role of the project manager in delivering those artefacts
- understand the ways in which the development process can affect quality.

Test your knowledge and understanding: defining appropriate models

We have discussed several key process models, but the Essential reading textbooks describe others. There is no 'one size fits all' process model and it is important for you to show that you can identify the strengths and weaknesses of the most-used models in a variety of contexts.

6. Consider the distinctive features of these key process models and the types of project that they are recommended for. For example, Pressman suggests that the waterfall approach is applicable when requirements are pre-defined and not going to change and the work is essentially linear. This may be the case with an embedded system or device driver.
7. Record your findings in a Decision Tree (a flowchart consisting of Yes/No branches) to select an appropriate process model based on the requirements of a project. For example, a decision tree for selecting a suitable pet might include the following:



8. You can also summarise your logic in a table with the key process models in the left hand column and the types of project on the right, building on the following:

Process model	Project type
Waterfall	Device driver (requirements are known)
	Embedded system (development is linear)

Chapter 2: Requirements engineering

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- describe the differences between elicitation, specification, validation and documentation of requirements
- understand the differences between system and user requirements and between functional and non-functional requirements
- use natural language and graphical representations of requirements within a requirements document
- produce a system model based on requirements.

Essential reading

Sommerville	Pressman
Chapter 4: Requirements engineering	Chapter 5: Understanding requirements

References cited

Grady, R. and D. Caswell *Software metrics: establishing a company-wide program.* (Upper Saddle River, NJ: Prentice Hall, 1987) [ISBN 9780138218447].

Overview

The process of identifying and recording requirements is explained in depth in both recommended textbooks. The importance of the process is that it yields an understanding of the scope of the project (the business needs to be met), the outputs from the project (artefacts) and the way of assessing the outcomes (benefits) that are achieved. It normally forms the contractual agreement between supplier and customer.

CORNERSTONE 6

A specification document must comply with the 'four Cs':

Clear: No ambiguities or confusing cross references to other documents.

Concise: No unnecessary detail, repetition or confusing cross-references to other documents.

Correct: Attributed, with no conjecture, paraphrasing or unsubstantiated opinion.

Complete: Verified as part of the V & V process.

The first step in scoping many projects is a feasibility study and this is very often a technical rather than a commercial decision. Examples include:

1. A technical study to see if a legacy database contains accurate enough data to allow an enhancement.
2. A pilot to evaluate the feasibility of reverse-engineering an existing process.
3. A trial to see how many extra sales agents the current network will support.
4. An evaluation of three vendors' COTS software.
5. A mock-up interface to see whether staff would feel comfortable using the proposed new Human Resource system.
6. A field trial to measure a round-trip time using a cloud architecture.

However, stakeholders should also be concerned with non-technical issues that affect a project's feasibility. According to Figure 1.6 in the previous chapter, the purpose of the inception phase of a UP project is to produce a set of artefacts that allow the two prime project stakeholder groups, the customer(s) and their supplier(s) to determine the feasibility of a new project (or new iteration of an existing project). The result will normally be expressed in a business case. It is vital that the business case is kept up to date throughout the life of the project, as a way of evaluating the ongoing viability of the project in response to changes in the status of risks.

Figure 1.6 also contains a much longer list of artefacts generated within the elaboration phase. One of the consequences of splitting the 'front-end ' of the project into inception and elaboration phases is that it avoids the need to create a detailed software development plan and user requirements statement before finding that the project cannot be cost-justified commercially. In an iterative, incremental or agile environment, some requirements may remain undiscovered until the construction phase.

CORNERSTONE 7

The project team invariably focuses on the outputs it is to create, while the client is concerned with the outcomes of using those outputs. 'Outcome-based specifications' address the client's needs and do not prescribe **how** they are to be met.

The output-based specification for a new sales-support system might include 'produce a weekly printed report from the database enquiries table summarising the status of sales enquiries received in the last week grouped by sales territory'. The rationale for this kind of statement might be that managers need information in that format in preparation for the weekly sales-team video-conference. A more likely reason, of course, is 'we've always done it this way'. The outcome-based equivalent would simply be: 'sales managers require up-to-date information on the status of enquiries in each sales territory'. This would allow the innovative supplier to suggest the use of an on-screen 'dashboard'. By putting the dashboard on the intranet, where it can be accessed by the sales team as a whole, time can be saved each week in the sales-team video-conferences.

The decision to proceed with a project is likely to be based on the answers to the following questions:

Questions the customer should be asking
PROJECT OUTCOMES:
Does the supplier know what we are trying to achieve?
PROJECT OUTPUTS:
Can we agree what the key deliverables and their acceptance process(es) will be?
BUSINESS JUSTIFICATION:
Do the potential benefits outweigh the costs and risk of taking the project forward?
Short/Medium/Long-term projections: WILL THIS STILL BE THE RIGHT DECISION IN ONE/THREE/10 YEARS' TIME?

Questions the supplier should be asking

OUTCOMES:

Has the customer explained why the project is important to them?

PROJECT DOMAIN:

Are we analysing the right problem? Are we listening to the right stakeholders?

APPROACH:

Can we agree an overall project architecture? Can we do the job?

POTENTIAL THREATS:

Do we understand the development process well enough to be able to identify and evaluate the key risks?

POTENTIAL BENEFITS:

Can we evaluate the potential business benefits (profit, avoiding disbanding the team, acquiring a new customer, positioning ourselves in a new market or developing new skills)?

Short/Medium/Long-term projections: WILL THIS STILL BE THE RIGHT DECISION IN ONE/THREE/10 YEARS' TIME?

Depending on the scale of the project, some or all of the following are likely to be used to help evaluate the answers:

Customer business information sources

- **Business vision** – Corporate goals.
- **Business rules** – Policies and constraints.
- **Business case** – Source of information about corporate values (intangible).

Project definition sources

- definition of scope
- user requirements, possibly supported by user profiles and key high-level use case definitions
- supplementary, non-functional, requirements
- interfaces: users, hardware, software, communications
- an inventory of risks with strategies for their avoidance or reduction
- project approach
- project architecture
- high-level project plan
- high-level resource plan
- project evolution.

Validation

Validating requirements locally with users – through mock-ups and scenario-based models – will reduce the risk of starting development on the basis of an incomplete or inaccurate specification of the static elements of the system. For example, one of the first things a web-designer is likely to ask a new client for is a list of examples of existing websites that embody the expected visual styling, tone of language, user functionality and overall business purpose.

Partially-functional prototypes provide another tool for eliciting feedback from users as the detailed design evolves. This has become increasingly valuable as developers gain access to re-usable library objects, especially user-interface objects, and acquire skills in using techniques from the field of usability.

Software Requirements Specifications

Descriptions of 'Software Requirements Specification' (SRS) documents can be found in Sommerville, Karl Viegers (see: www.processimpact.com/goodies.shtml) (the RUP-based origin of the outline referenced in Pressman) and, of course, the UP itself. These are very similar in structure, if not identical.

The content of each section should be clear from the following outline:

Introduction	Document purpose and project scope
Project overview	Business context for the proposed system
System features (from user requirements)	Functional requirements
External interfaces	Operational context
Supplementary requirements	Non-functional requirements

The lean thinking of agile methodologies such as Ambler's AUP emphasises reusing relevant source documents rather than rewriting or editing them. By the time the construction phase has started, the agile SRS could include UP artefacts containing documentation of business rules, domain model, organisation model, project glossary, analysis of automation opportunities, a business process model, technical requirements, use case model, a user interface model and definitions of acceptance tests. The exact mix will vary from project to project.

The SRS is not complete until the end of the high-level analysis phase, by which time the use cases or user requirements statements have been transformed into concrete system features.

CORNERSTONE 8

Functional requirements describe the way the system behaves.

Non-functional requirements describe the context within which that system behaviour is delivered.

User requirements are the functional and non-functional requirements that describe the business goals for the system.

System requirements are the functional and non-functional requirements that have to be met in order to achieve the business goals for the system.

Key concepts: gathering, analysing and formalising requirements

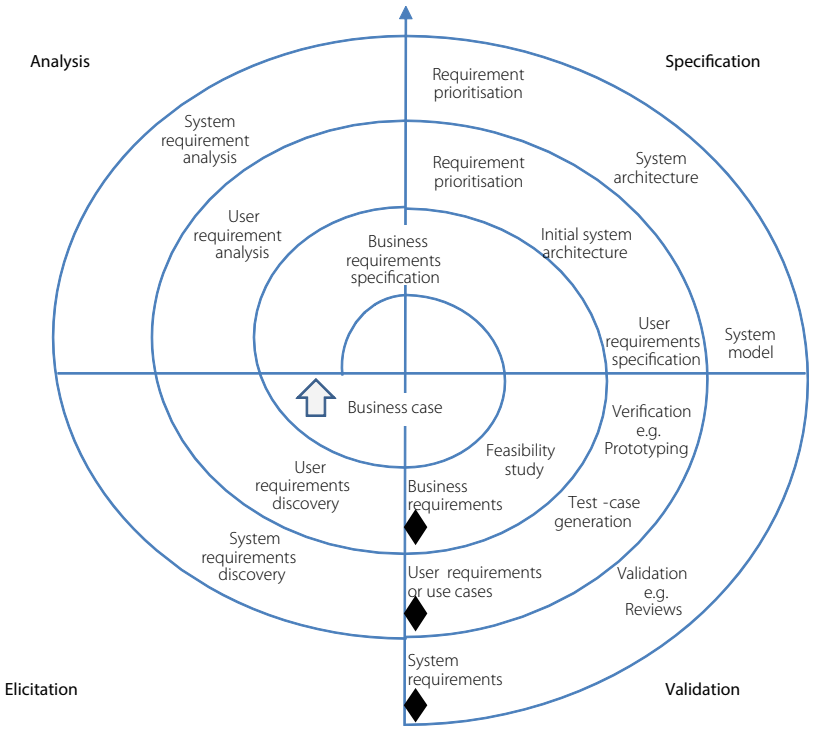


Figure 2.1: Requirements engineering, after Sommerville.

Figure 2.1 captures the logic of two diagrams from Sommerville, integrating details of elicitation and analysis activities within the process of defining an SRS and business, user and system requirements. The Spiral approach highlights the order in which these different specifications are normally produced:

- Business requirements define the scope of the project.
- User requirements define the system to be produced in sufficient detail to be able to plan the project.
- System requirements define the system in sufficient detail to allow it to be implemented.

The Sommerville and Pressman textbooks both describe the process of requirements engineering, including the various forms that a completed set of requirements can take – unstructured or structured narrative (natural language), high-level mathematical or logical expressions or graphical representations such as the UML. The process begins with the identification of the customer's requirements and there are various ways in which those requirements can be captured (elicited or discovered) and analysed.

The primary aims of analysis are to ensure the accuracy and completeness of the understanding of the customer's needs, which may influence the scope of the project. Prioritisation of the specific requirements allows functionality to be allocated to the first or subsequent iterations of the project.

The project management perspective

Modelling the business requirement

PMBOK® identifies seven reasons why a development project might be initiated. Since PMBOK® is not specifically concerned with software engineering, we have modified the list to reflect the types of project that you are likely to encounter:

- responding to market demand for a feature, product or service
- addressing internal needs or opportunities within the organisation
- servicing a customer request for a bespoke system
- technology-driven need to update an existing system
- constraints such as legal changes
- improving the way a system operates, such as reducing environmental impact
- infrastructure development (originally categorised as 'social need', such as providing drinking water).

Some projects will have clear monetary benefit (such as a successful upgrade to an existing system) while others may have less tangible outcomes such as improving customer satisfaction. In all cases, the decision to proceed is based on acceptance of the business case which gathers the evidence that delivering the project will result in the expected benefits. It will have to be revisited at various times during the project.

The starting point is normally 'the cost of doing nothing'. In other words, the factors in the decision are relative to that baseline, but are not absolute. This has implications when considering the acceptability of the system (discussed below), the trade-off between incompatible requirements, such as the time it will take to deliver something that 'works' and something that is robust.

It is the project manager's responsibility to provide accurate and complete evidence for the business case. As the project evolves, the business case will evolve to include cost and schedule estimates, which we will address in the next chapter of the subject guide. It will also include documentation of the requirements that have been identified and the response in terms of specific features, since these are the basis on which the costs are calculated. The project manager's main concerns are, therefore, whether the requirement is:

Understandable?	Complete within a well-defined context? (For example, if there are multiple user-types, the requirement may only apply to one user-type.) Stated in a clear, unambiguous way for the intended reader? Defined in a form that is consistent with the body of requirements? A description of a required outcome rather than a way of achieving that outcome? Attributable? Current (relevant to the current situation, not something learned from an out-of-date customer document)?
Implementable?	Necessary ('need to have' or 'nice to have')? Realistic? Measurable (non-functional requirements)? Testable? A prerequisite for another requirement? Dependent on the implementation of other requirement(s)?

This checklist is an example of a breakdown structure and every project manager should have such a framework to ensure that their analysis is complete.

FURPS – Functionality, Usability, Reliability, Performance, Supportability

'**F**unctionality, **U**sability, **R**eliability, **P**erformance, **S**upportability' (**FURPS**) (Grady and Caswell, 1987), developed at Hewlett Packard, is based in part on the thinking of Joseph M. Juran (see: www.juran.com). It defines the relationship between the functional and non-functional properties of a system in terms of its '-**ilities**'. 'An "**ility**" is a characteristic or quality of a system that applies across a set of functional or system requirements' (see: www.softwarearchitecturenotes.com/architectureRequirements.html). In other words, a given area of functionality must be implemented in accordance with a given set of non-functional requirements. Where there is a tension between two areas, it is the project manager's responsibility to establish what the customer's priorities are and what constitutes an acceptable balance between functionality, usability, reliability, performance and supportability. The following list of ilities may look like a simple checklist based on those categories, but it provides the customer with a mechanism for quantifying the importance of the various non-functional requirements, using a simple weighting function (a scale of 1 to 5).

FURPS+ (the '+' signifies that extra attributes have been added since 1987) is similar to ISO 9126 which divides **Supportability** into **Maintainability** and **Portability**. In either form, it is an important part of the discussion of measuring progress and also the related issue of the quality of that progress.

Functionality	Feature set, capabilities, generality, security.
Usability	Human factors, aesthetics, consistency, documentation.
Reliability	Frequency/severity of errors, recoverability, predictability, accuracy, mean time between failures.
Performance	Speed, efficiency, resource consumption, throughput, response time.
Supportability	Testability, extensibility, adaptability, maintainability, compatibility, serviceability, installability, localisability.

The relationship between requirements and system acceptability

A successful software engineering project is critically dependent on the accuracy and thoroughness of the requirements engineering process in order to:

- Determine the customer's functional and non-functional requirements.
- Balance those requirements to deliver the functionality, reliability, performance and supportability requested to an acceptable extent.
- Identify constraints that affect the project's duration, cost, choice of methodologies and standards.

Change

Evolution of a system is the subject of Chapter 9 of the subject guide. Here, we only need to say that the process of specifying and implementing a new version of a system should follow the process described above for the initial system. That is:

1. Changing business requirements drives the process through the Spiral model shown above – namely, high-level user requirements followed by revised system specification.
2. The decision to proceed is based on a business case.
3. The system is achievable in terms of balancing requirements.

Implications for project planning

Project planning, at a detailed level, is discussed in Chapter 6 of the subject guide. However, the requirements capture process described in the previous section is critical to success in planning what the development team will do as well as for providing evidence as to whether the project should proceed or not. The decisions that need to be made are:

1. Who will do the work?
2. What are the implications of the chosen project architecture?
3. How can the major risks be avoided or mitigated?
4. How can the work be broken down to make the project more controllable?
5. What resources need to be acquired?

These decisions are typically taken in parallel with and throughout the process of translating user requirements into system requirements.

Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- describe the differences between elicitation, specification, validation and documentation of requirements
- understand the differences between system and user requirements and between functional and non-functional requirements
- use natural language and graphical representations of requirements within a requirements document
- produce a system model based on requirements.

Test your knowledge and understanding: gathering evidence

Sommerville proposes a template for an SRS and it is important that you understand the likely sources of the evidence that you use to develop those requirements and how the project documentation evolves.

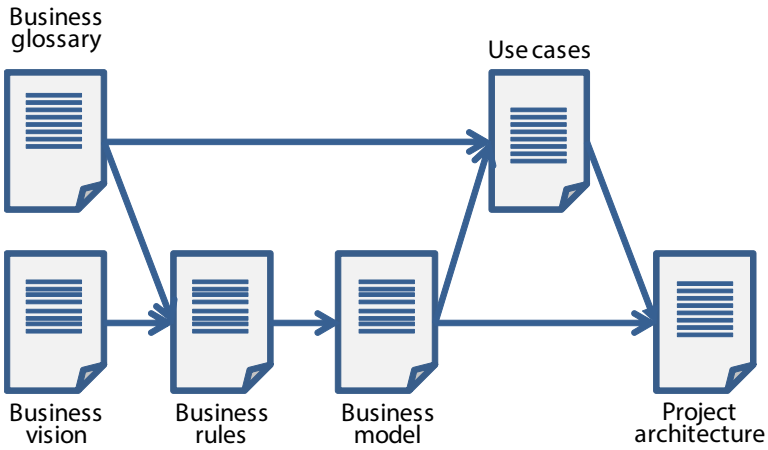


Figure 2.2: Extract from a Product Flow Diagram.

1. Review the content of this chapter, and in particular the outline SRS.
2. Now, construct a **'P**roduct **F**low **D**iagram**' (PFD)**, following the example in Figure 2.2 to explain the artefacts needed to produce an SRS.
3. Finally, divide the artefacts into those produced before the end of the inception phase and those produced during the elaboration phase.

Chapter 3: Planning, cost and schedule estimation

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- describe the structure of a project schedule
- apply resources to execute a project plan
- understand the constraints on the accuracy of resourcing and budgeting.

Essential reading

Sommerville	Pressman
Chapter 23: Project planning	Chapter 26: Estimation for software projects Chapter 27: Project scheduling

Further reading

Fenton, N. and S.L. Phleeger *Software metrics: a rigorous and practical approach*. (Boston: PWS Publishing, 1998) second edition [ISBN 9780534954253].

References cited

Hsu, C.C. and B.A. Sandford 'The Delphi technique: making sense of consensus', *Practical Assessment, Research & Evaluation* 12(10) 2007, pp.1–7; http://pareonline.net/pdf/v12n10.pdf

Overview

Successful execution of a software project is dependent on three planning activities:

1. Budget/resource-planning.
2. Scheduling.
3. Monitoring and progress management.

In 1995, the Standish Group estimated that software projects were costing US companies and government agencies $250bn/yr (see: www.spinroot.com/spin/Doc/course/Standish_Survey.htm). A third of this money ($81bn) would be on projects that were cancelled before they saw the light of day. A third of the balance ($59bn) would go on finishing software projects that would be completed, but exceed their original time estimates.

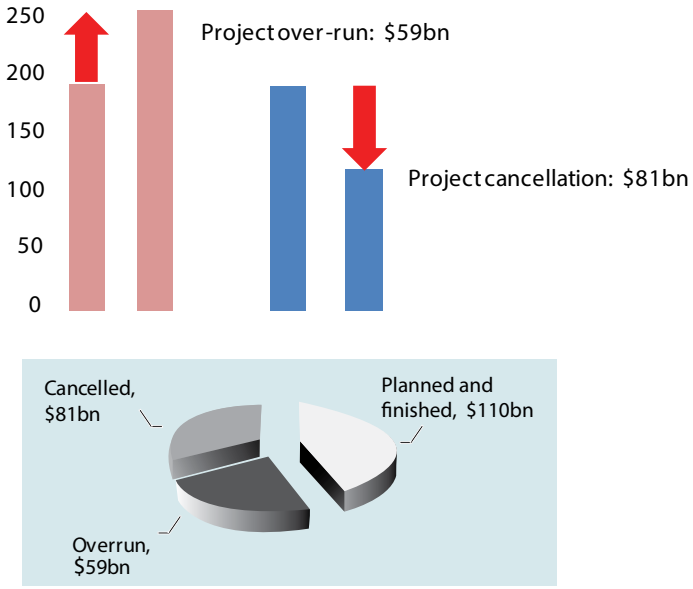


Figure 3.1: Project failures, after Standish.

The research has subsequently been repeated on several occasions and, while the numbers may vary year-on-year, there is no evidence of any real improvement in the number of projects that are completed on-time and on-budget, with all features and functions as initially specified or a reduction in the number of 'impaired' projects that are terminated prematurely.

The remainder, of course, are the projects that are completed and operational but over budget, over the time estimate, and offer fewer features and functions than originally specified.

Plan-driven development and agile development

The essence of a plan-driven approach to a project is that the customer and developer commit to a detailed project plan (specifying and scheduling release of a series of products) before entering the construction phase. The factor that remains (relatively) constant is the functionality and performance of the product. Deviation from plan results in a product going out of tolerance (time, cost or quality).

An agile development team does not expect to be constrained by a detailed plan (although high-level planning may be required by the principle of '**serial in the large, iterative in the small**') since the 'constant' is not the product but the timing of the iteration cycle. Functionality is dynamically prioritised and allocated to a planned release.

The two approaches are simply different ways of achieving the same goal: fulfilment of the customer's strategic criteria identified by Price Waterhouse Cooper (PWC). Not only time and budget but stakeholder satisfaction, benefit realisation, quality and return on investment. Sommerville makes the point that many projects will incorporate elements of both approaches and lists a series of questions that affect the way the project is managed. The implications of these technical, organisational and human issues can be represented as follows:

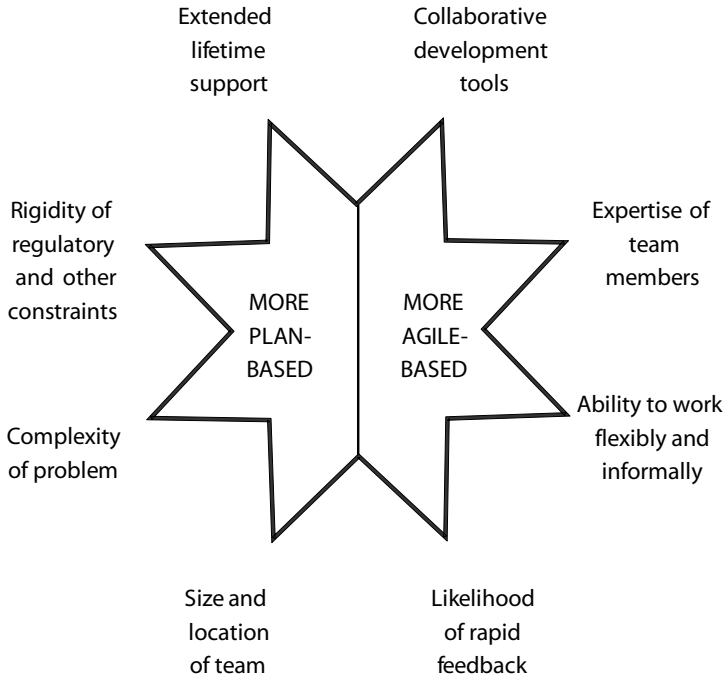


Figure 3.2: Constraints on development strategy.

Key concepts: time, money and quality

Scheduling: time

Scheduling is essentially the translation of the process model into a network of activities. Four factors determine how the network is constructed:

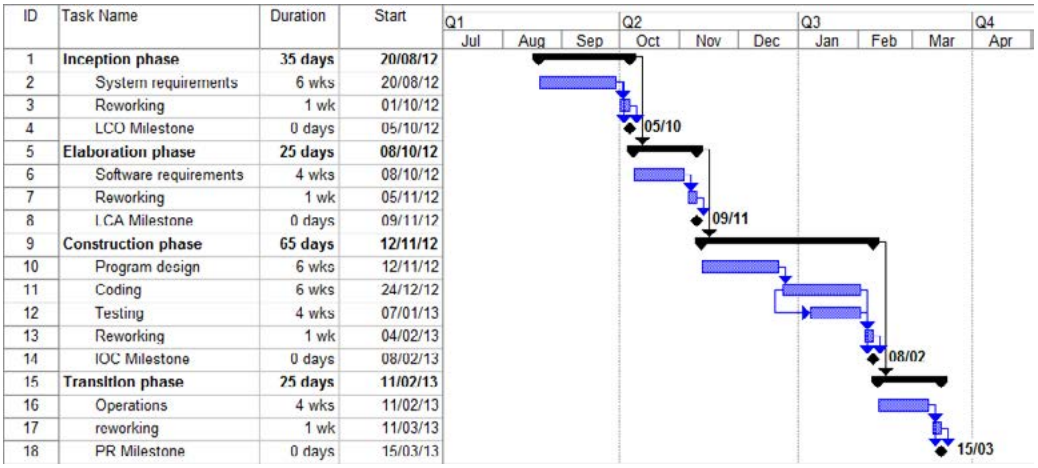
1. The way the project decomposes into **tasks** (work breakdown structure).
2. The availability of the **resources** needed to execute the tasks.
3. The estimated **time** to completion for each task.
4. The **interdependencies** that dictate prerequisites for starting a task.

Three additional factors affect the overall duration of the project.

5. **Reworking** caused by design/software errors.
6. **Reworking** caused by changes in customer requirements.
7. **Delays** caused by risks that materialise as issues.

The most common expression of the network is the bar chart or GANTT chart. This consists of rows reflecting the hierarchical organisation of tasks in the '**W**ork **B**reakdown **S**tructure' (**WBS**) with time as the horizontal axis. Each bar (representing the expected start and finish dates) has an implied or explicit relationship with others – its predecessors and successors. Overlap of parts of two bars means they are concurrent. Most project planning software (such as Microsoft Project) maps 'work to task' as well as 'task to time'. Resources can be allocated to tasks and as the project unfolds, the tool can track the rate of progress and consumption of resource.

The project plan for the process depicted in Figure 1.5 in Chapter 1 of the subject guide could be depicted as follows:



ID	Task Name	Duration	Start
1	**Inception phase**	**35 days**	**20/08/12**
2	System requirements	6 wks	20/08/12
3	Reworking	1 wk	01/10/12
4	LCO Milestone	0 days	05/10/12
5	**Elaboration phase**	**25 days**	**08/10/12**
6	Software requirements	4 wks	08/10/12
7	Reworking	1 wk	05/11/12
8	LCA Milestone	0 days	09/11/12
9	**Construction phase**	**65 days**	**12/11/12**
10	Program design	6 wks	12/11/12
11	Coding	6 wks	24/12/12
12	Testing	4 wks	07/01/13
13	Reworking	1 wk	04/02/13
14	IOC Milestone	0 days	08/02/13
15	**Transition phase**	**25 days**	**11/02/13**
16	Operations	4 wks	11/02/13
17	reworking	1 wk	11/03/13
18	PR Milestone	0 days	15/03/13

Figure 3.3: 'Waterfall' GANTT chart. Also available on the VLE.

This GANTT chart assigns a bar (and therefore resource) to reworking that arises from unsuccessful milestone reviews. This means that a known level of resources have been scheduled to reworking. The bar appears to the left (that is, before, in time) of the milestone, not after it as might be assumed. The reason is that the milestone has not been passed if reworking is taking place.

The WBS does not in itself carry any timing or inter-dependency information. One good way of informing the design of the GANTT chart is to design a Product Flow Diagram (PFD) linking the outputs of each task. It is actually easier to develop an accurate WBS from the PFD than it is to do the design the other way round.

Estimating: cost

There are five components of a costing calculation for software development:

1. **Scale** of problem
2. **Efficiency** of staff
3. **Unit cost** of staff
4. Project-specific **equipment costs** (for example, servers, licenses)
5. **Overheads** associated with staff time (travel, accommodation, utility equipment such as phones, laptops).

Estimating the work required to deliver the end result is the one difficult part of this equation, especially in the early days of the inception phase. Whether the unit of work is **'L**ines **O**f **C**ode**' (LOC)** or methods of a class, efficiency is an expression of the number of those units that a member of staff can produce, error-free, in a working day. This is relatively straightforward in the case of a stable, effective development team working on a familiar task and likely to be much more accurate than the estimate of the number of units of work required.

There are three basic methods for estimating the scale of the problem:

1. Guesses (estimates made by experienced practitioners)

   One way of improving the accuracy of expert estimates is to elicit smallest (s), largest (l) and most likely (m) values, where the expected value (E) is given by E = (s+l+4m)/6 assuming a beta distribution.

2. Revisions of previous guesses (based on the actual effort required to deliver similar projects)

   Where a pool of expertise is available, a 'Delphic consensus' (also known as the 'Delphi technique') can be used. Even an average is more robust than a single opinion.

3. Statistical manipulation of guesses, where the guesses become parameters of a calculation

   There are three significant approaches to statistical estimation of effort described in the recommended textbooks. These are:

   1. Constructive Cost Model (COCOMO II)
   2. Function Point Analysis (FPA)
   3. Object Oriented (OO) approach.

   Sommerville identifies several reasons why the result may not be used as the final price:

   Exchanging profit for opportunity such as:

   - entry into new market or working with new client
   - retention of rights to use code
   - gaining experience, skills or specialist staff.

   Exchanging profit later for cash flow now in order to pay the bills (mostly salaries). Remember that this is itself a risky option, profit or loss being '…**the very small difference between two very large numbers**'.

Performance: quality

Management of the delivery of quality is addressed in Chapter 8. However, the project plan itself must reflect the approach to quality. It is generally assumed that the goal of a project sponsor and, therefore, his or her designated project manager is to deliver 'on time' and 'on budget', but that is only part of the story. A Price Waterhouse Cooper survey (2007) (see: www.pwcprojects.co/Documentos/BRO Boosting Business Performance DC-07-1104-A v8[1].pdf) found that only 19 per cent of managers prioritised on-time delivery and a further 18 per cent budget as their primary criterion for success.

Nearly two-thirds, therefore, emphasised more strategic criteria such as satisfaction of their stakeholders (20 per cent), delivery of benefits (17 per cent), quality of the result (15 per cent) and return on investment (9 per cent). Part of the role of the project manager is to establish methods of monitoring progress towards such strategic objectives and this will affect the selection of the most appropriate SE methodologies for a particular project. The solution (and the way of managing its implementation) depends on the nature of the problem.

CORNERSTONE 9

ISO/IEC 12207 defines validation as confirmation that the final, as-built, system fulfils its specific intended use, that it is 'fit for purpose'. Verification confirms that the output of each activity (such as requirements capture, design, code, integration or documentation) fulfils the requirements or conditions imposed by the preceding activities.

This is often expressed in terms of validating that the 'right thing' is being built and verifying that the thing is being 'built right'.

The project management perspective

There are several project management methodologies that can be applied to software projects including '**PR**ojects **IN C**ontrolled **E**nvironments' (PRINCE2) (™ HM Government), the Project Management Institute's PMBOK® mentioned earlier and ITIL ('Information Technology Infrastructure Library') (™ HM Government).

We discuss the principles that underpin these methodologies throughout this subject guide. For example, PRINCE2 takes a product-centric view of a project, contains mechanisms for controlling the transition from phase to phase and divides work into work packages, each typically responsible for delivery of one major project artefact.

We will not examine your knowledge of the terminology or processes of a specific methodology, but students are encouraged to investigate one or more of the major ones. For example, there is a comprehensive web-based PRINCE2 reference resource available (see: www.crazycolour.co.uk/prince2/?title=Main_Page). A freestanding tool has been produced called 'Project In a Box' that includes a free downloadable 'community edition' (see: www.projectinabox.org.uk). This also contains support for ITIL and other methodologies.

Product-based and activity-based planning

Product-based planning is one of our cornerstones. It allows resources to be scheduled according to the role each resource plays in constructing the product and it allows progress to be monitored in terms of delivery of products rather than completion of tasks that contribute to those products. For example, three teams working in parallel are planned to produce components (user interface, the database transactions and the implementation of business rules) that will result in achievement of a major sub-system.

The modified PFD (Figure 3.4) shows the status of the sub-system or component. Design is complete, work has started on the business rules module and the other two modules are finished but not tested. Knowing that the user interface design team has completed their task tells you nothing about the status of the sub-system. You can only make a meaningful statement about the status of the higher-level product when all three components are ready for integration, which is when it reaches its first milestone (prerequisites are in place).

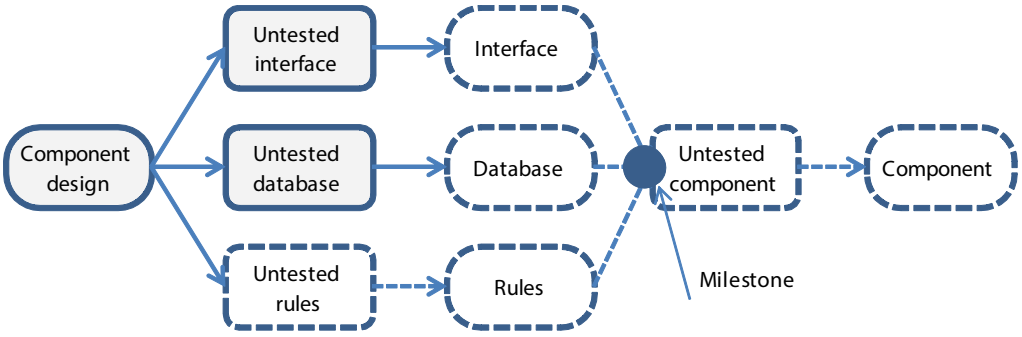


Figure 3.4: PFD showing product status.

Timing

The tracking GANTT (Figure 3.5) shows the activity-based view of the project on 26 September when the database work finishes and is ready for testing, three days late. The user-interface task is on schedule and 60 per cent through the unit testing sub-task.

But the third activity, the business rules work, is only 80 per cent complete. It has a scheduled nine days of work left before the tested component can be released. That would imply that the first milestone on 28 September, already revised to 3 October due to delay in the database task, will probably slip to 8 October. The finished artefact would be more than a week late.

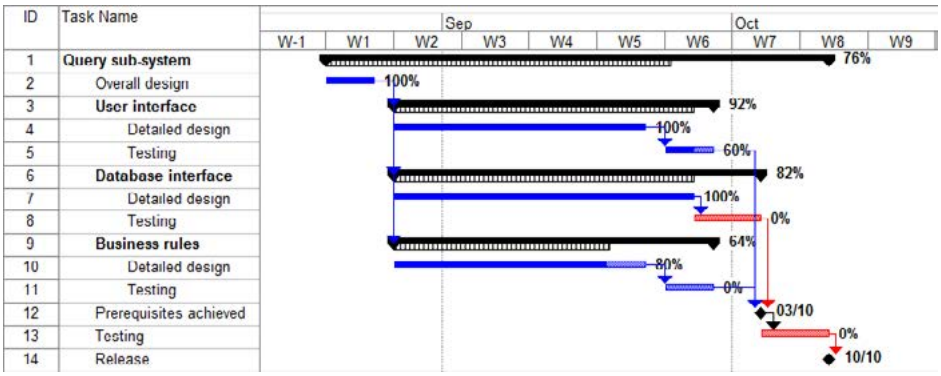


Figure 3.5: Tracking GANTT. Also available on the VLE.

Monitoring

Monitoring the plan requires an objective assessment of the extent to which a product has been achieved (often, internal milestones or 'checkpoints' are used for this) rather than a statement of the effort that has been used en route to that checkpoint. Consider the following:

'The GUI prototype has been evaluated by end-users and our business rules have been accepted by the client, so we are ready to begin the final iteration of the construction on time.'
This means that relevant checkpoints have been identified and the purpose of the milestone is understood.
'We were late producing the specification but have caught up by leaving some of the lower priority things till the next release.'
This means that the specification (for the current release) has not been approved. Therefore the milestone to start development has not been achieved. An exception plan (see Chapter 6 of the subject guide) is needed.
'We're three months into a twelve month project with a fixed team, so we must be 25 per cent finished by now.'
This is frequently an indication that there are no milestones, no resource plan and therefore no way of managing the transitions between activities.

Delegated responsibility and exception-based reporting

The example above does not consider how an 'early-warning' system can be put in place. The PRINCE2 approach is to delegate responsibility for monitoring progress on a product to the team actually doing the work. During planning, agreement is reached on the resources to be made available, the time available for doing the work and the quality of the end results – within specified tolerances. From that point on, the project manager only needs to know two things:

The **status** of the work (hopefully complete) when the milestone date is reached.

Early warning that the product is going out of tolerance for time, cost or quality.

The database task is currently three days, or 15 per cent, late. This is probably within tolerance. The business rules team, however, should have been aware by the end of, say, the third week of their design task, that they were at risk of over-running (and probably running out of resources). This should have raised an exception condition for the sub-system.

That is the limit of delegated responsibility. The project manager is then able to take action to rectify or normalise the situation, which would depend on the circumstances of the rest of the project but could be one of the following:

- Revise the delivery date for the sub-system.
- Try to honour the original schedule by assigning extra resource to the business rules team.
- Redefine the specification for the sub-system to reduce the work needed.

Once again, the project manager is in place to consider and strike a balance between time, cost and quality.

Accuracy of projections

Boehm (1995) observed a large variation (a factor of 4) in the initial estimated cost of a wide range of projects. One of the goals of a project manager is to reduce that variation, if not immediately then at least before the end of the elaboration phase of the project. A series of quality gates can be implemented, at which the uncertainty should be reduced significantly. For example, the completion of the initial requirements capture could be the first gate and aim to halve the uncertainty. This is illustrated in Figure 3.6 below, and forms a major part of risk management.

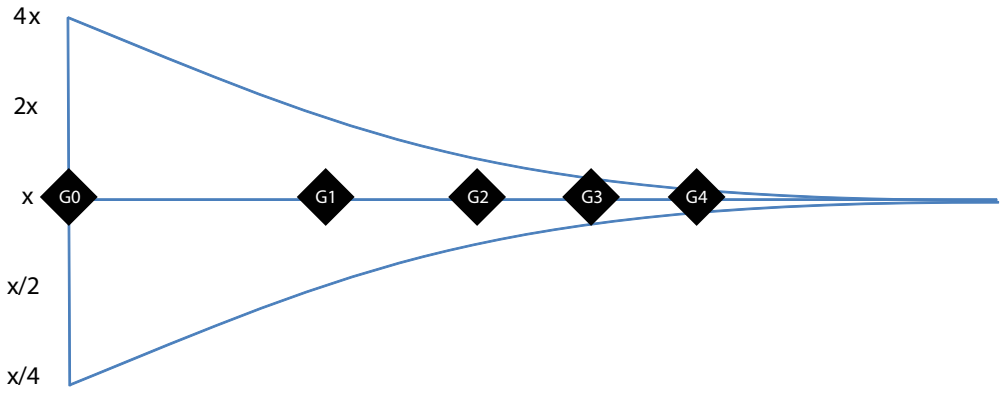


Figure 3.6: Controlling estimation error.

Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- describe the structure of a project schedule
- apply resources to execute a project plan
- understand the constraints on the accuracy of resourcing and budgeting.

Test your knowledge and understanding: The PRINCE2 approach

Several formal project management methodologies have been introduced in this chapter. These are mostly intended for use in large projects but we have focused on PRINCE2 because it is scalable to large and small projects and because of the availability of resources such as CC Training and ProjectInABox. It is important that you can demonstrate an understanding of this scalability.

PRINCE2 is not covered in either Pressman or Sommerville, but free web-based resources are available. We have identified several in this chapter (under the heading 'Project management perspective') and introduce exception-based reporting (see section 'Delegated responsibility and exception-based reporting'). A more detailed explanation of the methodology is presented in Chapter 6 (Figure 6.2).

There are several other key concepts such as dividing a project into stages with detailed Stage Plans and delegated responsibility through the use of Work Packages (discrete well-defined products).

Read about the principles of the methodology from a source such as CC Training (www.crazycolour.com/prince2/?title=Main_Page) and consider how it corresponds to the process models you have been studying. As you explore this, draw up two lists from the material you read – one of concepts that are familiar and one of concepts that are less familiar. This second list is likely to consist of some of the formal methods used in larger projects, such as managing stage boundaries and closing a project.

Summary of the inception phase

- SE processes include addressing capture and specification of the system needs (requirements engineering), translation of those needs into an executable software system (design and implementation) and checking that the result addresses the specification and meets the real needs of the users (validation).
- The way these processes interact can be represented in the form of process models and the RUP is a generic modelling tool.
- Requirements for systems evolve and the SE processes must be able to cope with change.
- Agile methods identify and respond to requirements incrementally, using short development cycles. XP uses test driven development and involves the customer as part of the team. Scrum provides a priority-driven project management framework.
- Agile is a valid alternative to a conventional plan-driven approach as long as it is understood by the development team, acceptable to the organisation and scaleable to address the communication needs of large development teams.
- A software requirements document is an agreed and normally contractual statement of functional requirements (what the system should do) and non-functional requirements (constraints to be imposed on the system).
- The requirements engineering process includes an iterative and validated process of discovery, analysis and documentation of changing customer requirements.

By now, you should be confident that you would be able to:

1. Define the scope of a project in terms of core requirements, key features and the main constraints and risks.
2. Produce a plan describing the development process that you would follow, supported by estimates of cost and schedule in line with the client's business case.

Part 2: Elaboration phase

Chapter 4: Risk management

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- describe the steps required in order to understand how a risk should be mitigated, monitored and managed
- give examples of risks created by the use of technical and human resources, the way the client and developer organisations work and performance-related risks
- identify probable causes of risks in those five areas and ways in which pro-active risk management can be used to reduce the probability and/or impact.

Essential reading

Sommerville	Pressman
Chapter 22.1: Project management (Risk)	Chapter 28: Risk management

Further reading

Charette, R.N. *Software engineering risk analysis and management.* (New York: McGraw Hill Intertext, 1989) [ISBN 9780070106611].

References cited

Lamri, M. 'Risk reduction with the RUP phase plan', IBM Developer Works 2003; www.ibm.com/developerworks/rational/library/1826.html

Overview

There are risks associated with all stages in the project. Sommerville and Pressman both describe the issues that a project manager may encounter in terms of the external factors of a SWOT analysis; opportunities as well as threats. We include opportunities because not all issues are negative. You need to be aware of potential positive outcomes, such as budget under-spend as well as the more common budget over-run.The RUP is respectful of the importance of risk, but its elaboration phase simply asks that the risks in the subsequent phases have been identified. The message is that if an activity can lead to exposure to an area of risk that is not controlled, then the activity should be deferred to a later iteration when the risk is better understood (Lamri, 2003). Boehm recognised this when he dedicated one quarter-turn of the Spiral within each revolution (phase of development) to risk identification and reduction.

Key concepts: Risk Mitigation, Monitoring and Management (RMMM)

The stages in management of risk are described in the recommended textbooks, and collectively known as '**R**isk **M**itigation, **M**onitoring and **M**anagement' (**RMMM**) .

Pressman describes the approach to risk identification advocated by the US Air Force (Air Force Systems Command pamphlet AFSCP 800-45), which is to look for the causes of risk by constructing a breakdown according to the aspect of the project that would be

affected – its performance, cost, maintainability or schedule. Another approach would be to consider risks in terms of the FURPS categories.

Identifying risks can be more complex than expected, since there are many different types of risk. One simple approach is to consider the risks that would affect the project, others that would affect the product that is being built and a third set that would affect the customer's business. Some of the most common risks in these three areas are shown in Figure 4.1. Note the overlaps.

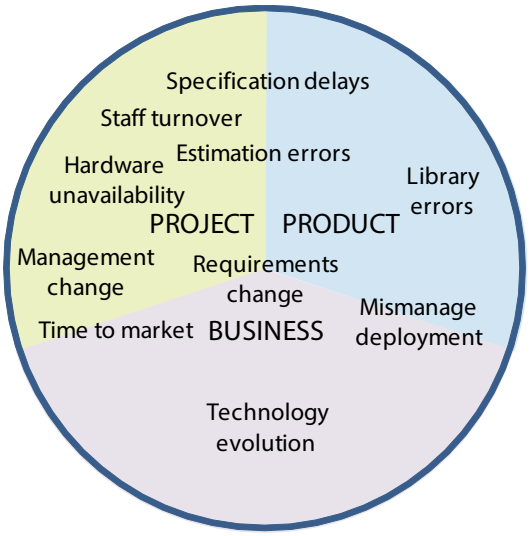


Figure 4.1: Risk categories and types.

The concept of **Control** is important. See Figure 4.2. It is not always possible to avoid a risky activity but once the risk is understood, a proper business decision can be taken as to whether the potential reward outweighs the potential penalty. If that is not a convincing proposition, the other alternative is to find the best way of minimising (known as mitigating) the consequences of the risk. Mitigation requires that the risk is identified and analysed in detail, in terms of its probability, timescale and potential impact. The risk management plan can then be drawn up and executed, with periodic reassessment and identification of new risks.

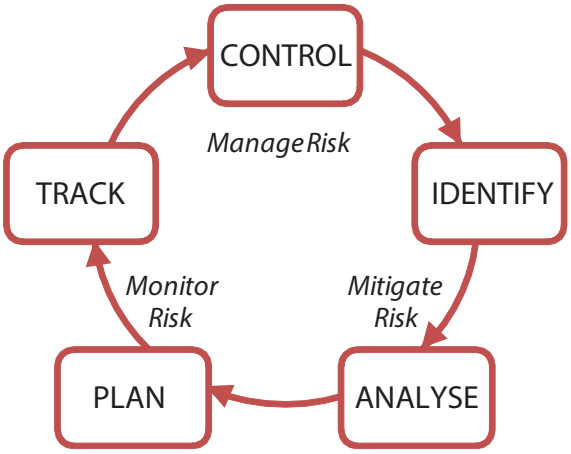


Figure 4.2: Risk management, a continuous process.

For example, Table 4.1 below illustrates some of the risks associated with requirements capture. The table does not include an assessment of the risk, since that will be project-specific.

Table 4.1: Common requirement risks.

Risk	Form of mitigation
Misunderstood requirements	i. Use of formal methods rather than unconstrained natural language ii. Validation task to ensure: (a) that the documented requirements mean the same thing to all stakeholder groups; and (b) that the meaning is consistent with the original elicited requirement
Changing requirements	i. agreement on target of frequent small stable increments in the system ii. avoidance of unnecessary (premature) analysis
Incomplete or inconsistent requirements	Validation methodology (for example, based on the UML) using breadth-first (project-wide) and depth-first (detailed) representations
Obsolete requirements	Traceability
Unnecessary requirements	Agreement on scope of project

Risk mitigation

The normal method of evaluating a risk is to apply two criteria – likelihood of the risk becoming an issue and the impact if it does so. This allows the most appropriate response to be devised. This is described in the textbooks and can be summarised in matrix form. Methodologies such as the UP and Spiral introduce a third criterion, proximity in time (namely, which phase of the project will be affected) since understanding of the risk will evolve over time.

HIGH probability **LOW** impact REDUCE likelihood	**HIGH** probability **HIGH** impact AVOID
LOW probability **LOW** impact ACCEPT (and monitor)	**LOW** probability **HIGH** impact REDUCE impact by contingency plan

As shown in the matrix, the most efficient strategy is to begin by reducing the significant criterion. If staff turnover is a high probability risk with moderate impact, then avoiding staff turnover by motivating and incentivising people is a more appropriate response than taking steps to reduce the impact in advance by adding backup resources to the team.

If something is unlikely to happen, it is probably not worth taking expensive precautions to make it less likely. Either simply accept the risk at the planning stage, or make a contingency plan, a 'second best' approach that can be invoked in the unlikely event of it being needed.

The box in the top-right is the situation to be avoided – an important risk that is very likely to happen. An example would be a specialist role within the project, such as knowledge engineering. Better to subcontract to a specialist than get an amateur to do the job badly.

Some project managers use a three-by-three matrix allowing assessment of **High**, **Medium** and **Low**. You can prioritise the risks to be mitigated by applying some function to the values given to each criterion of the assessment (such as multiplying values on a scale of 1–5, so a medium-likelihood and medium-impact risk scores 3*3=9, while a medium-high probability, high-impact scores 4*5=20). Pressman's approach to sorting a risk table can be used in this way. As Sommerville says, however, the scales are somewhat arbitrary at this stage in the analysis and the approach takes no account of the business case for mitigating individual risks.

We advocate evaluating each risk, however trivial, and assessing the outcome of the mitigation strategy as part of project planning. That way, the steps required to reduce the likelihood are incorporated in the project plan (such as avoiding risky design options or building contingency into the schedule). Contingency planning can also be scheduled, even if the plan itself does not have to be executed.

CORNERSTONE 10

The outcome of a risk analysis should reflect the status of the project after the mitigation has been brought into the project plan, in other words the risk register reflects the residual risk only. This implies that risk assessment is an iterated or continuous process in order to re-evaluate mitigated risks.

Risk monitoring

At the beginning of the RMMM process, the information available about risks is somewhat arbitrary and has to be made more accurate as the project evolves (and the proximity of the risk gets closer). One technique is to identify indicators of change in risk status – a high rate of test failures is one likely indicator that something is about to go seriously wrong. More formally, the PMBOK® identifies six elements to the Monitor and Control Risks process:

1. Regular reassessment can help identify new risks as well as changing the values in the existing register.
2. Risk audits are important when responsibility for risk is delegated (and when it is not).
3. Indicators in the form of variance and trend analysis can show the need for reassessment of, for example, the parameters used in COCOMO.
4. Technical performance measurement at key milestones gives an indication of the likelihood of meeting quality criteria.
5. Analysis of reserve budget, the planned contingency (augmented by the occasional under-budget task) will indicate the capacity to absorb an issue.
6. Discussion of risk status at project meetings supplements the sharing of information in the shared risk register and allows potential issues to be assessed from various perspectives.

Risk management

The PMBOK® makes the valid point that '…risk management becomes easier the more often it is practiced' and suggests that raising the profile of risk monitoring as a project activity makes it more likely that people will identify risks (and opportunities).

The project management perspective

The project manager is responsible for taking precautions against specific risks. It is not adequate to devise a management plan at the level of generic risk types described in the textbooks and something that may be a major risk in one project, such as moving to a new IDE, may not be relevant to another project after the IDE has been proven. The

project manager's response depends very much on the cost to the project of mitigating (reducing or preventing) the risk compared to the cost of the risk coming to fruition as a project issue. It is like making a business case – quantify the difference between taking an action and doing nothing.

Known, predictable and unpredictable risks

Predictable risks such as loss of key personnel, changes to requirements, underestimating testing time, problems in the project's external environment may have happened before but they are not the only risks to be identified and should not be treated in a generic way. Do not simply state that 'loss of key personnel is a risk' – identify: who are the key personnel for this project, what skills do they bring; and how committed are they?

Known risks are not the same as predictable risks. They are very project specific and often emerge from project documentation. Examples include requirements that translate into activities that the team has little experience with or deadlines and feature sets that the project manager is coerced into accepting by the client or his or her line manager. These are decisions that are most frequently described later as having been made 'against my better judgement'.

Finally, there are unpredictable risks. You cannot predict the unpredictable, but you can anticipate it. Make provision for the unexpected in the budget and the schedule and encourage all project members to look out for signs of a new risk coming over the horizon. Management of known, predictable and unpredictable risks is the focus of Robert Charette's work in the military, public and private sectors.

Risk quantification

Our understanding of risks evolves as the project does. For example, when planning the maintenance phase of a project, you will have to consider the volume of change requests that can be expected, the way the system has been implemented and the effect that has on the ease of making the changes and the ability of the project team to make the changes.

Early indicators that the maintenance phase will not run to plan can be found in slipping deadlines and high levels of errors to be fixed. The exposure to risk will be progressively refined as the project evolves and corrective actions are taken. There are, broadly, three ways of evaluating a risk, suitable at different stages in the project lifecycle.

Qualitative assessment

This approach is typically used at an early stage since it is: (a) relatively simple; (b) relatively quick; and (c) relatively cheap. The standard risk analysis matrix is used and the categorisation is an approximation based on the knowledge and insight of experienced practitioners.

Quantitative assessment

Statistical techniques such as 'Monte Carlo' and 'regression analysis' are among the techniques for projecting the likelihood and impact of a risk. Project management techniques (such as '**P**rogram **E**valuation and **R**eview **T**echnique ' (PERT) and COCOMO) can be used to provide some of the input data. The concept of known and predictable risks can help, because the severity of a risk that manifested itself in a previous project should be quantifiable.

Compared to a qualitative assessment, these techniques are relatively complex, time-consuming and therefore more costly. However, provided the methodology and raw data are sound, the result, applied to the top dozen or so risks (prioritised using qualitative methods) will be more accurate.

Quantifying cost

Even if the customer does not fully understand the project timeline, or the implications of delay, a discussion based on the cost of the delay is unlikely to be ignored. 'The release will be six weeks late because…' is somehow less compelling as an argument than 'Delivering on time will cost an extra $10k because…'.

The US Air Force model (See Air Force Systems Command pamphlet AFSCP 800-45.) includes cost as one of its four categories of risk, but the real analysis is often how to trade-off risks that have performance, schedule or maintenance implications against those that have cost implications – especially when managing a fixed-price contract.

The business case approach helps with risk quantification: if we allow this risk to affect the project, the implication is exposure to a potential liability of $X. The cost of halving the likelihood of the risk occurring is $Y. If Y is less than half of X, then the mitigation is cost-effective. The challenge is how to gain the agreement of the client and senior management that the liability really is $X, not $X/4 or 4*$X.

Risk management techniques

A risk is a theoretical factor (opportunity or threat) that may occur and may have an impact on the project. A project issue arises when a risk does occur and will have an effect unless something is done to prevent it doing so. Risks become issues unless steps can be taken to address them. Different types of risk require different approaches.

A '**R**isk **B**reakdown **S**tructure' (RBS) can provide a checklist of areas where risks can be found. An RBS can also capture the types of qualitative risk described by Pressman – such as those due to the size of the project, experience of the team, quality of development tools and stakeholder characteristics. It is good practice is to share the analysis of risks between client and supplier, between the various groups within the project team (for example, analyst, developer, tester) and with any subcontractors or service providers. This can lead to consideration of risks from different perspectives. The project risk register is where all information about project risks can be stored. It rarely starts as an empty list as some types of risk are predictable. They have happened before and the history can be analysed. PRINCE2 has a specific activity at the end of the project, which is to build an analysis of **lessons learned**. This can feed directly into the first iteration of risk planning for the new project.

Earlier we used the framework of **Project+Product+Business** to illustrate the range of risks but this may not be the best starting point for risk analysis in practice since many of the items in the diagram belong to two or more categories.

Sommerville suggests an RBS that starts with six categories: **technology**, **people**, **organisational**, **development environment**, **requirements** and **estimation**. Steps to quantify and manage these different categories of risk could include the following:

Managing technology risks

Typical issues with technology (excluding the development environment, see below) are in areas such as performance, availability and support. Trials can be undertaken to assess the scalability and efficiency of a platform or system; existing users can sometimes provide information about reliability, workarounds and optimisation.

Availability (in terms of delivery dates) may be affected by the need for customisation by the supplier and the schedule of new releases; maintenance costs can be estimated by evaluating similar products that are already in the market and may be in use within the organisation. It is vital to compare 'like with like'.

Example risk	Potential issue
Software may run too slowly	Client rejects software release
Mitigate by including performance tests in development cycle	Execute contingency plan to upgrade hardware

Managing people risks

Various drivers affect the attitude of staff to their employer and these are often inter-related. For example, a well-respected member of staff with status and visibility may place less importance on financial reward. Multi-skilled people (often categorised as 'key personnel') may find complex and challenging projects more attractive than repetitive turnkey systems (such as small company non-transactional websites) whereas someone motivated by stability and fear of failure would find the challenge too threatening.

The best indicator of people risks is past history. Staff performance and staff turnover are likely to remain relatively consistent as long as the drivers do not change. The one exception is the unpredictable risk of a new member of the team becoming a destabilising force. Other project managers in the organisation and the Human Resources/Personnel Department would be the most likely sources of insight into people issues.

Example risk	Potential issue
Key developer may leave the project	Key developer leaves before training of a replacement is complete
Mitigate by training several team members in key skills.	Execute contingency plan to transfer a senior developer from a lower-priority project.

Managing organisational risks

The organisation executing the project can affect the outcomes in many different ways. Practical examples of organisational risk (all from personal experience) give some indication of the scope for an issue arising.

- An in-house development team may be affected by a corporate decision to acquire more specialist skills in-house and reduce the use of subcontractors.
- A poor recent trading history may result in pressure to 'buy the work' by aiming to undercut the market.
- A change in ownership, or a decision to relocate, or a decision to enter a new market may have significant impact on the morale of the development team.
- Appointment of a new account manager with a different client-handling approach may affect the organisation's ability to renegotiate a live project.

Note that, like people risks, these four examples all emerge from change. Anticipating changes in the way the organisation operates is not simple. If the current operation is running smoothly, there is likely to be less appetite for change.

Conversely, when change becomes necessary, each line manager should have a clear idea of the likely impact.

Example risk	Potential issue
Project may lose high-priority status	Management change strategic direction and re-allocate resources to other projects
Mitigate by ensuring senior management are aware of strategic importance (for example, important new client, expansion of capacity).	Execute contingency plan to reschedule the project.

Managing environmental risks

Environmental, in this context, means the development environment, the responsibility of the RUP Environment discipline. This includes things like IDE and CASE tools where used, configuration management, language compilers, testing tools and so on.

Given that development practices tend to be evolutionary, rather than subject to global change, the likelihood is that issues in this category arise from badly implemented changes rather than change itself.

Two factors are central: the expertise and experience of using the selected tools and the consequences of an issue. Sometimes, mitigation of a risk is as simple as checking the compatibility between two tools. For example, a Microsoft Office tool which embeds references to external resources as absolute paths is a potential issue in terms of compatibility with the intranet. Another example would be lack of agreed standards and working practices leading to two developers using different compiler switches.

Example risk	Potential issue
Working files may be corrupted or lost	Disgruntled member of staff sabotages the working files
Mitigate by implementing version control, daily backups and off-site storage.	Execute contingency plan to stop all development work until the problem is fixed.

Managing requirement risks (and, more broadly, managing expectations)

Poorly stated, poorly understood, poorly validated and subject to revision, requirement risks are among the most obvious sources of issues for a project. That is not to say that changes to requirements create the most severe threats to a project, but that many projects fail to implement the steps necessary to mitigate the risk.

Example risk	Potential issue
Client may not understand the impact of changes to requirements	Client requests major change
Mitigate by involving client in the process of deriving specifications from requirements.	Execute contingency plan to negotiate an interim product release before starting work on new requirements.

Managing estimation risks (and, more broadly, managing the project plan)

Scheduling and resourcing are the subjects of other chapters. The point that needs to be made in the context of risk management is that managing risk is simply one facet of managing the project as a whole. Compare these two diagrams:

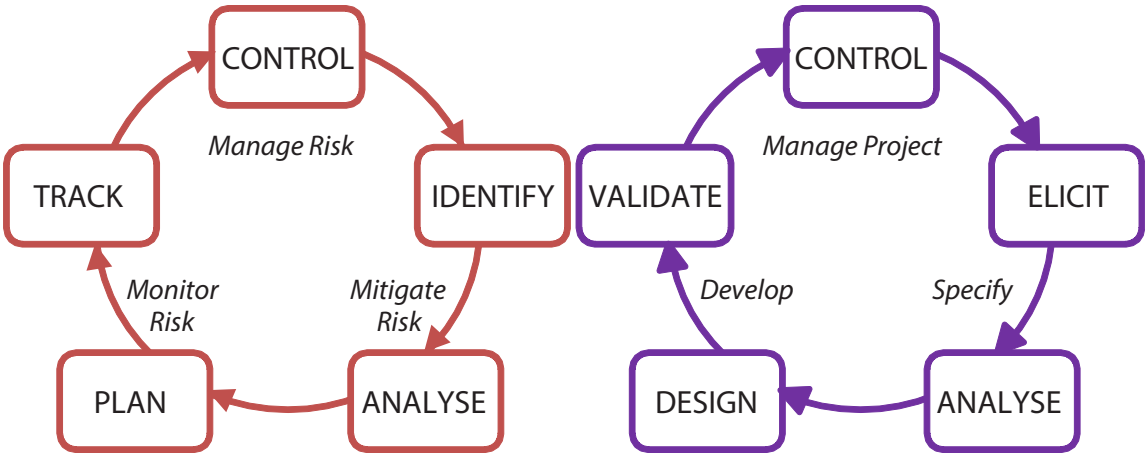


Figure 4.3: Control of risk (left), control of project (right).

Example risk	Potential issue
Lack of experience with client's preferred development tools may result in significant over or under-estimate of project resources	Project over/under performs
Mitigate by setting up a small-scale pilot and make training of developers a budget item.	At project-end, ensure that it is seen as part of an overall portfolio with a mix of successful and less successful projects.

Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- describe the steps required in order to understand how a risk should be mitigated, monitored and managed
- give examples of risks created by the use of technical and human resources, the way the client and developer organisations work and performance-related risks
- identify probable causes of risks in those five areas and ways in which pro-active risk management can be used to reduce the probability and/or impact.

Test your knowledge and understanding: risk breakdown structure

Risk management provides the mechanism for anticipating and dealing with potential issues. It often feels like dealing with the unknown but it is important that you are able to apply a comprehensive framework for thinking about risk.

There is no right or wrong way of making a Risk Breakdown Structure; the important thing is that it helps you identify the unexpected risks as well as the standard ones about resourcing, debugging and specification creep.

1. Use the information from this chapter to construct an example of an RBS, starting from the six overall headings used by Sommerville: **technology**, **people**, **organisational**, **development environment**, **requirements** and **estimation**. Focus on the risk indicators described by the PMBOK® process quoted above. Add specific risks to your example, categorised as known, predictable and unpredictable.

Notes

Chapter 5: Architecture, modelling and design

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- use UML graphical models to represent the context (external environment) in which the system will operate, the interactions between the system and that environment, the static and dynamic structures of the system and its responses to stimuli
- make informed decisions during the architectural design process
- base a system architecture design on a generic architectural pattern where appropriate.

Essential reading

Sommerville	Pressman
Chapter 5: System modelling Chapter 6: Architectural design	Chapters 6–7: Requirement modelling: scenarios and behaviours Chapter 8: Design concepts Chapter 9: Architectural design

Further reading

Ambler, S. *The unified process elaboration phase: best practices in implementing the UP*. (New York: CRC Press, 2000) [ISBN 9781929629053].

Kruchten, P. 'Architectural blueprints – the "4+1" view model of software architecture', *IEEE Software* 12(6) 1995, pp.42–50.

Rosenberg, D. and M. Stevens *Use case driven object modeling with UML*. (London: Academic Press, 2013) [ISBN 9781430243052] second edition.
See: www.iconixsw.com/Books.html

Stevens, W., G. Myers and L. Constantine 'Structured design', *IBM Systems Journal* 13 (2) 1974, pp.115–39.

References cited

Kazman, R. et al. *The architecture trade-off analysis method.* (Software Engineering Institute, 1998), CMU/SEI-98-TR-008.

Overview

A model can be used as a tool to stimulate discussion about, or to document, some aspect of an existing or proposed system. A set of models can be developed in sufficient detail to provide the basis of a system description – an architectural model – from which a detailed design can be developed.

The architectural design process is central to all process models and leads, for example, to the main output of the UP elaboration phase. It can include several iterations, starting with a high-level 'sketch' of what is known about the proposed system and its environment and evolving into a set of decisions about the physical distribution and hosting of functionality.

Use cases and the '4+1' model

The scope of a new software system can be defined in terms of **who** is going to use it, **what** they will use it for and **how** they will use it. Put into the terminology of software engineering, a use case model defines the actors ('who'); the use cases or functionality ('what'); and the external context or environment ('how').

Use cases are the basis for the architectural design of a system under the UP. A high-level use case suited to discussion of architecture may simply be '…a contact centre agent (the "who") checks the account balance for a customer (the "what") on the customer database over the intranet (the "how").'

Obviously, decomposing that single sentence into its functional elements will reveal several lower-level use cases which trigger further discussion about the architecture.

Does the system have to:

- **Authenticate the caller** (requiring secure access to customer data)?
- **Allocate a caller to an available agent** (requiring functionality to monitor agent activity)?
- **Do anything if no agent is available** *(*requiring additional functionality)?
- **Log the transaction** (requiring a logging database)?
- **Authenticate the agent before allowing database access** (may require a staff card or other form of ID)?
- **Interact with any other systems**?

If an iterative process model is being followed, these more specific use cases can delineate high-priority functionality required for the next release or iteration.

Peter Kruchten (1995) describes the result of architectural design as the implementation of '4+1' views of the proposed system, views being typically UML diagrams that represent parts of the relevant UP models as shown in Figure 5.1.

The '4+1' views tie together two views of the Design model (Logical and Process views) and views of the Implementation and Deployment models. The '+1' view is the emerging architectural design that supports the required use cases.

The purpose of the use cases is to describe functional requirements and behaviours. The reason for focusing on use cases and developing models using the UML, as described in this chapter, is that the UML provides a structure that allows you and your designers to concentrate on one thing at a time in a systematic way. Hopefully, that will be the right thing at the right time.

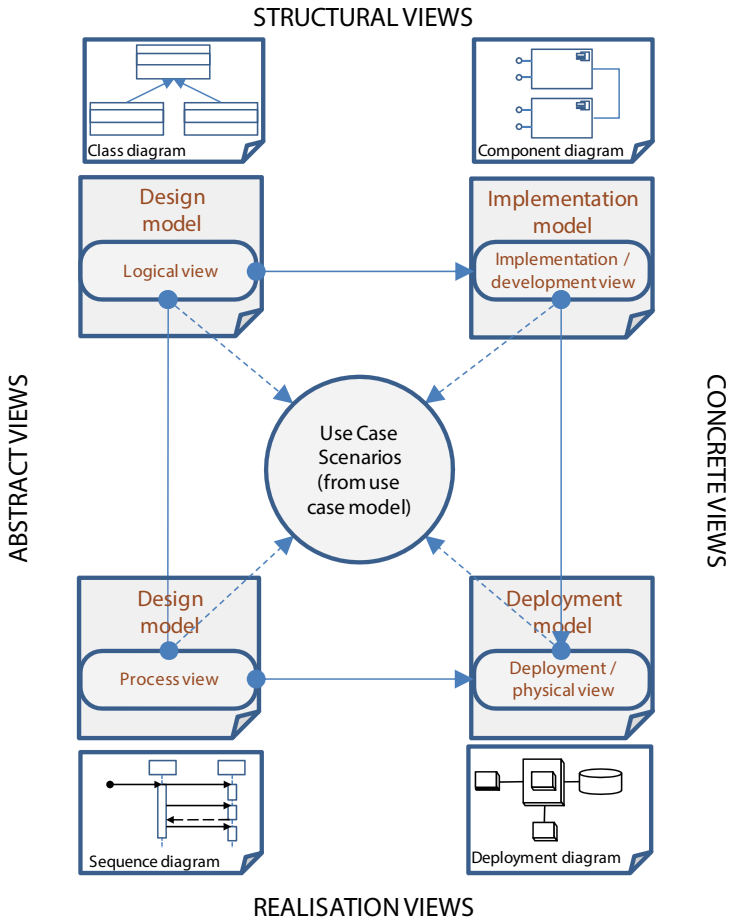


Figure 5.1: Kruchten's '4+1' views.

It is important that the use case model expresses the full behaviour of the system, that the set of use cases is complete (describes all required behaviours) and that they do not introduce spurious (unnecessary or unwanted) behaviour. It should be possible to map all functional requirements to at least one use case and to identify which specific use case demands each non-functional requirement. This allows system testing to be based on the use case model and the scenarios it describes.

Key concepts: classes, flows and behaviours

Classes

Architectural design processes, such as the '4+1' model, initially focus on the very high-level relationships between business objects such as actors, databases and existing services.

A class of objects is a set of objects that share certain attributes. When we talk about 'actors' in connection with requirements and architectural design, we are talking about a class made up of all people who may interact with the system. If some people have specific roles and responsibilities, they can be treated as belonging to a sub-class. An object-oriented approach to describing (namely, modelling) the business environment of a proposed system begins by defining the high-level interactions between actors and the system and between the system and other systems. This is frequently called a context model.

A journal publishing management system, for example, may have five classes of user: «publisher», «editor», «reviewer», «author» and «subscriber». An author may be a contributing author or «contributor» (able to take decisions on behalf of the others).

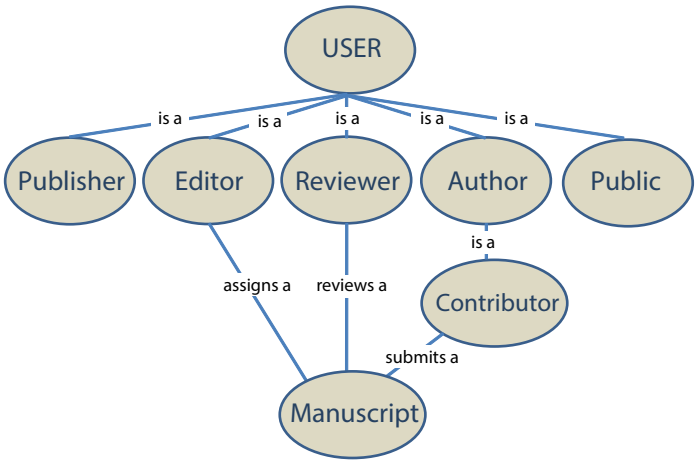


Figure 5.2: Academic review case.

Figure 5.2 shows the development of the context model for a specific use case, which is how the editor arranges for a set of reviewers to read a manuscript prior to acceptance. This is a static view of the logical components of the system. The next step is to consider how these static objects are related in terms of the flows between them.

Flows

Flow can be considered in two ways. One way is to concentrate on the interactions between objects (flow of control) and the other is to consider the relationships between system processes in terms of flow of data. Data flow diagrams are discussed later in this chapter. The approach reflected in the '4+1' model is to use UML sequence diagrams to describe the temporal logic of the interactions. This adds a process-oriented view to the evolving design model but flow is not the same thing as behaviour.

Behaviours

The objects in a system can, as already mentioned, have various states and they will be able to perform different roles according to their state. While a sequence diagram describes the order in which messages are created and processed and a communication diagram describes the form of those messages, a state diagram describes their effect. Depending on the project, one or more of these three models may be required.

Making the architectural model concrete

As well as expanding the design model to describe object flows and behaviours, the '4+1' process also describes how the abstract or conceptual design classes can be given 'physical' or concrete form as components in an implementation model. Finally, where these components are to be distributed between multiple physical nodes in the delivery platform, the architecture includes a deployment model.

CORNERSTONE 11

When deciding whether to use the UML during architectural design, you need to be able to answer yes to the following five questions.

1. Do we have good tools for doing things this way?
2. Do we understand how to use them?
3. Do we know what we are trying to achieve?
4. Will we document the decisions taken clearly?
5. Will the client understand the result?

In terms of the logic of the '4+1' model, the starting point is the top left with an abstract structural model of system context. This model needs to be made 'realisable' by adding flow and behavioural information. It also needs to be made more concrete by considering the interfaces needed to connect the components together. You start with a business model that contains a static view of the elements of the problem that are important, defined as classes. Sequence diagrams add temporal detail (which can be augmented by communication diagrams and state change diagrams) and the component diagram describes the interfaces they will support.

For obvious reasons, it is not possible to step from an abstract high-level model to a deployment model without considering the process and implementation views. Less obviously, it is not possible to get from the bottom left of the diagram to the top right in one step. That would compromise the separation of architecture and what is thought of as system design. Sommerville expresses the concern that high-level modelling naturally leads to the temptation to start design activities before the specification has stabilised. He also points out, with regard to agile approaches, that stable overall system architecture is a prerequisite for the first design iteration: 'Incremental development of architectures is not usually successful.'

The project management perspective

The purpose and limitations of requirements and architectural models

Not every practitioner you meet will have the same view of the importance of developing (and documenting) an architectural model in response to requirements captured at the beginning of the project. Ambler advocates the agile approach based on the maxim of 'travel light' and creating 'just enough' documentation. But he also says:

> 'Use cases are a good technique to document behavioural requirements but that's only a small part of the functional requirements picture and an even smaller part of the total requirements picture – they aren't very good at documenting business rules, user interface requirements, constraints, or non-functional requirements – which is why the UP includes something called a supplementary specification to contain all these other things. Requirements drive things, use cases don't.'
>
> (Ambler, 2000)

Boehm's Spiral model was intended to encourage a milestone-based way of thinking about the early phases of project definition, an approach that is also found in the UP. At the end of the day, it does not matter methodologically whether you use a '**D**ata **F**low **D**iagram' (**DFD**) or a class diagram as the starting point for discussion. The important thing is that the discussion is informed by the model. Similarly, you are going to get a comparable result whether you describe functional requirements in bullet points or in a use case model.

Representation of data flows

The Data Flow Diagram has been part of the software engineer's toolkit since the mid-1970s and work on structured design at IBM (Stevens et al., 1974) and elsewhere. A DFD is a diagrammatic representation of

[INPUT]->[PROCESS]->[OUTPUT]

In fact, a DFD is not a single diagram, but a hierarchy that starts at the contextual level (the 'system' and all the relevant data flows in and out of it), and continues adding more layers (each expanding the flows of one of the sub-systems or processes of its parent) until the sub-diagram can fit on a page as a form of activity diagram.

A DFD is a model of the way data in a system are stored, how they are processed and what happens to them. If your application is not 'data-centric' (for example, a graphics package processing messages from its user or a car's electronic control system processing

messages from sensors) then a DFD only shows part of the overall architecture, even with DFD Real Time extensions.

Architectural genres

The development of an architectural design need not be a process that always starts with a clean sheet of paper, since there will normally be some precedent that can be copied or learned from. The recommended textbooks identify several key 'genres' of software application including:

The **Model**, **View**, **Controller architecture** which provides a mechanism for implementing more than one user interface (view) to allow users to interact with an application and its data (model) by providing a mapping (controller) between user actions and state changes in the model. This is particularly relevant to web-based applications where interaction between the user's browser and web-server has to be channelled through the HTTP protocol.

Client-server architectures which resolve the need to separate local and remote functionality but typically address the need to provide one user interface to control multiple server-side processes rather than allowing multiple user-interfaces for one set of data as would happen using the MVC model.

Layered architectures which build up a suite of services based on lower-level functionality that is common across a range of services or applications. Whereas MVC allows multiple interfaces and client-server allows multiple mappings between interface and application, a layered architecture allows multiple application processes to co-exist.

Pipe and filter which builds up the required functionality from discrete processes in a pipeline to provide a sequence of transformations, such as import data, process the data, export the result.

Architectural design decisions and assessment techniques

Architectural design is a process of trading off the benefits and disadvantages of various options that can be applied to the logical, behavioural, structural and physical aspects of the design problem. Many of these options come from standard architectural patterns for different genres of application and will offer different solutions to meet non-functional requirements as well as functional ones. One of the first considerations, therefore, is how well an existing logical or physical architectural pattern delivers the required performance, security, safety, availability and maintainability.

Rather than starting the assessment by choosing and evaluating one pattern, the textbooks recommend broadening the scope of the assessment into a kind of tournament, eliminating one or more options after each round. The justification is that often a modification of a less obvious approach will out-perform the more natural contender. The '**A**rchitectural **T**rade-off **A**nalysis **M**ethod' (**ATAM**) (™ Carnegie Mellon University) is described in the Software Engineering Institute (SEI) publication with the same name (see Kazman et al., 1998).

Reminder of learning outcomes

Having completed the Essential reading and activities, you should be able to:

- use UML graphical models to represent the context (external environment) in which the system will operate, the interactions between the system and that environment, the static and dynamic structures of the system and its responses to stimuli
- make informed decisions during the architectural design process
- base a system architecture design on a generic architectural pattern where appropriate.

Test your knowledge and understanding: familiarity with the UML

The 'Four + 1' views provide a useful framework for modelling and documenting a project based on the UML. It is important that you can show the ability to work with the UML tools, to be able to read your team's documentation even if you do not have to produce diagrams yourself and to make sure that you use the optimal tool for each task.

Below (in Figure 5.3) is an informal class diagram to describe the manuscript review use case.

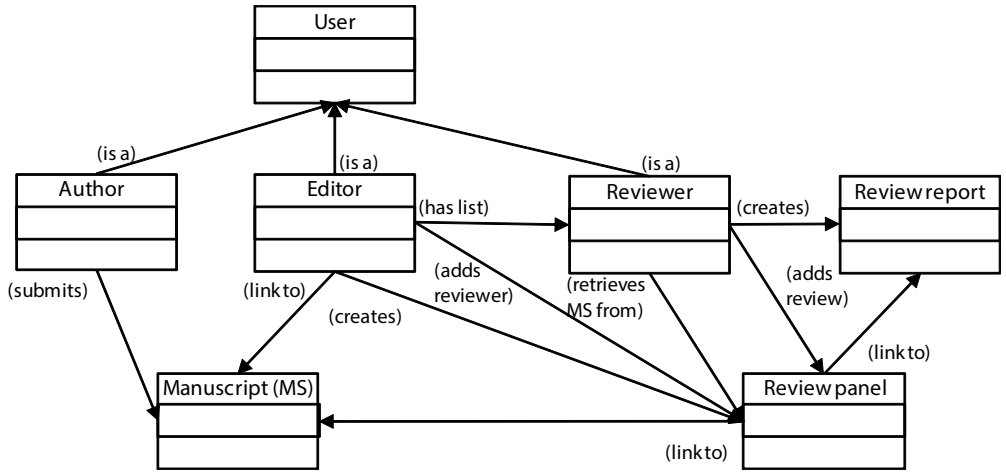


Figure 5.3: Class relationships.

Now try building on this diagram using your understanding of the use case and applying several different UML tools, such as:

1. A **state diagram** to show the lifecycle of the review panel class.
2. A **sequence diagram** to show what interaction takes place between the author, editor and reviewer classes.
3. A **communication diagram** to define the nature of those interactions.
4. An **activity diagram** to show the logic of the approach taken.

Notes

Chapter 6: Managing the execution of the project plan

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- describe the main functions of a project manager
- understand the dynamics of a development team
- identify the tools and infrastructure required for the development environment
- describe the roles and responsibilities for configuration management.

Essential reading

Sommerville	Pressman
Chapter 22.2: Managing people Chapter 22.3: Teamwork Chapter 24.4: Software measurement and metrics Chapter 25: Configuration management	Chapter 24: Project management concepts Chapter 25: Process and project metrics Chapter 22: Software configuration management

Further reading

Sommerville	Pressman
Chapter 3: Agile Software Development	Chapter 3: Agile Development

References cited

Cockburn, A. and J. Highsmith 'Agile Software Development: the people factor', *IEEE Computer* 34(11) 2001, pp.120–22.

Jacques, D. and G. Salmon *Learning in groups: a handbook for face-to-face and online environments.* (London: Routledge, 2006) fourth edition [ISBN 9780415365260].

Schwaber, K. 'Agile processes and self-organisation', Pearson Informit Articles, Software Development & Management > Agile, 9 November 2001; www.informit.com/articles/article.aspx?p=24015

Overview

This chapter follows on from Chapter 3 of the subject guide, which was about the creation of the project plan. Here we are concerned with the execution of that plan, which PMBOK® calls '**Direct and Manage Project Execution'**. The project manager is responsible for scheduling, executing and monitoring the tasks or activities that accomplish project objectives and lead to assurance and handover of deliverables. This includes:

- resourcing (materials, tools, equipment, facilities and staff) including procurement
- training, managing, representing (and sometimes defending) the team
- gathering data on project cost, schedule, technical and quality progress
- managing risks and responses to risks and issues
- issuing change requests and adapting the project plan, scope and environment where necessary.

The project manager is also responsible for:

- establishing and managing channels for communicating project status to team members, senior management and other stakeholders
- process improvement, including analysis of lessons learned.

Some commercial software development companies will provide two interfaces between the company and its clients, a project manager who represent the interests of the company and an account manager who is seen to be acting on the customer's behalf. The value of this is that the project manager does not meet the client on a day-to-day basis. It works like a front-office/back-office arrangement ('I am sorry, I asked the project manager but she said she couldn't help.').

CORNERSTONE 12

Formal documents are critical. They tend to be kept (if not actually read) and they tend to change over time. Do not simply flag the changes by giving a document a different date in its filename or by changing the name from V0.2 to V0.3. Give formal documents a history section and keep them (without ever changing the filename) in a document archive such as Sharepoint. The LEAN philosophy acts as a good guide: if you are not going to keep it, don't bother updating it. Conversely, if you do not maintain it, there is little point in keeping it!

Team size

Doug Putnam (see: www.qsm.com/process_improvement_01.html), from the company QSM, reports research that in terms of productivity, small teams are best, with productivity reduction in teams of around nine people or more. Jacques and Salmon (2006) found in a different context (learning) that in groups of more than six people, differentiation of roles occurs and face-to-face communication reduces. Above 12 people, sub-groups emerge. It appears that the Roman army was organised hierarchically down to the equivalent of 'team leaders', each responsible for a tent of eight men. This corresponds to the range that both Putnam and Jacques and Salmon were concerned about, 7–12 people.

Sometimes a project team sub-divides quite naturally, with 'horizontal' specialist teams servicing more than one project, such as a 'soft skills' group (quality assurance, process improvement, documentation, staff development and user training) and a technical support group (operation of the development environment, configuration management and the use of SE tools). These would each have its own team leader. At other times, the team can be split into two or more because of the architecture of the project (for example, one implementing business rules, one building a database, the third implementing the GUI). If there is just one single team, the roles of team leader and project manager will often be merged.

The Airley Council critical practices

The US Navy's '**S**oftware **P**rogram **M**anagers **N**etwork' (**SPMN**) has published a list of 16 'Critical Software Practices for Performance-Based Management' extracted over a five year period from some 200 project risk assessments by members of the Airley Software Council (see: http://cross5talk2.squarespace.com/storage/issue-archives/2001/200103/200103-Evans.pdf). There are three dimensions, concerned with the integrity of the project, the way the system is constructed and the stability of the end result. As can be imagined, the focus is on methodologies for large projects, but the principles are scalable.

Table 6.1: Airley Council critical practices.

Dimensions	Approach	Actions
Project integrity	Plan and implement the project within known constraints, requirements and expectations.	Adopt continuous risk management. Estimate cost and schedule empirically. Use metrics to manage. Track earned value. Track defects against quality targets. Treat people as the most important resource.
System construction	Use proven methods of assuring compliance with verified requirements throughout the project lifecycle.	Adopt lifecycle configuration management. Manage and trace requirements. Use system-based software design. Ensure data and database interoperability. Define and control interfaces. Design twice, code once. Assess re-use risks and costs.
Stability	Test for and correct all defects that prevent compliance with agreed requirements and expectations.	Inspect requirements and design. Manage testing as a continuous process. Compile and smoke test frequently.

Key concepts: governance and communication flows

Andy Murray has published a White Paper on governance issues and practices (see: www.best-management-practice.com/gempdf/PRINCE2_and_Governance_White_Paper_Nov2011.pdf) in relation to PRINCE2-2009, for which he was the lead author. Paraphrasing slightly, he views project governance as being about maintaining the communication channels between all the various project stakeholder groups to ensure that the best possible decisions about the future of the project are being taken.

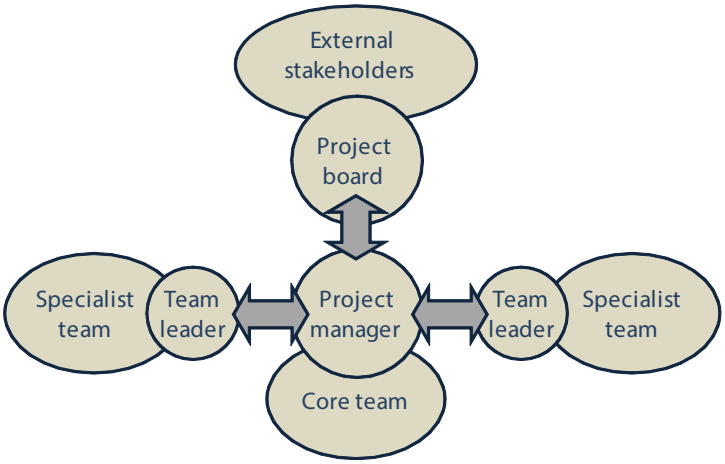


Figure 6.1: Governance in a project.

The project manager is the hub for governance within the project – the essential process of '**Directing a Project'**, in PRINCE2 terminology. As well as managing the flow of information between the various bodies shown in Figure 6.1, the project manager gets authorisation to make the transition from phase to phase of the project and other, less formal, advice and support from the project board. Where the project is part of a broader procurement there may be a hierarchy of project managers and project boards.

Delegation, schedules and plans

The PRINCE2 approach is shown graphically in Figure 6.2, although many artefacts are not shown or described. These will typically be found within the '**P**roject **I**nitiation **D**ocumentation' (**PID**). Starting up a project requires a sponsor's mandate, justified by a business case. These are the only two documents that are normally produced by a higher authority than the project manager, who is not appointed until the mandate has been produced.

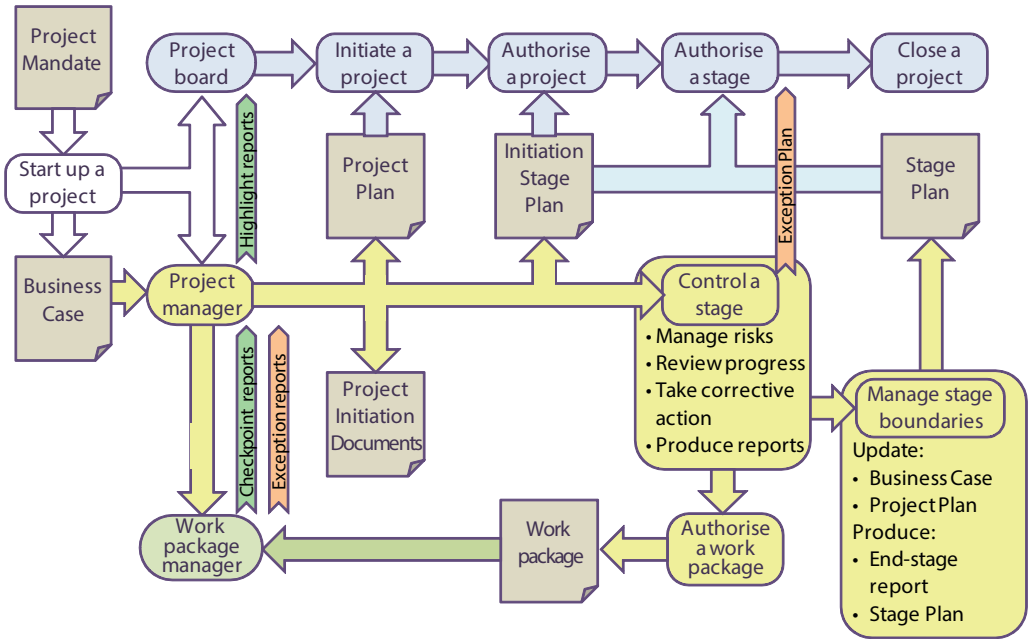


Figure 6.2: Delegated responsibility in PRINCE2.

A project board is appointed and is responsible for authorising each of the key lifecycle transitions in the project: initiating the project (akin to the inception phase, but not iterated), authorising it to start and then authorising each stage (phase) until the project

is closed down. These decisions require input from the project manager including risk, quality, communication and configuration management strategies and mechanisms for controlling the project. The business case is updated, as it will be prior to every stage transition.

After producing the stage plans (which essentially have the same function as UP phase plans; see: Masoud Javadi; http://ce.sharif.edu/courses/89-90/2/ce484-1/resources/root/iteration planning.pdf), the project manager's main responsibilities are in controlling the current stage of the project and preparing for the next. Just as the project board delegate these responsibilities to the project manager, so she or he delegates responsibility for each project deliverable (specified in a 'work package') to the work package manager or team leader.

Periodically, work package leaders prepare checkpoint reports which the project manager summarises into highlight reports for the project board and other stakeholders.

Checkpoint reports

A checkpoint report covers three reporting periods – actions outstanding from the previous period, progress made in the current period and results planned in the next. In addition to these statements of progress, a checkpoint report confirms whether the work package is still 'within tolerance' (see Chapter 3 of the subject guide) and provides any necessary update on the status of risks and issues. A checkpoint report may be required at a specified frequency (weekly, monthly, other) or on the dates where milestones have been scheduled.

With fixed frequency reporting, the reporting period should be long enough for staff to not consider the report as a burden, but short enough that the report remains concise and relevant.

Highlight report

The recommended structure provides the current status of each work package and each product (artefact) and reports on any corrective actions taken, plus projections for the next reporting period. In addition, the project manager should summarise the tolerance situation across the project as a whole, itemise the status of any change requests and any changes in risk status.

Handling exceptions

The thin vertical bars on Figure 6.2 show how the regular flow of checkpoint and highlight reports from work package, through the project manager, to the project board and identification of an exception condition, where a product is likely to be later, lower quality or more expensive than planned. The project manager may, if necessary, propose some corrective action (an exception plan) as part of the **Control a Stage** process. This is then submitted to the project board as a proposed modification to the current stage plan.

The project management perspective

Communication

Communication has been at the heart of the discussion about governance and it is at the heart of project management in general. If information needs to be shared, then a formal document structure acts like a checklist to make sure nothing is missed out and a peer review makes sure that it will be understood. Remember the four C's? (Cornerstone 6 from Chapter 1 of the subject guide.)

There are numerous channels (other than formal reports) which fit different purposes and different people:

- An occasional early evening 'gathering' followed by pizza, aimed at newcomers to the team.

- A Friday afternoon 'show and tell' when a project is rolled out.
- A regular hour long Monday morning or Friday evening debrief aimed at management (this is what I did last week and this is what I aim to do next week).
- A daily five minute heads-up before work starts.
- The team coffee break ('let's get out of this room and talk about it').
- Whiteboards (this is what we're doing), blogs and links pages, wikis (collective memory).

However, if something has to be agreed or signed off, it normally needs to be written down or said face-to-face.

Motivating people

When people talk about motivating staff, they tend to mean incentivising them. People are motivated by different things: wealth, popularity, security, dominance, respect, achievement. Cockburn talks of 'rewards' – 'pride in work', 'pride in accomplishment' and 'pride in contribution'. Achievement is a motivation that you, as project manager, have some control over because you can give people tasks that they like doing and are able to perform well.

This is not just a question of people having a personality-type that makes them likely to fit a given role but of behaving in an appropriate way for the role. The significance is that you cannot modify your personality in the way that you can change your behaviour. Cockburn and Highsmith (2001) say that 'Agile development focuses on the talents and skills of individuals, moulding the process to specific people and teams.'

Given a team member whose productivity is dropping and whose relationships with other team members is becoming erratic, do you offer them redundancy, a performance-related pay rise, a desk away from the other team members, or a different role in a different team?

The answer is, of course, none of the above until you have got to the root of the problem, but assuming the alternative is losing somebody you have invested time and effort in developing as a member of the company, the new team role is normally worth trying.

CORNERSTONE 13

A team is not a bunch of people with job titles, but a congregation of individuals, each of whom has a role which is understood by other members. Members of a team seek out certain roles and they perform most effectively in the ones that are most natural to them. (Dr R.M. Belbin)

Working in teams

A quick summary of the nine Belbin team roles (see www.belbin.com or www.srds.co.uk/cedtraining/handouts/hand40.htm) should be a sufficient introduction to the team roles theory.

Plant	The creative problem-solver.
Monitor evaluator	The impartial judge.
Co-ordinator	Dominant positive thinker who focuses on objectives.
Resource investigator	The networker.
Implementer/ Company worker	The practical strategist who does the jobs others will not.
Completer finisher	The team's quality assurer.
Team worker	The oil that keeps the team running smoothly.
Shaper	The source of the drive that keeps the team focused.
Specialist	In-depth knowledge of a key area.

A successful team will normally include one or more of each of the above, with people sometimes taking secondary roles in small teams. Each role is likely to lead to an acceptable weakness.

Pressman lists key attributes of a team (not just an agile team) and its members as follows:

- Competence
- Common focus
- Collaboration
- Decision-making ability.

Agile teams have additional attributes:

- Fuzzy problem-solving ability
- Mutual trust and respect.

Self-organisation of a team was identified by Ken Schwaber (2001):

> 'Nothing de-motivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments it made itself.'

In SCRUM, the title of 'project manager' or 'team leader' is avoided by the term 'Scrum master'.

Agile project management

Agile methods were introduced in Chapter 1, where the idea of 'serial in the large, iterative in the small' was attributed to Scott Ambler. Here, where the focus is on establishing project management procedures for your project, it is necessary to consider whether an agile approach is about 'doing different things' or 'doing things differently'. The way we have interpreted this is that a project run on agile principles does the same things, at a high level, as a conventional project, but does them differently in order to do them better.

By high level, we mean that all projects should be about:

- Helping customers to discover and agree the purpose of the project.
- Developing software that provides an appropriate trade off between the various non-functional requirements.
- Supporting the integration of the result into the customer's operational environment.

By doing things differently (in order to do them better), we mean that there are various potential benefits of an agile approach:

- Avoiding a comprehensive requirements analysis of areas that are likely to change, since that will not add value for the customer.
- Making the step-size in an incremental pattern of releases as small as possible in order to remove waste.
- Being prepared to respond to the customer's priorities and to maximise the flow of value, by scheduling frequent small iterative releases of additional functionality.

There are many case studies of how well or badly agile methods have been used, but the fundamental reason for taking an 'agile approach' is that it will enable things to be done better. This tends to mean wholehearted commitment to a recognised agile methodology such as SCRUM (see: www.scrumalliance.org/pages/what_is_scrum) or XP (see: www.extremeprogramming.org).

We were once asked to help a client work out why a project was going wrong and the answer was evident. The project manager had a list of functional requirements which

were: (a) very small and largely independent steps; and (b) fairly predictable, which led us to believe that they were the project manager's understanding of the problem, not the customer's. The project manager had allocated them to individual team members together with estimates of how long the component would take to produce. This estimate included developing a specification, shared with other members of the team. The only visible management information was a set of tables, one per fortnight, listing the releases.

This situation reflects a number of the most visible attributes of 'agility', specifically from SCRUM, but the result to date could not be described as 'doing things better':

- The project manager was effectively 'in denial' and frequently absent.
- The only information available to the customer was that every item in the table of releases was late and the only way to find out why was to talk to the individual developers.
- The developers felt, understandably, that their project manager had let them down and their jobs were on the line if 'their' component was responsible for the customer's anger.

The solution was radical but effective:

- The entire project team, excluding the project manager, was transferred to come under the customer's direct supervision.
- All development work was frozen while the customer and the development team reviewed the scope and status of the project.
- The inter-dependencies between components were documented and a short-term list of priority modules was agreed between the customer and the developers.
- The customer's 'go live' date (the cause of much of the pressure once delays appeared) was replaced by the first of a series of 'alpha' releases for QA purposes. The customer agreed that although the 'go live' date was important for marketing reasons, marketing was (at this point in time) less important than having a product to sell.

Once the viability of the project had been assured, the development team (three people) took the decision to assign themselves specific roles. The team consisted of:

- a **plant**, who took responsibility for the system architecture level of the project, including database services
- a **shaper**, who took responsibility for the development environment, re-use libraries and change requests
- a **completer finisher** who took responsibility for quality assurance and user interface design

The product went live six months later.

The development environment

The UP explicitly recognises the need to manage the development environment with its 'Environment' discipline which is involved in assembling and configuring tools and services at the beginning of each phase, and providing support throughout the phase. Importantly, it is also the discipline that is responsible for defining project level processes (or procedures) such as documentation, change control and so on.

The scope of environment management, therefore, includes templates and guidelines for modelling, documentation and reporting, coding conventions (naming, scoping, defaults, exceptions, etc.), as well as tools such as in-house re-usable libraries (patterns, classes, components, sub-systems), version management, system configuration tools and problem tracking, the host development environment (syntax-aware code-editing and compilation, debugging and testing tools (including a simulated deployment platform if required) and graphical editing tools and templates).

Configuration management

Configuration management is central to the efficiency of the development process and the accuracy of the final result. At its simplest, it reconciles the different versions of components created during the evolution of the software system with the various configurations of those components in each of the versions or 'builds' of the complete system or a sub-system. This is represented graphically, although not comprehensively, in Figure 6.3. It does not, for example, reflect the complexity of what are called the 'configuration items' – software, data, configuration files, libraries and sub-systems, an installation script and documentation.

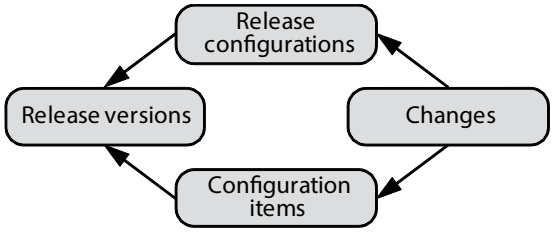


Figure 6.3: Configuration management.

The diagram does not show processes for managing change requests (see Chapter 9 of the subject guide) or defining the functionality of a release or increment. Each release version has a baseline, which is a 'frozen' configuration that can be recovered from the archive versions of configuration items. *IEEE standard 828-1998* (see http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?reload=true&punumber=5835) is a basis for defining your own configuration management plan.

Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- describe the main functions of a project manager
- understand the dynamics of a development team
- identify the tools and infrastructure required for the development environment
- describe the roles and responsibilities for configuration management.

Test your knowledge and understanding: management practices and team roles

This chapter focuses more on project management than on software engineering. However, the two need to be working side by side.

1. As an activity, consider the project management skills you have learned in this chapter and their importance in each of the UP phases and activities in Figure 1.6. Produce a table that allows you to document your thoughts based on the following:

Management technique	Inception phase	Elaboration phase	Construction phase	Transition phase

   Cornerstone 1 differentiates between the creative and flexible nature of a project manager and the formulaic approach of a project administrator. It is important that you are able to demonstrate the ability to apply a variety of techniques to keep your project on track.

2. Secondly, explore the published material on the web about Belbin 'team roles' (such as: www.srds.co.uk/cedtraining/handouts/hand40.htm).

   Now, try to establish your own preferred team role and any secondary role that you fit.

   Do you accept the described weaknesses? Now, think about how you could use this technique to work out:

i. An office seating plan for a team of eight people.

ii. The ideal team of three roles to send in to a critical meeting with a client.

Summary of the elaboration phase

- A system model defines the proposed response to requirements, showing the interaction between the system and its users as well as structural and behavioural views of the system and a representation of the boundaries of the system, its context of use.
- UML use case diagrams and sequence diagrams represent interactions; class diagrams represent the static structure of the system; and activity diagrams and state diagrams can be used to represent dynamic behaviours.
- A software architecture describes decisions about how a system is organised conceptually, logically, physically and functionally and how it will be implemented. An architectural pattern documents knowledge about generic architectures such as Model-View-Controller, Layered Architecture, Repository, Client-server and Pipe and Filter.
- A project manager is supposed to deliver acceptable project results on time and within budget. A software project manager faces three challenges not experienced by other kinds of engineering practice: the output is intangible, the processes are less mature and projects are often novel or innovative. These challenges have implications for four key management disciplines: risk management, quality management, change management and people management.
- Effective risk management is about proactively identifying, analysing and minimising the consequences of threats to the project.
- Relatively small and cohesive teams are normally more effective. Members of a team should be well-motivated. Good communication is essential.
- The cost of a software project may be estimated from experience or algorithmically (e.g. COCOMO) but the price of a project normally also depends on market forces, the level of risk for the developer and, sometimes, the ability to adjust the scope of the project to allow a more 'competitive' price.
- Scheduling matches resources to activities over time, taking account of inter-dependencies. Milestones are control points within that schedule that test whether a project activity is executing to plan.
- Plans may address the entire project, the current iteration or phase of the project leading to a major release, or an agile window of as little as two weeks (XP).
- The modular and evolutionary nature of software demands that changes are introduced and documented in a controlled way at the component level (version management), while the components are being configured and assembled into an executable image and related documentation (system building) and after acceptance testing (release management).

By now, you should be confident that you would be able to:

1. Validate that the current project plan and architecture design are complete and able to deliver the project vision and requirements.
2. Demonstrate that major risks have been identified and that each is being managed appropriately.

Part 3: Construction phase

Chapter 7: Detailed design

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- apply object-oriented principles in the detailed design phase
- describe the use of UML documentation for decomposing the architectural design into usable object-oriented software
- explain the purpose of design patterns and give examples of design patterns for data-processing, process-control and e-commerce applications.

Essential reading

Sommerville	Pressman
Chapter 7: Design and implementation Chapter 16: Software re-use Chapter 17: Component-based SE	Chapter 10: Component-level design Chapter 12: Pattern-based design

References cited

Gamma, E., R. Helm, R. Johnson and J. Vlissides *Design patterns: elements of reusable object-oriented software*. (London: Addison Wesley, 1994) [ISBN 9780201633610].

Lethbridge, T. and R. Laganière *Object oriented software engineering: practical software development using UML and Java*. (London: McGraw Hill, 2005) [ISBN 9780077109080].

Martin, R.C. 'Design principles and design patterns' 2000; www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

Overview

The relationship between modelling, design and implementation

The modelling of system architecture, as described in Chapter 5 of the subject guide, begins with a description of the context in which the system will exist. This is then decomposed into a set of scenarios or use cases that define the functionality of the system. The design process takes that decomposition a step further, identifying the components that will be needed in order to implement the required functionality. By giving the components interfaces, the designer describes the way they connect together and interact.

The preceding paragraph is completely generic and agnostic about the methodology being used. It will be equally true of a set of scenarios described using:

- Data flow diagrams
- UML '4+1' views
- XP story cards
- other structured natural language descriptions.

It will be equally true whether the development team envisages a suite of processes within conventional software modules or is following an object-oriented approach of

stepwise refinement of a set of high-level business objects. It will be equally true of a project to deliver an embedded system as it is of a web application. Let us continue in generic fashion, talking only of components.

There are three types of component:

1. Functional sub-systems.
2. Enablers that allow other components to execute.
3. Interfaces to the underlying functionality of the platform.

For example, in an MVC web application, one would find code that:

- builds an HTML page reflecting the current state of the application
- extracts the data from the application to populate the web page
- binds the application to a user's browser using HTTP requests.

Robert Martin (2000) summarises a list of basic design principles. These are (expressed in generic terms):

Open-closed	A module should be open for extension but closed for modification.
Liskov substitution	An extended module should preserve the functionality of the original.
Dependency inversion	Depend on abstractions, not on concretions.
Interface segregation	Many client-specific interfaces are better than one general purpose interface.
Release re-use equivalency	The granule of re-use is the granule of release.
Common closure	Parts of a module that change together belong together.
Common re-use	Parts that are not re-used together should not be grouped together.

Example of an object-oriented approach

In Chapter 5 of the subject guide, we introduced a scenario based on an academic publishing management system, specifically the manuscript review process as it applied to three Actors (Author, Editor and Reviewer).

At a context level, we also need to represent the manuscripts that are submitted to the system by the authors. At a more detailed level, we still need to understand how the review process works in the existing manual system and how it could be automated in the new system. The activity diagram in Figure 7.1 describes the manual process as it applies to each of the three actors.

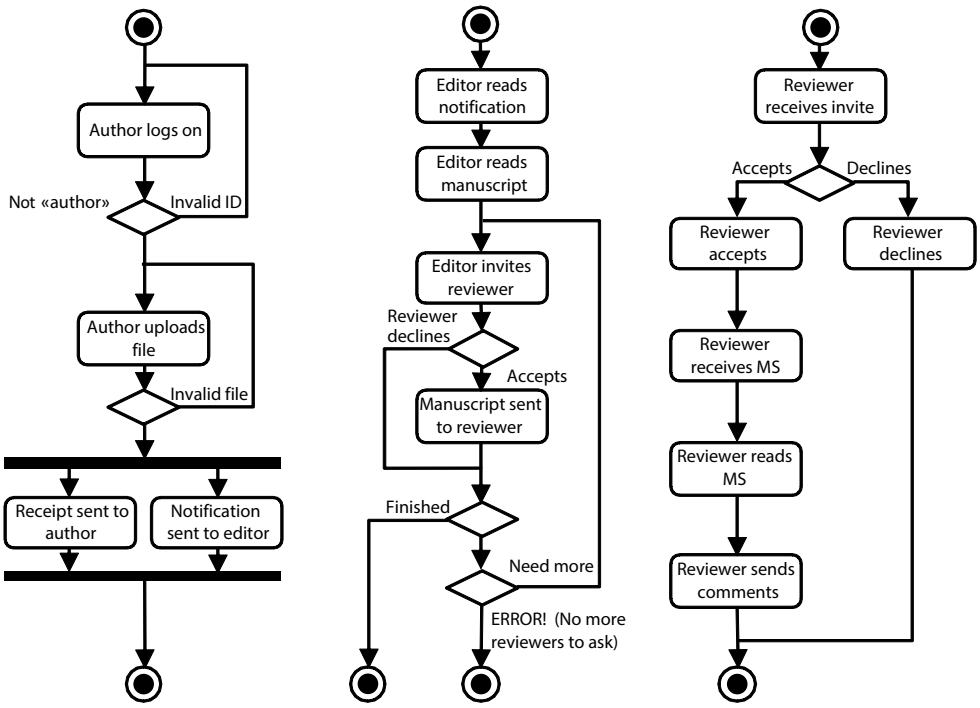


Figure 7.1: Example of an activity diagram.

By the time you start on component-level design, you are identifying the 'support' classes that are required by the main business objects, assigning properties to those classes and giving them methods that allow other objects to access and possibly modify those properties. For example, a 'manuscript' is a word-processed file.

Should the system's main data object be derived from a file object, with an extra status property («in review», «being revised», «approved», «published»)? Or is there value in having a «manuscript» class that can 'contain' a file but can also have the state «empty»?

What is the relationship between the «manuscript» object and the review process? Should a «manuscript» object contain a list of reviewers? Or should there be a «review panel» object that holds a reference to the «manuscript» object as well as a list of reviewers?

This is typical of the kind of dilemma that software developers face, and one of the reasons for XP's use of pair programming. Two of the aspects of good design (Lethbridge and Laganière, 2005) that developers need to work through are:

Cohesion

Cohesive class design encapsulates functionality efficiently – each component supporting a single primary function or set of functions related to a type of data, drawing on services from lower-level service layers, not higher-level ones.

Coupling

Coupling is a form of dependency between two otherwise unrelated classes based on a need to expose properties or methods of one class to the other, creating complexity and risk of error. Lack of functional, structural or data cohesion can cause the following dependencies:

- **Functional**: when one object or component has to control functionality that exists inside another, includes or imports another, uses another as a parameter or relies on functionality of another.
- **Structural**: when several components rely on a global variable, or communicate with infrastructure components.

- **Data**: when one component modifies the data of another, when two components share (and pass to each other), long complex data structures or when a variable in one component is declared as being of a data type defined in another.

None of these design considerations are always avoidable or inherently 'wrong' but a decision has to be reached after considering a number of 'what if …' scenarios. It is just one example of the general case that if the answer is simpler then it is probably better, otherwise known as the '**KISS principle** (**K**eep **I**t **S**imple, **S**tupid)'.

Agile object-oriented design

In Chapter 6 of the subject guide we said: '…a project run on agile principles does the same things, at a high level, as a conventional project, but does them differently in order to do them better.'

In terms of detailed design and implementation, an agile project would, in simple terms, have reached the construction phase with less effort spent on formal documentation and more effort spent on learning from the customer and the use of prototypes.
There are two critical differences between agile construction and more conventional development practices that can be traced back to the waterfall. Firstly, design-modelling, implementation and validation are executed in parallel, not in sequence. Secondly, this should allow a larger number of smaller increments to reach the customer. The desired outcome of these two changes is that developers can respond more rapidly to change and errors get detected earlier. There are specific features of each of the main agile approaches.

XP

'**E**xtreme **P**rogramming' (**XP**) – a summary:

- Maintains a strict adherence to the use case or user story – no speculative additions to the scope.
- Rapid resolution of design problems using prototyping (known as spike solutions).
- Commitment to object-oriented design, exemplified by the use of CRC cards.
- Use of refactoring to allow 'continuous improvement of the internal design'.
- Test-driven prioritisation.
- Pair programming.

SCRUM

- Compression of iterated lifecycle (requirements, analysis, design, evolution, delivery) into 30-day SPRINTS with no changes introduced.
- Goal of the SPRINT is defined by analysis of outstanding issues (the BACKLOG).
- Daily 15-minute team meetings.
- Partially functional software increments (the DEMO).

DSDM

'**D**ynamic **S**ystems **D**evelopment **M**ethod' (**DSDM**®) (™ Dynamic Systems Development Method Ltd.)

- Begins with feasibility study and business study.
- Iteration of prototypes demonstrating the emerging functional model.
- Refinement of prototypes (following 80:20 rule) in design and build iteration.
- Implementation of the (operationalised or 'acceptably incomplete') prototypes.

Key concepts: design patterns and re-use

Sommerville divides re-use into four categories, shown below, with examples.

Abstraction	Architecture/design patterns
Objects	Libraries included with tools like c++ and Java libraries
Components	ORBs for CORBA, the .NET framework, JavaBeans
Systems	COTS packages and ERP systems

Design patterns

Design patterns are a way of 'encapsulating experience'. They are like architectural patterns, except in terms of numbers. Gamma et al. (aka 'The Gang of Four') have published a book of these (Gamma et al., 1994), to which Michael Mahemoff added a table of contents and some useful learning aids (see: http://mahemoff.com/paper/software/learningGoFPatterns). The Hillside Group have produced a catalogue of design pattern resources (see: http://hillside.net/patterns/patterns-catalog).

The original 'Gang of Four' book provided a simple structure – each pattern had a name, a statement of the problem it addressed, a statement of the solution and analysis of the consequences or constraints.

Pressman dedicates a chapter to the use of 'creational' patterns that encapsulate functionality within a class with an interface, but hide the details of how it is constructed, structural patterns that describe the way classes can be organised and aggregated and behavioural patterns that provide solutions for how classes can 'cooperate' and interact. He also introduces the idea of a framework, executable software that accepts implementations of patterns as 'plug-ins'. Sommerville provides additional material in his web resources section.

As patterns have become more widely adopted, creators of patterns (especially those intended for in-house use) have tended towards making a pattern a more general purpose container for design documentation. Patterns provide a higher level description of a concept that can be re-used than a component encapsulated as a black box with an interface.

Design for re-use

Sommerville discusses, in Chapter 16, issues of software re-use, both in terms of re-using existing software such as application frameworks and COTS, but also in terms of developing software that can be re-used. You may also wish to read Chapter 17 on 'Component Based Software Engineering' and Chapter 19 on 'Service-Oriented Architecture' (SOA).

Sommerville identifies 14 approaches to re-use, only some of which have been covered in this subject guide (architectural and design patterns, component-based development, application frameworks, COTS, language-specific libraries). Students with Pressman's textbook should explore the following additional sources of re-usable software using the internet:

- legacy system wrapping
- service-oriented architectures
- product lines derived from an existing architecture
- ERP systems
- configuration of generic vertical applications such as accounting packages
- domain models
- programme generators
- aspect-oriented software development.

Re-use can speed up the development process because it allows the use of software that has already been verified and validated as a solution to a specific problem (possibly one that required specialist knowledge). This in turn leads to the benefit of a known cost for re-use rather than the more open-ended question of how much it would cost to develop without re-use. One additional cost to be considered in a re-use scenario is the cost of finding the right component to use – research to locate something that probably does what you need (a similar problem to the identification of design patterns) plus the cost of validating that it is: (a) robust enough; and (b) compatible with your development environment.

These criteria apply equally when you are considering how to make your own software re-usable. You need to describe the purpose and any constraints and validate the reliability of the code and logic. One of the scenarios would be a development team that has a corporate policy for internal re-use, where there are also business decisions to be taken, such as whether the additional cost of packaging and supporting the end result is justifiable.

If the solution is part of a framework, for example, do the other components that are compatible with the framework address a broad enough set of application types and domains? The Open Source paradigm, which provides another way of accessing and providing re-usable software, is too complex to address here.

The project management perspective

Ensuring software quality and reliability

Chapter 8 of the subject guide addresses the day-to-day principles of ensuring the fitness for purpose of the outputs from the chosen development methodology. Here our concern is slightly different – have the correct decisions been taken to ensure that the chosen development methodology is appropriate for the project?

A checklist is presented below:

1. Functional and non-functional requirements
    - Was the analysis exhaustive, in which case does the project plan make suitable allowance for the inevitability that some of those requirements will change?
    - If the specification of requirements is incomplete, does the release schedule allow for suitable iterations of design and increments of delivered functionality?
    - Does the customer understand the implications of this?
2. Architecture and system design
    - Has an architectural pattern been identified?
    - Has the possibility of acquiring re-usable components or design patterns been evaluated?
    - Is there a requirement for this project to deliver re-usable components?
    - Does the development team understand the implications of this?
3. Construction
    - If you have decided to adopt an object-oriented and/or agile approach to the construction phase, does the development team have the relevant experience?
    - Have suitable metrics been established for monitoring progress?
    - Are appropriate plans for communication and reporting procedures in place?
    - Is the development environment fit for purpose?
    - Does your line manager understand the implications of this?
4. Signing off the elaboration phase
    - Is there a quality plan for the project?
    - Is there a suitable test plan in place?

- Is the risk management plan up-to-date?
- Is there a suitable change management plan in place?
- Do you, as project manager, understand the implications of this?

Standards

One additional bullet point that could be added to the list above is whether the necessary standards have been identified and are in place. The important thing is that standards are necessary, whether these have endorsement from the International Standards Organisation (ISO), such as the ISO 9000 family of quality standards, or are simply determined in-house as answers to regular problems.

Normally, references to standards will be found in the project quality plan. For ISO 9001 (see: www.iso.org/iso/iso_9000) accredited organisations, the scope of the project quality plan will be defined in a corporate quality manual.

Examples of areas where standards are needed include those listed in Table 7.1 below. (Note: this is not a comprehensive list.)

Table 7.1: The need for standards.

Scope	Reason
Structure of software requirements specification and other documents with contractual ramifications	Ensure readability and completeness.
Use of UML or similar frameworks	Ensure accuracy and readability.
Application of metrics for progress reporting	Ensure consistency.
Style of source code documentation	Ensure maintainability.
Analysing, documenting, reporting and responding to issues: • Risk management • Change management	Ensure decisions are taken at correct time by the correct person (e.g. no changes in specification allowed during a sprint).
Documentation of decisions	Ensure decision is known (including responses to issues).
Document history	Ensure changes are known and correct version is in use.

Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- apply object-oriented principles in the detailed design phase
- describe the use of UML documentation for decomposing the architectural design into usable object-oriented software
- explain the purpose of design patterns and give examples of design patterns for data-processing, process-control and e-commerce applications.

Test your knowledge and understanding: development strategies

Detailed design is driven by requirements and architectural design. We have emphasised re-use through design patterns and object oriented approaches. It is important that you can demonstrate an understanding of the ability to apply these appropriately.

Each of the Essential reading textbooks (Pressman and Sommerville) has one or more case study applications that illustrate the key points in each chapter. Take a look at these.

Now, build a decision tree to show how the best strategy for the construction phase can be selected for one of the case studies.

Notes

Chapter 8: Quality management

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- define quality as it applies to software engineering and identify the key quality-related activities that take place at each phase of the project lifecycle
- explain the advantages and limitations of qualitative and quantitative quality measures and give examples of each
- explain the difference between verification and validation and the purpose of testing at the sub-function, component, system and user levels.

Essential reading

Sommerville	Pressman
Chapter 8: Software testing	Chapter 17: Software testing strategies
Chapter 24: Quality management	Chapter 14: Quality concepts
	Chapter 15: Review techniques
	Chapter 16: Software quality assurance

Further reading

Linger, R. 'Cleanroom process model', *IEEE Software* 11(2) 1994, pp.50–58.

Myers, G. *The art of software testing*. (Oxford: Wiley, 2011) third edition [ISBN 9781118031964].

References cited

Chidamber, S.R. and C.F. Kemerer 'A metrics suite for object-oriented design', *IEEE Trans. Software Eng.* 20(6) 1994, pp.476–93.

Fagan, M.E. 'Design and code inspections to reduce errors in program development', *IBM Systems Journal* (15)3 1976, pp.182–211 (available from IBM under Reprint Order No. G321-5433).

Kan, S.H. *Metrics and models in software quality engineering*. (London: Addison Wesley, 2002) [ISBN 9780201729153].

Overview

PRINCE2 defines the purpose of a '**Q**uality **M**anagement **S**trategy' (**QMS**) as:

> '…describing the quality techniques and standards to be applied, and the responsibilities for achieving the required quality levels'.

The term most often associated with the quality of a project's output is that it is **'fit for purpose'**. In other words, the output satisfies project requirements **to an acceptable extent**. As discussed in Chapter 6 of the subject guide, the goal of a project manager is often to achieve the best possible trade-off between conflicting requirements. However, a statement of requirements may mean different things to different stakeholders (think of the interpretation of phrases like 'reasonable response time') making verification of compliance more or less impossible.

Stephen H. Kan (2002) describes a number of quality-related frameworks used by the IT industry, including 'CUPRIMDSO' – which stands for **C**apability (namely, functionality), **U**sability, **P**erformance, **R**eliability, **I**nstallability, **M**aintainability, **D**ocumentation, **S**erviceability and **O**verall customer satisfaction. This is a refinement of the broader FURPS quality model from the engineering sector, described in Chapter 2 of the subject guide.

Quality frameworks such as CUPRIMDSO concentrate on the 'ilities' of a system, which are mostly highly subjective statements of non-functional requirements. Concepts like accessibility and maintainability cannot be expressed in absolute or quantified terms.

Quality standards like ISO 9001 and maturity models like CMMI shift the focus away from assessment of the product (see Chapter 2 of the subject guide, regarding ISO/IEC 12207) towards assessment of process:

1. Can we validate that the processes used are able to meet requirements if applied correctly?
2. Can we verify that the process was applied correctly in this instance?

If a process is executed correctly, then the result should be consistent with the results achieved on other occasions where the process was adopted. Figures 1.2, 1.3, 1.4 and 1.5 show process models that allow a project manager or a quality manager to review the performance of each phase in the model. For example, execution of test cases determines whether the functionality being tested is correct, but formal inspection of a test plan helps to verify that it addresses all areas of functional requirements. A code inspection would check for common errors such as uninitialised variables, potentially infinite loops (control variables that can be modified), parameter type mismatches and failure to release dynamically-allocated resources. For IBM in the 1970s, the waterfall model allowed fault avoidance through Fagan inspections (1976) before the transitions between phases. The agile team of today would argue that incremental techniques like SCRUM and test-driven development achieve the same thing, but more efficiently.

ISO 9001 – processes that can be trusted

A friend who used to run a training facility described preparation for ISO 9001 accreditation. Having provided documentation of the policy for staff development for the trainers, they were surprised to be asked to provide similar evidence for staff operating the coffee machine. The consultant explained that if two classes of 30 delegates arrived for a 15 minute coffee break, then coffee had to be dispensed promptly and predictably, otherwise some delegates would be left without refreshment.

CMMI – processes that can be measured

CMMI Level 4 requires evidence in the form of metrics to establish whether the process can deliver the required quality of result (for example a cup of coffee every 12 seconds) and whether the process has been executed properly (all cups served in 12±3 seconds). The analogy could be extended to address non-functional requirements as well: temperature of coffee when served, choice of biscuits, cleanliness of cups, and so on. Note that if on a particular day the performance was 10±2 seconds, this would be evidence that the process had not been executed properly, not some kind of unplanned 'improvement'.

Quality Management Strategy (QMS)

Following the PRINCE2 definition given at the beginning of this chapter, the QMS defines three things: **What** level of quality is required? **How** is the required quality to be achieved? **Who** is responsible for achieving it?

If your organisation is accredited for ISO 9001 or CMMI Level 3 or higher, you have a ready-made framework, but it still needs to be populated with the specific requirements of the project. The recommended textbooks both provide detail of quality procedures and test strategies. Table 8.1 summarises just when in a typical project lifecycle the procedures need to be deployed.

Quality review is shown in the table as being associated with phase-end milestones but quality assurance is a continuous activity that includes review of the following:

- **Compliance** with agreed standards and conventions including use of document templates, configuration management procedures and conformity with programming style rules.
- **Application** of the 'four "C's"' (see Cornerstone 6).
- **Verification** (testing and/or inspection) and responses to outstanding issues.
- **Acceptability** of products produced within the phase.

Verification addresses two of a project manager's ways of responding to the risk of faults in the software: **fault avoidance** (identifying mistakes that would lead to faults in the program if not corrected) and **fault detection** (testing and debugging). A third strategy (**fault tolerance**) applies to critical software where techniques such as redundancy and diversity reduce the risk of a failure.

Table 8.1: Quality considerations.

Workflow	Scope	Quality considerations
Business modelling	*Business processes modelled in a form understood by both business and software analysts*	Quality planning and definition of procedures (QA) Inspection of artefacts
	End of phase	Quality review as part of lifecycle objectives milestone or equivalent
Requirements	*Functional requirements*	Inspection of artefacts
	Non-functional constraints – trade-offs and decisions	
	End of phase	Quality review as part of lifecycle architecture milestone or equivalent
Analysis and design (the '4+1'views, see Figure 5.1.)	*Static structure (logical view)*	Prototyping Inspection of artefacts
	Dynamic behaviour (process view)	
	Physical organisation of components (implementation view, optional physical view)	
Implementation	*Building components and sub-systems*	Inspection of artefacts Unit testing
Testing	*Component testing*	Component testing
	System testing	System (integration) testing
	End of phase	Quality review as part of initial operational capacity milestone or equivalent
Deployment	*Acceptance testing* *Roll-out*	Customer/user testing
	End of phase	Quality review as part of product release milestone or equivalent
Support	*Configuration and change management*	Quality audit
	Project management	Quality audit
	Development environment	Acceptance testing

Test plans

Test plans ensure that coverage of possible faults is comprehensive. Fault detection, as its name suggests, can expose the existence of a fault, but it cannot guarantee that a piece of work is fault-free. A test plan covers both verification (have we built the thing **right**?) and validation (have we built the **right** thing?) against requirements. Note that prototyping at an early design stage can also contribute to the validation process.

A test plan normally covers a single complete iteration of the lifecycle. Testing can be split into four stages and users may be involved at any stage, most typically in alpha-testing at the component level (including user-interface testing), beta-testing an internally verified system and acceptance testing.

Unit testing

Verify the behaviour of the individual classes or methods for the application as defined in the process view of the design model.

Component testing

Verify the behaviour of fully-tested units when combined to form components defined in the component model. This is often described as 'white-box' or 'clear-box' testing since you are using knowledge of how the box has been built to test the use/misuse of interfaces and, in real-time systems, their timing.

System testing

Verify the behaviour of the complete system of fully-tested components (including any re-used or bought-in components) concentrating on functionality required by the use case model. This is often achieved by 'black-box' where you are not interested in how the box works, only whether it responds to external stimuli such as user action or input data. Note that one set of tests has to be for the correct response to invalid input, which is rejection.

Acceptance testing

Validate the compliance of the complete, tested, system with functional and non-functional requirements; that is, from the customer's perspective. This will include assessment of performance and overall acceptability of the quality of the result.

Test cases

Testing is non-trivial and a test plan needs to identify appropriate strategies for each stage. The recommended textbooks go into a number of test strategies in considerable detail. Here, we summarise some basic principles.

During unit testing, each unit should be exhaustively tested by invoking each function or method that it provides using the range of values of each parameter or attribute those functions require (including invalid and missing values) and the function results for each state change that those parameters cause.

For example, a logging unit that is either active or inactive requires six tests to prove a function to change the state and a further six tests to prove a function that generates a report and optionally flushes the log data:

Table 8.2 Test conditions.

Function	State	Parameter	Outcome
Log (on/off)	*Inactive*	None	Error
		False	No change
		True	Change state to active
	Active	None	Error
		False	Change state to inactive
		True	No change
Report (flush/ noflush)	*Inactive*	None	Error
		False	Error
		True	Error
	Active	None	Error
		False	Generate a report and preserve log data
		True	Generate a report and then delete log data

Note that there may be additional implied conditions that need to be tested, whether or not explicitly specified.

Activity

Try to define the additional implied conditions that need to be tested. When you have done this, have a look at the answers in the sample revised table at the end of this chapter.

During component testing, the number of discrete combinations could become unmanageable. Methods of reducing the scale of the problem include **boundary-case testing** (many errors are manifested at the edges of applicability), **equivalence partitioning** (selecting representative values for each class of expected input such as positive and negative numbers plus zero, eight/sixteen/thirty-two bit numbers, and so on) and **orthogonal array testing** (selecting a sample of parameter values uniformly from across each parameter region).

System testing can often focus on the original use cases, scenarios or user stories, since each will identify one or more functional requirements and executing tests based on the entire use case model should exercise each requirement at least once. Another strategy is to graphically determine how many discrete paths there are through the application code. Typically, this represents the responses of the system in each of its states to user input such as menu selections. By performing each of those user actions in every possible state, you are guaranteeing that every line of relevant code is executed at least once.

Acceptance testing by the customer should follow a plan that was established at the time of contract, since that is the only mechanism that a supplier and customer have of determining when the work is finished. It includes the same kind of scenario-based testing as executed internally when performing system tests, but also has to evaluate the acceptability of the compliance with non-functional requirements. The acceptance plan should, therefore, define the users for whom usability is an issue, a profile of how frequently each function is performed and how critical performance is, and an indication of the consequences of different types of failure.

Other forms of testing

Regression testing

Once a test has been performed, the validity of the result only lasts until the next time the system is changed, since even apparently unconnected faults elsewhere may compromise the object that has been tested. For example, another process may exhaust the supply of file handles which the tested unit assumed to be essentially limitless. Executing a complete set of manual tests every time a new build is achieved would be unwarranted, but by capturing the details of the test and automating it, this becomes a manageable problem.

Agile test-driven development

Agile methods based on frequent small increments can be tested by considering each increment as if it were a new system build. Because the increments are small, there is limited opportunity for low-level errors to be introduced and the focus can be on the functionality addressed by the increment. However, in a test-driven development process where refactoring of existing code is performed at each iteration, regression testing becomes essential to ensure the stability of the existing code.

Testing web-based applications

Web-applications either present static information (which still has to be 'tested' for spelling mistakes, level of language, accessibility and so on) or an interface to a database-driven system (see Figure 8.1). In either case, a discrete set of elements of system testing must be applied, corresponding to the various elements of application design.

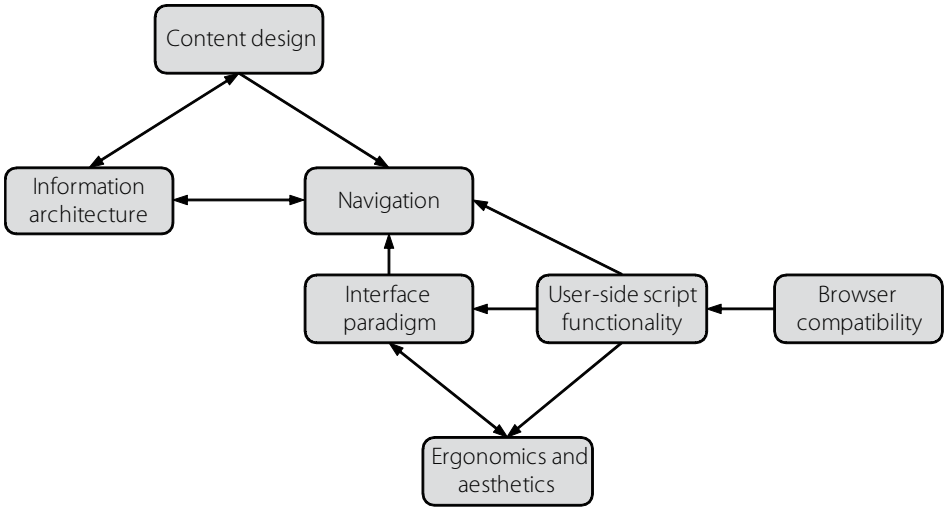


Figure 8.1: Web-application design considerations.

Key concepts: quality assurance, quality control and quality management

These three terms are often used interchangeably, but it is important to differentiate between the three separate areas of responsibility. Put simply, **quality management** is a process that defines the **quality assurance** targets and **quality control** techniques that deliver quality to the customer.

Quality management

Quality management is a parallel activity to project management, sometimes undertaken by the same person, which establishes the quality goals for a project. The quality management plan defines the areas of responsibility for quality assurance and the techniques of quality control that should be used. The quality manager is responsible for the delivery of a quality-assured output or set of outputs from the project.

Quality assurance

Quality assurance is the achievement of a quality target by implementing quality controls. Under PRINCE2, responsibility for achieving the required level of quality is delegated to the work team building the artefact. The only quality control that cannot be undertaken by the software development team is the final system testing prior to handover, for which a special independent test team is normally deployed.

Quality control

Quality control includes the range of testing activities described in this chapter, techniques for artefact inspection and formal reviews within the development team. Reviews can be exhaustive (such as design walkthroughs) or representative (such as sampling a portion of code to ensure that naming conventions are being followed, inline comments are appropriate, code fragments are not too large and so on). Reviews should be:

1. scheduled
2. concise and purposeful
3. documented
4. focused on the work, not the person.

The project management perspective

Quality and culture

Attitudes to quality vary and attitudes towards the assurance of quality vary even more. Traits that can be found in most teams include:

Deferring important tasks that 'get in the way':

– like commenting code.

Leaving workarounds in place rather than fixing the cause of the problem:

– like commenting out an error-trap because you have no error handler for it.

Assuming someone else will do the task at some time:

– like updating the risk register.

Jumping to the next task without considering if prerequisites have been met:

– like not waiting to sign off the component diagram before starting to design message-passing protocols.

These are all attitudes towards the process of assuring quality and can be categorised as examples of: 'I've got better things to do with my time, like reaching the next delivery date on the schedule.'

Attitudes to quality as a feature of the work being done are normally positive. After all, why would anybody want to produce a sub-standard result? However, if somebody does not see that the work being done is sub-standard or they feel under pressure, 'better things' can be joined by:

- 'It's not a question of how, it's how much.'
- 'I have always done it like this.'
- 'Nobody ever looks at a design document once the project has gone live.'
- 'It will do.'

All of the above statements are indications of a problem. It is the project manager's responsibility to overcome resistance to change in the culture of the organisation, as in Figure 8.2.

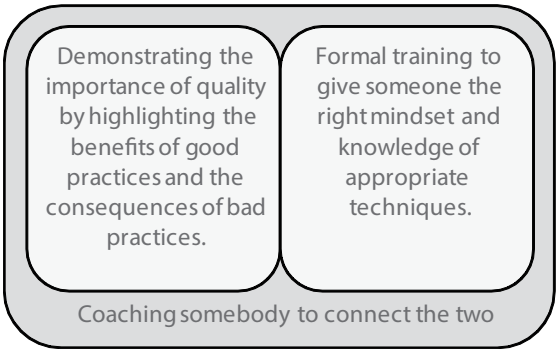


Figure 8.2: Promoting quality.

Cleanroom

The Cleanroom methodology is described by Richard Linger (listed as Further reading for this chapter) and provides a radical alternative to conventional development strategies with their system/detailed design and unit-component-system testing. However, we introduce it here, rather than in Chapter 3 of the subject guide, because its distinguishing feature is its approach to formal quality proofs rather than testing to find errors.

A Cleanroom project plan divides the project into well-defined increments, each the responsibility of an independent software team. Increment-planning focuses on the accuracy of time-to-completion assessment because of the inherent inter-dependencies leading to a critical path across the elaboration and construction phases of the project.

Detailed requirements are elicited for each increment in a conventional way. Three types of design model are produced from analysis of these:

1. A '**black-box**' functional specification defines the transition rules required by the increment to produce specific behaviour in response to a sequence of stimuli (external events). This is decomposed until a black sub-box defines a single class of behaviour. This behaviour is either defined as a mathematical function or specified in natural language.
2. A '**state box**' architectural specification defines the black box behaviour in terms of the state-transitions that result from the external events and the data and services that are required to achieve each state-transition. Thus it encapsulates (completely) the black box it is describing.
3. A '**clear box**' technical design specification defines the procedures or methods that are required to implement the services needed by the state box. Each procedure can be decomposed into clear sub-boxes – ultimately, if necessary, until a sub-box defines one programming construct that is logically indivisible (that is, it contains no conditional statements).

Each planned increment has a hierarchy of separated design elements for each increment. Each box can be formally verified mathematically, but most faults can be eliminated by applying formal assertions about the logic expressed in the design, specifically the unconditional relationship between a sequence of stimuli and a behaviour, a conditional relationship (**if then else**) and an iterated relationship (**looping**). Finally, the verified lowest level boxes can be converted into the required programming language.

In parallel with this sequence of activities, a 'statistical test' plan is developed, designed to concentrate on the statistically most likely usage patterns and the behaviours of the program that are invoked. Successful verification at the code level results in certification that the increment is fit for integration. Just as other process models would assign a special 'test team' to system testing, so Cleanroom assigns a 'certification team' to analyse the black box and determine the most efficient sample of usage patterns and their stimuli to test.

Ultimately, the weak link in a Cleanroom environment, as in other situations, is the accuracy and completeness of requirements capture.

Metrics: what to measure, how to measure and why

Performance measurement applies to all phases of a project and the metrics can be used for purposes other than indicators of quality. For example, measures of the quantity of some artefact such as use cases or functional requirements are indicators of complexity (and therefore potentially requiring more management time). Measures that correlate effort to those artefacts can be used in other projects to help estimate the effort required to produce a use case, add one functional requirement to a specification, and so on.

Some measures are not intuitive:

1. There is an inverse relationship between the number of bugs caught in unit testing and the scale of maintenance work required to remove the remaining bugs.
2. The average length of names (variables, functions, modules) is an indicator of the quality of documentation.
3. The number of bug reports for a beta release is an indicator of poor testing. You would expect the number of bug reports to increase with the number of beta testers available to find bugs but a relatively bug-free product could attract more users because it is so robust.

The degree to which cohesion (see Chapter 7 of the subject guide) is kept high and coupling low is a reflection of the design problem as well as the coding style. The effect of 'improving' these measures will be found in run-time performance (which could deteriorate because a less efficient design has been used to reduce the coupling, for example). This is an example where analysis of the metric is possibly better as a diagnostic tool to explain poor performance rather than as a measure of performance per se.

Metrics for classes

Chidamber and Kemerer (creators of the CK suite)(1994) identified six measurable attributes of a class in a schema still supported today in spite of criticism of its lack of rigour by theoreticians.

Weighted Methods per Class (WMC)

The weighting is an expression of complexity. The more methods that a class has and the more complex they are, the more difficult it will be to create, test, maintain and re-use.

Depth of Inheritance Tree (DIT)

The deeper the hierarchy of super-classes from which a class inherits properties and methods, the more complex the class becomes.

Number of Children (NOC)

The more children a class has, the more dependencies to be tested in that class.

Coupling between Object Classes (CBO)

Coupling (see Chapter 7) leads to more inter-dependencies that have to be tested when either class is edited.

Response For a Class (RFC)

The larger the number of possible ways in which an object can respond to a stimulus (a message from another object) the more complex the class that object is a member of.

Lack of Cohesion in Methods (LCOM)

The more properties in a class that are accessed by more than one method of that class, the greater the coupling of methods within the class (rather than between classes, CBO).

These are all 'static' measures, so-called because they are derived from documentation and are not extracted from the execution of the system.

Among the more useful dynamic measures are performance 'benchmarks' (load time, execution time, capacity, number of threads in use) and diagnostics (frequency of errors/ exceptions, results from 'stress tests' such as number of simultaneous users, memory leakage attributable to specific processes).

Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- define quality as it applies to software engineering and identify the key quality-related activities that take place at each phase of the project lifecycle
- explain the advantages and limitations of qualitative and quantitative quality measures and give examples of each
- explain the difference between verification and validation and the purpose of testing at the sub-function, component, system and user levels.

Test your knowledge and understanding: quality management

1. Consider the lifecycle of an agile project and of a linear waterfall approach. As an activity, identify the stages in each where the key quality management introduced in this chapter should be applied. Document your findings in a table:

Quality management tool	Use in an agile project	Use in a linear project

Quality in a software project is limited by the weakest link in the chain. Rigorous requirements validation and change management cannot deliver a high-quality product if module-level testing is not equally rigorous, for example. It is important for you to demonstrate the ability to maintain a high standard of quality management throughout the project.

2. Now, identify the five key strategies of the Cleanroom methodology and draw a detailed model of the process, showing how these essential strategies are integrated.

Sample answer to activity about unit testing

Consideration of the issues implicit in this problem leads to recognition that there are actually five device states, not two: inactive, empty, active (has content), in-error and busy. This is shown in the revised table (8.3). Verification of the system behaviour is required in each state. If the unit is in an error state, the two functions must behave appropriately (not simply raising exceptions that never get processed). Even if the unit is active, it may not have any data which means that a null report must be created. Equally, if report generation takes a long time, then we must consider the situation if a state-change is requested while the unit is busy, before the report is ready (whether transfer is by file I/O or via shared memory). Finally, if there are possible errors then there should be error-handling and that needs to be tested as well. Sommerville presents a generalised input-output model of program testing in his chapter on software testing.

Note also that there is a potential timing/synchronisation issue with the shared memory model, in that Report(flush) could coincide with the creation of a new log entry.

Table 8.3: Implicit test conditions

Function	Log State	Parameter	Outcome
Log (on/ off)	*Inactive*	None	Error
		False	No change
		True	Change state to empty
	Empty	None	Error
		False	Change state to inactive
		True	No change
	Active (has content)	None	Error
		False	Change state to inactive
		True	No change
	In error	None	Error
		False	Error
		True	Error
	Busy	None	Error
		False	No change
		True	No change
Report (flush/ noflush)	*Inactive*	None	Error
		False	Error
		True	Error
	Empty	None	Error
		False	Generate a null report
		True	Generate a null report
	Active (has content)	None	Error
		False	Generate a report
		True	Generate a report, flush log and change state to empty
	In error	None	Error
		False	Error
		True	Error
	Busy	None	Error
		False	No change
		True	No change

Summary of the construction phase

- The models that represent a software architecture have to be iteratively refined (the size of the step depends on the methodology) and developed into descriptions of components (objects) within a system, through definition of their behaviours and interfaces.
- Since behaviour is encapsulated within the software implementation of an object, components of a system can be re-used if their interfaces are published and a new system can re-use existing components.
- An alternative form of reuse is achievable through Open Source software, which encourages modification of object behaviour.
- An Integrated Development Environment allows development to take place without transferring software to the target machine for execution. Configuration management within the IDE is central to system evolution and the management of change.

- Testing indicates the presence of error. It does not establish the lack of errors. Test cases based on typical usage scenarios and test cases that have been successful before should be selected.
- Developers test their own object and methods (unit testing), components built from those tested units and the partial and complete systems assembled from tested components. Ideally, a separate team should test a system before it is released. Final testing, including acceptance testing prior to deployment, should be the responsibility of system users.
- Development standards and best practices should be documented in a Quality Assurance manual. Together with performance metrics generated by QA procedures and formal testing, these are the main quality management tools available. Reviews are the most commonly used techniques, both to provide peer review at a formative level and to verify compliance with quality standards of the deliverables.
- Metrics can be used to benchmark a system prior to a change, to identify anomalies at the component level or to characterise the system as a whole.

By now, you should be confident that you would be able to:

1. Validate that the system being developed is fit for purpose.
2. Deploy it for beta-testing with the target user community.
3. Provide those users with the necessary documentation and support, including training.

Part 4: Transition phase

Chapter 9: Evolution

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- describe the key decisions that have to be taken when scheduling the next iteration of a system
- explain how process effectiveness can be measured and improved with explicit reference to GQM and CMMI.

Essential reading

Sommerville	Pressman
Chapter 9: Software evolution Chapter 26: Process improvement	Chapter 29: Maintenance and reengineering Chapter 30: Software process improvement

References cited

Cockburn, A. *Agile software development: the co-operative game*. (Boston: Addison Wesley, 2006) [ISBN 9780321482754].

Lehman M.M. and L.A. Belady *Program evolution: processes of software change*. (London: Academic Press, 1985) [ISBN 9780124424418].

Overview

This chapter is about the relationship between product (or system) and process and the role that process plays in determining fitness of the system over time. Improving the quality of the end result (or just keeping pace with the rate of change of requirements) raises a number of questions about whether the development process has placed sufficient emphasis on supportability.

Evolution of the product

One aspect of software engineering that all commentators will agree on is that it is important that the effort involved in the chosen development approach is proportionate to the project being delivered. Since evolution is part of the development lifecycle, the processes used during the project (such as design documentation, test plans, configuration management and so on) remain important after the project has been 'rolled-out'. However, in a fast-moving incremental project, the rate of change can be difficult to keep up with. This is particularly important in a 'depth first' incremental project which is supporting incremental versions (real, live releases) that are being rolled out for the client to use. For example:

A fault is reported on a newly-delivered system that is being used by several of the client's offices. Debugging reveals an assumption that has been made which is unsafe in one specific regulatory environment. The client is informed and the developers are mandated to remove the incorrect default value from the relevant on-screen form. An updated version of the module that implements the view is shipped to the office that reported the fault which verifies that the fault has been rectified. A notification is published in the client's in-house newsletter, but no other offices request an update.

The consequences are:

1. The requirements documentation still defines the unsafe default, meaning that if the system is ever re-engineered from the original specifications, the problem will be re-introduced.
2. The formal test-plan for the reworked module has not been updated, so that the module would fail a test that it had previously passed, should the system ever need to go through Verification and Validation again.The baseline in the configuration management system has not been updated, exposing the client office to the risk that their 'repaired' module will be replaced by the original when the system is updated more formally, unless the module in question is also updated.
3. The code-base now includes a form with no default value for one of its fields and no record that the default has been removed, with the likely result that if the module is ever modified, the new baseline will behave in an unexpected and unexplained way.

Since the client's helpline has not been informed of the change, it will be unable to help.

To avoid this problem, a planned approach to evolution is required, recognising the existence of external drivers (essentially, customer requirements such as changes in the regulatory or business environment) and operational drivers (such as performance issues or 'downtime' due to uncorrected faults). These trigger a process that may lead to a decision to generate a new release of software, changes in customer-support, user-training or product documentation. The need has to be analysed and a business case made for the change. This will include consideration of the impact of not responding to the change (the 'cost of doing nothing') balanced against the estimated cost of developing and rolling out a new release, as represented in Figure 9.1. Note how low-priority updates may get deferred until the next iteration of the cycle.

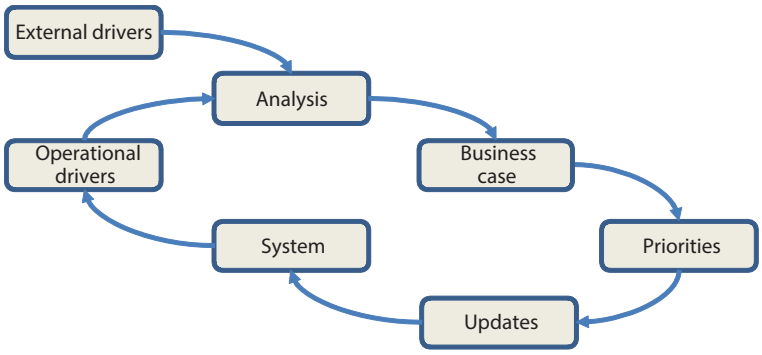


Figure 9.1: Operational and external drivers for change.

Process improvement

Four factors affect the ability to improve a software development capability:

- tools used for software engineering
- time, budget and resources available
- quality of people
- quality of process.

Process improvement addresses the last of these. We can define a process as 'an activity or workflow that has an outcome for the team or part of the team'. This definition differentiates between: (a) activities that add value to the team; (b) activities that produce waste in the form of outputs that are not used; and (c) 'private' activities that only benefit the individual who does them and do not contribute to the flow of value to the customer.

The quality of the process may be defined – and measured – in various ways:

1. How easy is it for people to perform the process? Is it well-documented? Is it used consistently?
2. Are the benefits clear? Is the process visible and is it measurable? Is it acceptable to those using it?
3. Is the process reliable? Is it robust? Is it easily maintainable? Is it supported (for example, through user training?
4. Is the process efficient? Is it automated where appropriate? Do the benefits outweigh the overheads?

Agile process improvement

Perceived net benefit is the key determining factor in whether the development team will support the use of a process. The outcome will depend on whether the goal is project-based improvement (the agile approach) or organisational improvement (focusing on CMMI)

Cockburn (in *Agile software development: the co-operative game*, 2006) makes the assertion that it is feasible for organisations to adopt agile methods if they have defined standard processes that form the basis for project-specific processes (that is, CMMI Level 3); whereas it is unlikely that an organisation using agile methods would see the benefit from investing in CMMI accreditation. The question is not whether CMMI (or quality-related initiatives like ISO 9001) prevents the use of agile methods (although objective evidence that they were being applied consistently would be required); it is whether it **adds value**.

The value of CMMI is something that each organisation that consists of more than one software development team has to answer for itself; that is, any organisation that includes an in-house SE department, or an SE business with multiple teams.

Key concepts: process improvement, maturity modelling and agility

One common approach to process improvement, '**P**lan **D**o **C**heck **A**ct' (**PDCA**), is shown in Figure 9.2. It is cyclic, beginning with identification of the process to be improved, the estimated value of changing the process and the changes to be made. It assumes that: (a) you know what the processes used are; and (b) how they really work (not how people tell you they work), in order to build and empower a team to deliver the change. It is necessary to prepare those affected by the change before the trial.

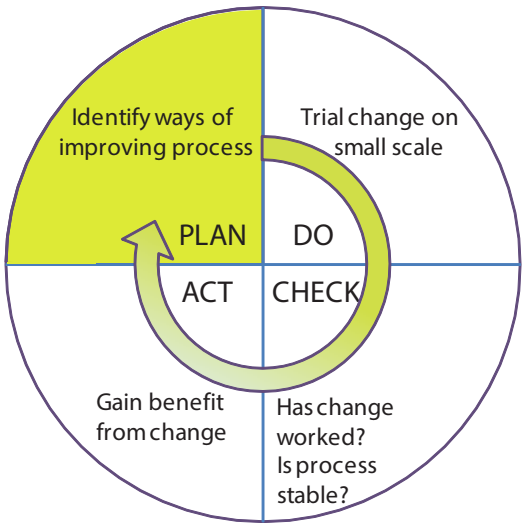


Figure 9.2: Plan Do Check Act.

Modelling the process allows you to perform any possible simplifications before setting up a data-collection framework to produce a baseline for measuring the improvement. '**G**oals, **Q**uestions, **M**etrics' (**GQM**) can be used for this.

Maturity modelling

CMMI (Version 1.3) is a maturity model that defines five levels of capability. These are often visualised as a set of steps, since it is necessary to progress one level at a time, from chaos to optimisation of the way the organisation works. The intermediate steps put defined processes in place, apply those processes across the organisation and implement measurement, rather than just monitoring, of performance. There are specific versions of CMMI for development, services and acquisitions.

CMMI Level 3 describes an organisation that has developed and deployed key processes in the areas of project management, process management and engineering. CMMI Level 3 (see: www.sei.cmu.edu/reports/10tr033.pdf) requires that:

> '…processes are well characterised and understood, and are described in standards, procedures, tools, and methods. The organisation's set of standard processes, which is the basis for maturity Level 3, is established and improved over time. These standard processes are used to establish consistency across the organisation. Projects establish their defined processes by tailoring the organisation's set of standard processes according to tailoring guidelines.'

There are 22 process areas covered by CMMI V1.3. With specific regard to project management processes, CMMI Level 3 requires evidence of:

- Integrated Project Management (IPM)
- Project Monitoring and Control (PMC)
- Project Planning (PP)
- Quantitative Project Management (QPM)
- Requirements Management (REQM)
- Risk Management (RSKM)
- Supplier Agreement Management (SAM).

The project management perspective

The need for change – Lehman's Laws

Lehman and Belady conducted a series of empirical studies of the evolution of large software systems at IBM during the 1970s (Lehman and Belady, 1985), resulting in '**Lehman's Laws**' of system change discussed in the recommended textbooks.

1. A large software system driven by business need undergoes continuing change or becomes progressively less useful (the '**Law of Continuing Change**').
2. As a software system evolves, it becomes more complex and requires work on improving the design structure (the '**Law of Increasing Complexity**').
3. Large systems tend to evolve independently of individual management decisions, constrained by organisational constraints such as time to respond to the need to change and structural factors such as the need to limit the size of change steps (the '**Law of Self-Regulation**').The rate of global activity in a large software project is statistically invariant and is not affected by changes to resourcing and staffing (the 'Law of Conservation of Organisational Stability').
4. The rate of adding new functionality to successive releases of a large program is statistically invariant and is constrained by the need to repair faults introduced by previous releases (the '**Law of Conservation of Familiarity**').

5. Continued user satisfaction depends on enhancement of functionality with each release (the '**Law of Continuing Growth**').
6. System quality will decline unless changes to the operational environment are addressed (the '**Law of Declining Quality**').
7. Finally, feedback is essential to improve the processes involved in evolution of the system in order to achieve significant improvement (the '**Law of Feedback Systems**').

Managing maintenance

The maintainability of a piece of software is determined by the ease with which it can be modified by someone other than the original developer to fix bugs or to address changing requirements and/or migrated to a different platform or release of system software.

Among the factors that affect maintainability are:

- modularity and encapsulation of functionality, improving coherence and reducing coupling (interdependencies) between modules and libraries
- quality and completeness of documentation including adherence to agreed standards for defining requirements and naming conventions
- re-use of software which allows maintenance to be shared between several 'stakeholder' teams (one example would be an open source community)
- avoidance of tools, languages and so on with a limited skills-base.

This last point cannot always be enforced. According to a 2006 report from the Australian National Audit Office (see: http://anao.gov.au/~/media/Uploads/Documents/2005%2006_audit_report_24.pdf), the Australian over-the-horizon radar system, a project begun in 1990, was written using FORTRAN, Pascal, C, C++, Coral 66 and various other languages, totalling about two million lines of source code. The difficulty in recruiting engineers with Coral 66 expertise in particular, a language developed in the 1960s, has led user organisations like the British Royal Air Force to develop their own training courses.

Re-engineering and refactoring

One specific aspect of improvement that is likely to accrue from the combination of object-oriented design and an agile approach is regular and frequent reappraisal of the way objects have been implemented. This is called refactoring and is aimed at improving maintainability.

Refactoring allows:

- elimination of duplicated code
- improvement of structure
- structuring of data
- removal of redundant code.

For older, legacy, systems the maintenance options are often limited to wrapping middleware around the system to allow integration with new systems or re-engineering. Re-engineering can be applied to translating or rewriting of source code (assuming that to be available) or reverse-engineering from the legacy system, improving program structure and the way data is handled.

Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- describe the key decisions that have to be taken when scheduling the next iteration of a system
- explain how process effectiveness can be measured and improved with explicit reference to GQM and CMMI.

Test your knowledge and understanding: Plan, Do, Check, Act methodology and Task Breakdown Structure

As you will have learned from investigation into PRINCE2, project managers identify and act upon lessons learned as a way of triggering process improvement. Plan, Do, Check, Act is one of the ways of managing this process. It is important that you demonstrate an understanding of the factors involved in process improvement.

1. Consider a scenario where one of the risks associated with requirements capture (discussed in Chapter 4 of the subject guide), frequently becomes an issue within the organisation you work for. Clearly, the underlying problem must be addressed.
2. Consider the way you would use the Plan, Do, Check, Act methodology to evaluate the source of the problem and introduce change to your company.
3. Describe your approach using a Task Breakdown Structure or flowchart.

Summary of the transition phase

- Software evolution is a natural consequence of requests for change to an earlier development. Lehman's laws provide insights into the nature of change in large systems where the cost of maintenance (bug fixing, porting and enhancing) normally exceeds development costs.
- Making software easier to maintain becomes more important as a system evolves. Reengineering the system to replace the old version or frequent small changes that improve maintainability but do not add functionality should be considered when the cost of maintenance is disproportional to the residual business value.
- Improvements to the three project manager concerns of quality, price and development time required can be achieved by the elimination of waste (inefficient processes, reworking) and the use of mature and more manageable engineering good practices.
- Targets for improvement should be exposed to a process of measure (time, resources and events); analyse and model (using a process modelling tool like the RUP); and finally, change. The CMMI process improvement model provides a framework for setting and achieving goals.

By now, you should be confident that you would be able to:

1. Validate that the objectives for this iteration of the system have been met and that all artefacts are complete.
2. Initiate the planning of another iteration of system development.
3. Apply lessons learned.

Part 5: Review and revision

Chapter 10: Case studies

Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- describe the contribution made by the 'big-bang' approach to the failure of the 1992 LASCAD
- describe the business benefits for Microsoft of adopting a formal iterative process model.

Essential reading

London Ambulance Survey case study

Finkelstein, A. 'Report of the inquiry into the London Ambulance Service (Feb. 1993)', international workshop on software specification and design case study; www.cs.ucl.ac.uk/staff/a.finkelstein/las/lascase0.9.pdf

Microsoft Case Study

Cusumano, M.A. and S. Stanley Smith 'Beyond the waterfall: software development at Microsoft', Working Paper #3844-BPS-95, 1995; http://dspace.mit.edu/bitstream/handle/1721.1/2593/SWP-3844-33836288.pdf

Further reading

Link to case studies: www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/index.html; www.safehomeassured.com

Overview

This chapter discusses two case-studies. Both are relatively old but highlight important lessons to be learned. The first investigates a catastrophic failure in deployment of a system. The second discusses the contribution made by adoption of process models to the evolution of a major software developer. In addition, we use our own case study of a journal publishing management system to demonstrate the use of the UML as a design tool (see Chapters 5 and 7 of the subject guide). Students also have access to the large-scale case studies of the Essential reading authors.

Please note other case studies may be added to the VLE, or will form the basis of coursework assignments.

London Ambulance Service case study

Reliability and security

Robust software is not just free of bugs (or relatively so), it does not just implement the stated requirements, it behaves predictably and is as Sommerville says '…without failures and available for use when it is required'. This was not the case with the London Ambulance Service Computer-Aided Despatch (LASCAD) system when it went live in 1992. As specified, the system raised on-screen 'exception reports' requiring operator intervention to address situations like delays in status reporting (which happened during shift changes); vehicles not being where they are supposed to be (because the crew took

a vehicle other than the one despatched by the system); and delays in responding to calls. However, the exception report appeared on the same screen as the queue of calls to be allocated, resulting in:

'…exception messages and awaiting attention queues scrolling off the top of the screen of the allocator ("exception rectifier") [or allocator's screen].' (See: www.cs.ucl.ac.uk/staff/a.finkelstein/las/lascase0.9.pdf)

Note that the exceptions were the result of human behaviour (as described above), not 'bugs' in the system. However, the cause of the problem was inadequate analysis of the requirement, which should have identified the risk of combining two separate processes. A great deal has been written in the literature on this particular case and it is worth searching out and reading some of the many articles.

Microsoft case study

Flexibility in software development

This case study, produced by the respected authors from the MIT Sloan School of Management and IBM, discusses in detail the approach taken by the Microsoft company to its software development – as the authors describe:

> '…(it was) labeled the "synch and stabilise" process. Since 1988–1989, the company has gradually been introducing techniques that do add structure to software product development but which also depart from the waterfall model. Many aspects of what Microsoft does resemble concurrent engineering and incremental development practices found in other software companies and in companies in other industries.' (See: http://dspace.mit.edu/bitstream/handle/1721.1/2593/SWP-3844-33836288.pdf)

The history of the company's use of software development techniques, stemming from the early waterfall models to the the 'synch and stabilise' process is analysed and much detail given about the problems and practicalities of software engineering in a large organisation. In a pertinent insight into the future (the article was written in 1995), the authors note:

> 'Microsoft groups need to understand, however, that complex new operating systems and network communications systems require more advanced planning and architectural design work than the company's desktop applications products.' (See: http://dspace.mit.edu/bitstream/handle/1721.1/2593/SWP-3844-33836288.pdf)

Other case studies

Both Essential reading textbooks, and many other sources also, have case study material and there is a wealth of material on the internet, much of which is useful and would make appropriate extended reading or viewing (many of these studies are now available in video formats).

A good starting point is the websites linked to the recommended textbooks, and to many other of the SE books referenced in previous chapters.

Sommerville's case studies

A selected set of Sommerville's list of case studies (which all have supporting material) is below, accessible via the link given under Essential reading at the start of this chapter and described here:

The case studies here are of two kinds. Namely, those introduced in the book (1–3), where we provide additional material about the systems, and other studies based on real systems that you can use as a source of examples.

The insulin pump system

See: www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/InsulinPump/index.html

- An embedded control system for a personal insulin pump, which is used by diabetics to mimic the function of the pancreas and hence control the level of glucose (sugar) in their blood.
- This study is used to discuss general issues of safety and safety-critical systems.

The mental health care patient management system

See: www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/MHCPMS/index.html

- A patient management system designed for use in clinics attended by patients suffering from mental health problems which records details of their consultations and conditions. It is a secondary safety-critical system as system failure can lead to decisions that compromise the safety of the patient or the medical staff caring for the patient.
- This study is used to discuss general issues around the design of information systems where the system dependability is important and security is a significant concern.

The wilderness weather system

See: www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/WildWeatherSys/index.html

- The software for a wilderness weather station that collects weather information in remote areas, part of a national weather information system.
- This study was developed to illustrate object-oriented development and also as a basis to discuss more general issues of systems engineering – the need for some kinds of system to be entirely self-contained.

The Ariane launcher accident

See: www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/Ariane5/index.html

- This case study describes the accident that occurred on the launch of the Ariane 5 rocket, which exploded shortly after take-off. The subsequent enquiry showed that this was due to a fault in the software in the inertial navigation system.
- This case study illustrates issues with requirements specification, multi-organisational working, critical systems validation and some of the problems of software reuse. It also shows the organisational complexity of systems development and how organisational issues can lead to systems failure.

Pressman's case study

Pressman and Lowe provide as a case study in the book *Web engineering: a practitioner's approach* full details of a fictional company, SafeHomeAssured.com, described below:

The intent of this website is to provide … an operational version of selected elements of the SafeHomeAssured.com Web application (WebApp) coupled with appropriate Web engineering models and other supplementary information that would help a Web engineering team to develop the WebApp. In addition, we provide links to a broad array of categorised Web engineering resources that may be of use to those who must build industry-grade WebApps.

The whole scope of Web Engineering design is covered in such links but of most use will be the following sections under the 'Web Engineering Resources' (WebE) section of the website:

- Project Planning and Problem Formulation
  www.safehomeassured.com/process.html

- Analysis Modelling
  www.safehomeassured.com/analysis.html
- Design
  www.safehomeassured.com/design.html

The case study itself commences here:
www.safehomeassured.com/casestudy/index.htm

Appendix 1: Bibliography

Ambler, S. *The unified process elaboration phase: best practices in implementing the UP*. (New York: CRC Press, 2000) [ISBN 9781929629053].

Beyer, H. *User-centred agile methods. (Synthesis lectures on human-centred informatics)*. (Morgan & Claypool Publishers, 2010) [ISBN 9781608453726].

Boehm, B.W. 'A spiral model of software development and enhancement', *IEEE Computer* 21(5) 1988, pp.61–72.

Booch, G., J. Rumbaugh and I. Jacobson *Unified modeling language user guide*. (New York: Addison Wesley Professional, 2005) second edition [ISBN 9780321267979].

Charette, R.N. *Software engineering risk analysis and management*. (New York: McGraw Hill Intertext, 1989) [ISBN 9780070106611].

Cockburn, A. *Agile software development: the cooperative game*. (Boston: Addison Wesley Professional, 2006) second edition [ISBN 9780321482754].

Fenton, N. and S.L. Phleeger *Software metrics: a rigorous and practical approach*. (PWS Publishing, 1998) second edition [ISBN 9780534954253].

Gamma, E., R. Helm, R. Johnson and J. Vlissides *Design patterns: elements of reusable object-oriented software*. (London: Addison Wesley, 1994) [ISBN 9780201633610].

Highsmith, J. *Agile project management: creating innovative products*. (Boston: Addison Wesley Professional, 2010) second edition [ISBN 9780321658395]; www.pearsonhighered.com/educator/product/Agile-Project-Management-Creating-Innovative-Products/9780321658395.page

Jacques, D. and G. Salmon *Learning in groups: a handbook for face-to-face and online environments*. (London: Routledge, 2006) fourth edition [ISBN 9780415365260].

Kan, S.H. *Metrics and models in software quality engineering*. (London: Addison Wesley, 2002) [ISBN 9780201729153].

Kruchten, P. *The rational unified process – an introduction*. (Boston: Addison Wesley Professional, 2004) third edition [ISBN 9780321197702]; www.pearsonhighered.com/educator/product/Rational-Unified-Process-The-An-Introduction/9780321197702.page

Lethbridge, T. and R. Laganière *Object oriented software engineering: practical software development using UML and Java*. (London: McGraw Hill, 2005) [ISBN 9780077109080].

Linger, R. 'Cleanroom process model', *IEEE Software* 11(2) 1994, pp.50–58.

Lund, A. *User experience management: essential skills for leading effective UX teams*. (Morgan Kauffmann, 2011) [ISBN 9780123854964].

Myers, G. *The art of software testing*. (Oxford: Wiley, 2011) third edition [ISBN 9781118031964].

PMI, A guide to the project management body of knowledge. (PMBOK® guide) (Newtown Square, PA: Project Management Institute, 2013) fifth edition [ISBN 9781935589679]; www.pmi.org/en/PMBOK-Guide-and-Standards/Standards-Library-of-PMI-Global-Standards.aspx

Pressman, R.S. *Software engineering: a practitioner's approach*. (New York: McGraw Hill, 2010) seventh edition [ISBN 9780071267823 (pbk); 9780073375977 (hbk)]; www.rspa.com/sepa7e.html; http://highered.mcgraw-hill.com/sites/0073375977/information_center_view0; www.mhprofessional.com/product.php?isbn=0073523291

Rosenberg, D. and M. Stevens *Use case driven object modeling with UML*. (London: Academic Press, 2013) second edition [ISBN 9781430243052]. See: www.iconixsw.com/Books.html

Royce, W.W. 'Managing the development of large software systems: concepts and techniques', *Proceedings of IEEE WESTCON* (Los Angeles, 1970), pp.1–9; http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf

Scacchi, W. 'Process models in software engineering', in J.J. Marciniak (ed.) *Encyclopedia of software engineering*. Volume 2 (New York: John Wiley and Sons, Inc., 2001) second edition [ISBN 9780471210078].

Sommerville, I. *Software engineering*. (Boston: Addison Wesley, 2010) ninth edition [ISBN 9780137035151]; www.cs.st-andrews.ac.uk/~ifs/Books/SE9/; http://catalogue.pearsoned.co.uk/catalog/academic/product/0,1144,0137053460-NTE,00.html

Appendix 2: Revision support

Collected together below in Appendix 2 are what you should have gained from each main chapter, in terms of your abilities to carry out these tasks and activities, and to understand the processes underlying them in terms of the four main phases.

This is followed in Appendix 3 by the collected set of all cornerstone points made in the subject guide to aid revision. Appendix 4 contains 'Revision guidance notes', while Appendix 5 contains a Sample examination paper. Finally, Appendix 6 contains an outline marking scheme, but no sample answers.

Summary of main processes in the four phases

Inception phase	Scoping and justifying a project.
Elaboration phase	Defining a response to stakeholder needs.
Construction phase	Managing the provision of the functionality agreed.
Transition phase	Producing a product release.

1. Inception phase

- Define the scope of a project in terms of core requirements, key features and the main constraints and risks.
- Produce a plan describing the development process that you would follow, supported by estimates of cost and schedule in line with the client's business case.

Software processes

- Understand the concept of a process model as a way of describing a software development project and differentiate between common approaches such as waterfall, iterative and agile.
- Identify the key artefacts produced during an iteration of the software development process.
- Understand the role of the project manager in delivering those artefacts.
- Understand the ways in which the development process can affect quality.

Requirements engineering

- Describe the differences between elicitation, specification, validation and documentation of requirements.
- Understand the differences between system and user requirements and between functional and non-functional requirements.
- Use natural language and graphical representations of requirements within a requirements document.
- Produce a system model based on requirements.

Planning, cost and schedule estimation

- Describe the structure of a project schedule.
- Apply resources to execute a project plan.
- Understand the constraints on the accuracy of resourcing and budgeting.

2. Elaboration phase

- Validate that the current project plan and architecture design are complete and able to deliver the project vision and requirements.
- Demonstrate that major risks have been identified and that each is being managed appropriately.

Risk management

- Describe the steps required in order to understand how a risk should be mitigated, monitored and managed.
- Give examples of risks created by the use of technical and human resources, the way the client and developer organisations work and performance-related risks.
- Identify probable causes of risks in those five areas and ways in which pro-active risk management can be used to reduce the probability and/or impact.

Architecture, modelling and design

- Use UML graphical models to represent the context (external environment) in which the system will operate, the interactions between the system and that environment, the static and dynamic structures of the system and its responses to stimuli.
- Make informed decisions during the architectural design process.
- Base a system architecture design on a generic architectural pattern where appropriate.

Managing the execution of the project plan

- Describe the main functions of a project manager.
- Understand the dynamics of a development team.
- Identify the tools and infrastructure required for the development environment.
- Describe the roles and responsibilities for configuration management.

3. Construction phase

- Validate that the system being developed is fit for purpose.
- Be able to deploy it for beta-testing with the target user community.
- Provide those users with the necessary documentation and support, including training.

Detailed design

- Apply object-oriented principles in the detailed design phase.
- Describe the use of UML documentation for decomposing the architectural design into usable object-oriented software.
- Explain the purpose of design patterns and give examples of design patterns for data-processing, process-control and e-commerce applications.

Quality management

- Define quality as it applies to software engineering and identify the key quality-related activities that take place at each phase of the project lifecycle.
- Explain the advantages and limitations of qualitative and quantitative quality measures and give examples of each.

- Explain the difference between verification and validation and the purpose of testing at the sub-function, component, system and user levels.

4. Transition phase

- Validate that the objectives for this iteration of the system have been met and that all artefacts are complete.
- Initiate the planning of another iteration of system development.
- Apply lessons learned.

Evolution

- Describe the key decisions that have to be taken when scheduling the next iteration of a system.
- Explain how process effectiveness can be measured and improved with explicit reference to GQM and CMMI.

Notes

Appendix 3: Cornerstones

CORNERSTONE 1

Project management is a creative activity. It requires a combination of knowledge, insights and skills in order to deliver the required outcomes for each unique project. If you do not take into account the specific challenges and rely instead on a single standard approach, you are not a project manager, you are a project administrator.

CORNERSTONE 2

Project management is about controlling the strengths and weaknesses of the project team (and enterprise) and the opportunities and threats that can arise and lead to issues for the project, the 'internal' and 'external' factors of a SWOT analysis.

CORNERSTONE 3

A process model is a high-level and incomplete description of the way a project is controlled and how it evolves. It introduces interfaces between actors and workflows and transitions between activities. Your choice of model does not fundamentally affect WHAT you are going to do; customers still need to know that their requirements are the reference point for design and code still needs to be tested before and after integration. What the model allows you and your team to do is focus on the important transitions during the project.

CORNERSTONE 4

The purpose of a software project is to meet a client's need by delivering a software system. That system decomposes into sub-systems and components, and each component requires a number of products to be built, tested and integrated. There are two approaches that a project manager can adopt – focus on the product, or focus on the work required to deliver that product.

CORNERSTONE 5

A best practice in common use is the breakdown structure. A Work Breakdown Structure, for example, decomposes the project into phases, decomposes phases into activities and activities into tasks and sub-tasks. You can describe dependencies from one to another through a Work Flow Diagram (e.g. a GANTT chart). Similarly, you can represent the relationships between artefacts from a project in a Product Flow Diagram, which then defines the order in which they must be completed. You can also use breakdown structures to ensure that an analysis of factors such as risks or competences is comprehensive.

CORNERSTONE 6

A specification document must comply with the 'four C's':

Clear: No ambiguities or confusing cross references to other documents.

Concise: No unnecessary detail, repetition or confusing cross-references to other documents.

Correct: Attributed, with no conjecture, paraphrasing or unsubstantiated opinion.

Complete: Verified as part of the V & V process.

CORNERSTONE 7

The project team invariably focuses on the outputs it is to create, while the client is concerned with the outcomes of using those outputs. 'Outcome-based specifications' address the client's needs and do not prescribe **how** they are to be met.

CORNERSTONE 8

Functional requirements describe the way the system behaves.

Non-functional requirements describe the context within which that system behaviour is delivered.

User requirements are the functional and non-functional requirements that describe the business goals for the system.

System requirements are the functional and non-functional requirements that have to be met in order to achieve the business goals for the system.

CORNERSTONE 9

ISO/IEC 12207 defines validation as confirmation that the final, as-built, system fulfils its specific intended use, that it is 'fit for purpose'. Verification confirms that the output of each activity (such as requirements capture, design, code, integration or documentation) fulfils the requirements or conditions imposed by preceding activities.

This is often expressed in terms of validating that the 'right thing' is being built and verifying that the thing is being 'built right'.

CORNERSTONE 10

The outcome of a risk analysis should reflect the status of the project AFTER the mitigation has been brought into the project plan, in other words the risk register reflects the residual risk only. This implies that risk assessment is an iterated or continuous process in order to re-evaluate mitigated risks.

CORNERSTONE 11

When deciding whether to use the UML during architectural design, you need to be able to answer yes to the following five questions.

1. Do we have good tools for doing things this way?
2. Do we understand how to use them?
3. Do we know what we are trying to achieve?
4. Will we document the decisions taken clearly?
5. Will the client understand the result?

CORNERSTONE 12

Formal documents are critical. They tend to be kept (if not actually read) and they tend to change over time. Do not simply flag the changes by giving a document a different date in its filename or by changing the name from V0.2 to V0.3. Give formal documents a history section and keep them (without ever changing the filename) in a document archive such as Sharepoint. The LEAN philosophy acts as a good guide: if you are not going to keep it, don't bother updating it. Conversely, if you do not maintain it, there is little point in keeping it!

CORNERSTONE 13

'A team is not a bunch of people with job titles, but a congregation of individuals, each of whom has a role which is understood by other members. Members of a team seek out certain roles and they perform most effectively in the ones that are most natural to them.' (Dr R.M. Belbin)

Notes

Appendix 4: Revision guidance notes

Revision technique

The following activities are recommended as part of your revision plan:

1. Review the subject guide and the directed reading given.
2. Write down – in your own words – the definitions and concepts introduced.
3. Find a good example to support or help explain each definition or concept.
4. Review any examples given in the subject guide and from recommended reading.
5. Try to integrate the key concepts you have learned with new content (from the textbooks, lectures, class discussions and exercises, and your own study).
6. Study and learn the content, key definitions and basic definitions.
7. Make good use of the additional links and further information provided to help reinforce key content and introduce new perspectives.

Examination technique

To achieve a good grade, you need to prepare for the examination and be able to solve problems by applying the knowledge you have gained from the course. A good approach is one of: **read**, **think**, **plan**.

READ questions carefully

1. Choose your questions.
2. Make sure you fully understand what is required, and have identified all subsections in a question.
3. Judge the relative effort and time.
4. Work out a time allocation for each part of the question.
5. Read the question carefully once more.

ANALYSE the questions and THINK about what you are going to do

1. Is it a **knowledge** or a **definition** question?

   This type of question requires knowing definitions in the subject guide, or the recommended reading. Questions are likely to begin with:

   a. What is…?
   b. Explain what…
   c. Describe…
   d. Give a definition of…
   e. Identify and name…etc.

   You will have to demonstrate, at the very least, basic knowledge of key terms.

2. Is it an **analysis** question?

   This type of question requires a good understanding of important concepts so that they can be tested with reference to what you have learnt and how you apply that knowledge. Questions are likely to begin with:

   a. Discuss…
   b. Explain why…
   c. Imagine…
   d. Develop…, etc.

As well as the the definitions, key terms and concepts that you have learnt you will have to demonstrate reasoned, step-by-step arguments to:

i. Make a point.
ii. Explain why that point is important.
iii. Explain the significance.
iv. Give supporting examples.

3. Is it an **evaluation** question?

   This type of question requires you to weigh-up and assess a set of issues and then to demonstrate the conclusion you have reached, and why, or to make some appropriate recommendation or judgement. Such questions may consist of instructions to complete certain tasks and are likely to begin with:

   a. Write a…
   b. Draw a diagram…
   c. Analyse…
   d. Demonstrate the pros and cons of…
   e. Draw up a table showing…, etc.

   You will have to assess good and bad points, advantages and disadvantages, pros and cons, and write well about how you came to those conclusions. You will have to explain and describe any supporting evidence, showing that you understand it.

PLAN carefully what you will do

1. Identify the type of question.
2. Draft a plan, outline or spider diagram.
3. Plan a more detailed outline of your answer.
4. Check that your answer is specific to the question.
5. When you know what you want to say, write your answer.
6. Check what you have done and read over what you have written.
7. Go on to the next question.

Appendix 5: Sample examination paper

Duration: 2 hours 15 minutes

There are FIVE questions in this paper. Candidates should answer THREE questions. All questions carry equal marks and full marks can be obtained for complete answers to THREE questions. The marks for each part of a question are indicated at the end of the part in [.] brackets.

Only your first THREE answers, in the order that they appear in your answer book, will be marked.

There are 75 marks available on this paper. Each question is worth 25 marks.

Question 1

Project lifecycle

a. The RUP is described as a 'goal-driven' methodology. Name and describe the goals of EACH of the four phases of the RUP. [12]

b. Explain ALL of the following terms used in the RUP: "iteration", "milestone", "artefact". [6]

c. Draw up a table, comparing the potential benefits and possible drawbacks of the RUP approach against ONE other commonly used process model. [7]

Question 2

Requirements engineering

a. Explain why EACH of the following requirements are gathered: business, operational, functional, non-functional. [8]

b. Explain the reason for adopting FURPS or FURPS+ as part of your requirements-gathering methodology. [8]

c. List the essential elements of the customer's business case, needed before decision to proceed can be taken. [9]

Question 3

Change management

a. Identify the TWO main types of change to a system, and explain the typical causes of 'requests for change'. [5]

b. Describe how CMMI Level 3 best practices can allow a Project Manager to reduce the need for change. [12]

c. LEAN thinking is about eliminating waste, by avoiding the need for reworking. Discuss how the agile philosophy of "envision, explore and deploy" contributes to change management. [8]

Question 4

Risk management

a. Describe the concepts underlying the RMMM approach to risk. [3]

b. Explain the differences between BOTH pairs of terms:

   i. "risk" & "issue"

   ii. "mitigation" & "contingency". [4]

c. Produce a table (like that illustrated below) populated with examples of EACH of the six categories of risk proposed by Sommerville. [6]

Risk	Probability	Impact	Response

d. For EACH of the examples you select,

   i. Provide an initial risk-assessment (i.e., low, medium or high probability; and low, medium or high impact).

   ii. Provide the most appropriate response. [6]

e. Describe how BOTH risks and issues are managed, in terms of reporting and planning. [6]

Question 5

Design practices

a. Explain the contents of a use-case and its role in requirements gathering, architectural design and testing. [8]

b. Show in a diagram the purpose of a design model as an abstract representation of the proposed system and its relationship to the concrete views provided by the implementation model and the deployment model. [9]

c. Describe how the UML can be used to represent the logical, process, development and deployment views of the system. [8]

END OF PAPER

Notes

Appendix 6: Outline marking schema for Sample examination paper

Question 1: Project lifecycle

a. (12) – 3 marks for each RUP goal description (inception, elaboration, construction, transition).

b. (6) – 2 marks for each correct definition of terms.

c. (7) – 3 marks for description of the 'other' process model; 4 marks for quality of comparison. Must be in table format.

Question 2: Requirements engineering

a. (8) – 2 marks for each correct explanation of a requirement type.

b. (8) – 2 marks each for identifying and describing at least four of the following: functionality, usability, reliability, performance and supportability.

c. (9) – 3 marks each for benefits, costs and risks correctly identified.

Question 3: Change management

a. (5) – 1 mark each for correctly defining the following: 'request for change' (requirement creep), 'off-spec' (bug), change to business requirements, trade-off in non-functional requirements and misunderstanding.

b. (12) – 2 marks each for sufficiently describing the following: monitoring, planning, measuring, requirements-management, risk management and supplier management.

c. (8) – 2 marks each for explanation of agile manifesto values: (namely, people over process, software over documentation, collaboration over contract and responding to change over planning).

Question 4: Risk management

a. (3) – 1 mark each for correctly describing the RMMM terms: mitigation, monitoring and management.

b. (4) – 2 marks each for differentiating between actual issues and potential issues (risks) and between reducing/preventing risks and reducing impact of issues.

c. (6) – 1 mark each for valid examples of risks: technology, people, organisational, development environment, requirements and estimation.

d. (6) – 1 mark each for examples with appropriate response to H/M + H/M (avoid), H/M + L (mitigate), L + H/M (contingency), L/L (monitor).

e. (6) – 3 marks for description of exception-based reporting (checkpoints, etc.); 3 marks for explanation of response (exception plan).

Question 5: Design practices

a. (8) – 2 marks for description of a use case; 2 marks each for the three roles (functional requirements, interfaces to external systems, development of test plan).

b. (9) – 3 marks for explaining that the design model translates the logic of the use case into a description of system processes; 3 marks for explaining that the implementation model organises the design into buildable components; 3 marks for explaining that the deployment model describes the physical organisation of those components.

c. (8) – 2 marks each for presentation of class, sequence, component and deployment diagrams.

Notes