Comparative Software Engineering: Review and Perspectives – Guest Editors' Introduction

Yingxu Wang* and Dilip Patel**

* Centre for Software Engineering, IVF
Argongatan 30, S-431 53, Molndal, Gothenburg, Sweden
Tel: +46 31 706 6174, Fax: +46 31 27 6130
Yingxu.Wang@acm.org

** School of Computing, IS, and Mathematics
South Bank University
103 Borough Road, London SE1 0AA,UK
dilip@sbu.ac.uk

Abstract: Engineering is a set of disciplines seeking solutions for complicated problems and systems that could not be done by individuals. The aim of engineering is to repetitively produce complicated artefacts in an efficient way. This paper describes a set of generic engineering principles and an engineering maturity model. With the engineering principles and model, the nature and status of software engineering are analysed. Interesting findings on what software engineering can learn from generic engineering principles are presented. This paper intends to show the nature, status and problems of software engineering, as well as its future trends, based on the comparative studies between the generic engineering principles and software engineering practices.

Keywords: Engineering, software engineering, engineering principles, engineering maturity model, nature of software engineering, cross fertilisation

1. "To Be or Not To Be?"

To many professionals engineering means systematic planing, teamwork, rigorous process, repeatability, efficiency. Software professionals have been arguing the term "software engineering" and its implication for three decades since Frits Bauer invented it in 1968 [1-2]. Yet, still some fundamental questions remain, such as:

a. Is software development an engineering discipline?

b. Are software developers engineers or craftsmen?

There were completely different assertions and opinions on the above issues of "to be or not to be" that is still confusing the academics, practitioners, and students [3 - 7] in software engineering and in the software industry.

In investigating these fundamental problems, the authors find the myth was caused by a confusion of time in perceiving software development as, or as not, an engineering discipline. The authors' answer to the question whether software development is an engineering discipline is simply 'no.' While more precisely: it is 'not' at present and in the past, and it is going to be and should be 'yes' in the future. Currently, software development is evolving from the laboratory-oriented and all-round-programmer-based practice to an industry-oriented and process-based platform, and software developers are experiencing changes of roles from craftsmen to regulated professionals – the software engineers. The practices of the former are based on personal talents, tastes and art, while those of the latter are based on disciplined processes and repeatable professional activities.

2. What is the Nature of Software Engineering?

Conventional industries are producing artefacts from raw materials via engineering approaches; Software industry is producing software solutions for problems via software engineering. In 1975 Hoare identified four characteristics of software engineering [8] – professionalism, vigilance, sound theoretical knowledge, and tools. These characteristics were quite generic for perceiving software engineering at that time. In a further development in 1996, Wasserman [9] described eight technical characteristics of software engineering as follows:

- Abstraction
- Methods and notations
- Prototyping
- Modularity and architecture
- Lifecycle and process
- Reuse
- Metrics
- Tools and integrated environments

Wasserman's vision mainly covered the technical characteristics of software engineering. Therefore Hoare's view has been seen as a balance to show both the sights of the forest and trees in describing the young discipline of software engineering.

To investigate the nature of software engineering in a systematic way, the authors find there are five categories of characteristics of generic engineering principles, which software engineering can borrow. They are the categories of engineering aim, organisational, technical, managerial, and social characteristics [6]. According to the classification, a key to software engineering is the category of engineering organisation, which characterises the following features:

- Apply systematic processes
- Support co-operative work
- Adopt division of work
- Establish standardisation
- Adopt tools and machinery
- Plan actual schedule
- Optimise resources allocation
- Derive predictable outputs
- Seek controllable quality

These characteristics determine the level of maturity in engineering organisation. All of them are applicable towards maturing the software engineering discipline. To further explore the nature of software engineering, the authors found it is useful to contrast the three-generation definitions of software engineering.

The first definition of software engineering was provided by Bauer [1] more than three decades ago. In his paper Bauer defined software engineering as: "*The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*"

One of the well-accepted second generation definitions of software engineering, by McDermid [3], is as follows: "*Software engineering is the science and art of specifying, designing, implementing and evolving - with economy, timeliness and elegance – programs, documentation and operating procedures whereby computers can be made useful to man.*"

Along with the evolution of software engineering research and practices, the authors find it is useful to shift the concept of software engineering from conventional laboratory orientation to an industrial orientation. This results in a new perception on software engineering [10, 6] as follows: "*Software engineering is a discipline that adopts engineering approaches, such as established methodologies,* *processes, tools, standards, organisation methods, management methods, quality assurance systems, and the like, in the development of large-scale software seeking to result in high productivity, low cost, controllable quality, and measurable development schedule*.”

Contrasting the differences between the three definitions of software engineering leads to the distinctions noted in Table 1. Table 1 shows how the concepts surrounding software engineering can be enhanced, particularly in terms of its means, aims and attributes; resulting in a deeper understanding of software engineering.

Table 1. Analysis of Representative Definitions of Software Engineering

No.	Nature	Means	Aims	Attributes of aims
1	A method	Engineering principles	Software	- economy - reliability - efficiency
2	A science and art	Life cycle methods: - specification - design - implementation - evolving	Programme and Document	- economy - timeliness - elegance
3	An engineering discipline	Engineering approaches: - standards - methodologies - tools - processes - organisational methods - management methods - quality assurance systems	Large-scale software	- productivity - quality - cost - time

Some points regarding the perceived nature, means, aims and their attributes in the answers can be made. The first definition proposed software engineering as a method or approach to software development; the second definition focused on scientific methods and art for programming; and the third definition portrays software engineering as an engineering discipline for developing large-scale software in the software industry.

The above discussion shows that there is a need to shift from the conventional laboratory-oriented and all-round-programmer-based software engineering perception to a modern industry-oriented and process-based software engineering platform. This is going to be a significant trend in the organisation and implementation of software engineering in the software industry.

3. What are the Principles of the Generic Engineering Approach?

Engineering approaches and engineering disciplines emerged during the 19th century industrial revolutions. Before the industrial revolution, people produced goods as craftsmen at small or limited scales and they learnt things by doing.

For improving productivity as well as quality, and for lowering the requirements for skills in mass production, a generic industrial engineering approach [11, 12] had been formed as outlined in the following phases:

a. To identify repeatable work processes

b. To identify standard and reusable components of products

c. To adopt division of work (people specialised in a defined role in processes)

d. To equip specialised tools for the roles and processes

e. To recognise management as a profession for organisation of the processes and for
co-ordination of the roles

These key steps of a generic engineering approach form the basis of almost all the existing engineering disciplines. Historically, every engineering discipline in the modern industries has been developed and matured in the same approach: first a kind of art, then a discipline of engineering.

For instance, in the early 19th century and even earlier, watches were produced manually, and so there were no identical watches. At this stage, the watchmakers were characterised as craftsmen rather than engineers. This resulted in low productivity and high price, and one would perhaps need to find the original watchmaker in order to have a watch fixed. In the middle and late 19th century, the industrial revolutions addressed some of the problems and introduced the approach to engineering. Taking the watch manufacturing industry, as an example again, it was a significant achievement when the industry could massy produce watches by machines, and all watches were identical so that parts were interchangeable among watches of the same brand. At this stage, the traditional watch-smiths had become engineers, who were responsible and skilled for one or limited mass production process of watch manufacturing.

This generic industry approach to engineering is also applicable to software engineering, although it has often been ignored in research and practice. Towards establishing an engineering discipline, software engineering is going to repeat the history of industrialisation as we reviewed in this section. The findings are also useful to support the answers in Section 1 for the fundamental questions about software engineering and software engineers.

4. How the Other Engineering Discipline Matured?

By comparing the differences between an engineer and a craftsman in the time dimension as discussed above, we may elicit a generic engineering maturity mode from the history of industrialisation. With the mode of engineering maturity, we are able to examine the status and maturity level of software engineering as an engineering discipline, and to predict its future development.

Looking at the time dimension, engineering is a discipline matured from arts of craftsmen in terms of scale and rigour. While in the professional dimension, engineering is a discipline parallel to sciences, in which engineers are working on technology development and mass production by applying scientific principles and inventions. When asking how the industrial revolutions had changed the traditional individual or family watchmakers, history tells us that the manual watchmakers had disappeared except a few that existed as a special profession. We may also find similar evolution traces in other traditional manufacturing engineering disciplines. This indicates a universal engineering maturity model (EMM) in the industries as shown in Table 2.

In the EMM model for industry disciplines, there are four levels of engineering maturity such as the *emerging, art, engineering,* and *post-engineering* ages. The key characteristics of each level in EMM have been identified as shown in Table 1. Applying the EMM model, we may find some examples of existing engineering disciplines that are already at level 4 – the post-engineering age, such as civil engineering, mechanical engineering, and electrical engineering. Electronic engineering would be at level 3 – the engineering age, since it is still under rapid development within the context of a wide range technical innovation. Biological engineering is an example of those at level 1 – the emerging age of an engineering discipline.

Based on the EMM model, we can predict that software engineering as a young engineering discipline, is going to be matured in the same way: from art to engineering. Checking with the engineering maturity characteristics, software engineering is found to be a good example of a level-2 discipline in the art age, while it is under a transition toward level 3 – the engineering age.

Table 2. The Engineering Maturity Model (EMM)

Maturity level	Description	Characteristics
1	The emerging age	- Being a branch of an existing discipline - Demands in sciences and/or industry have been identified - Common theories and foundations have been formed - A group of professionals has been emerged
2	The art age	- Varying professional practices - Individual stamps and influences both design and implementation - All processes are dependent on personal talent, art and hobby - Work is experience-based and doing by learning - Individual tends to be wizard for everything in all processes - Chasing new methods and/or technologies before their validation has been proven
3	The engineering age	- Adoption of work division - Established processes - Reinforced standards - Stable professional practices - Defined best practices - Well developed theories and foundations - Proven methods and technologies
4	The post-engineering age	- Well defined processes - Well defined standards - Precisely defined professional roles within a discipline - Refined theories and foundations - Refined methods and technologies - Giving birth of new disciplines

5. What is the Status of Software Engineering as an Engineering Discipline?

Analysing the status of software engineering against the EMM model as a reference system, it can be found that software engineering is actually a young discipline, which is located in the art age and is in a transition to the engineering age. Though, there is still some way to go for software engineering to be a matured engineering discipline.

A detailed analysis of characteristics of software engineering at different maturity levels is provided in Table 3. From Table 3 we can see that current software development practices and software engineering education are still located in the age of art and are progressing towards the age of engineering, because the centre of education and practices is mainly craftsman-and-laboratory-environment-oriented. For software engineering to evolve into a mature engineering age there is still much to do as described in the last column of Table 3. Although, historically, it is encouraging to see that software engineering, as a new engineering discipline, has matured to between levels 2 and 3 in just three decades; the other existing engineering disciplines, e.g. civil engineering and etc., have taken hundreds of years to reach their current levels of maturity.

Table 3. Characteristics of Software Engineering in Different Ages

No.	Characteristics of SE in the art age	Characteristics of SE in the engineering age
1	Individual perception on software development activities.	Team perception on software development activities.
2	A programmer, as software developer, is a master of all skills needed for programming.	A software engineer skills for a single or limited development process(es).
3	Final products reflect personal talent, art, hobby and experience.	Final products are based on sound theory, proven methodologies and common practices.
4	A software developer is a person who has multi-roles as of requirement analyst, designer, programmer, tester, and even user.	A software engineer is a person who has specific role in one of the processes as listed on the left.
5	Software product is a personal solution to an application.	Software product is a standard and regulated solution to an application.
6	Programming is personal interaction between the programmer and a computer.	Programming is a group interaction between all roles and processes, including the user(s).
7	Program is written for machines rather than for human being.	Program is written for colleagues involved in all processes rather than only for machines.
8	Programmers are not trained in formal ways, but believe learning by doing.	Software engineers are trained in formal ways and following common rules.
9	Knowledge transfer is hard in programming, and design and implementation of software is regarded as personal experience.	Knowledge transfer is defined and carried out by hierarchical processes at organisation, project, and individual levels.
10	Problems to be solved are limited in small scale.	Problems to be solved are large-scale software development for complicated systems.
11	Program maintenance is relied on the original designer.	Software maintenance can be carried out independently from the original developers.
12	An individual virtually runs a program using one's mental power for software validation.	Software validation is carried out by rigorous architecture design, testing and logical deduction.
13	A programmer is self-sufficient and self-managed for all processes.	Software engineers are mutually related with a defined as well as the pre- and post-processes.
14	Local available of materials, tools, and solutions.	Global available of materials, components, tools, and solutions.

6. What Can We Learn from the Generic Engineering Principles?

The most important principle of generic engineering organisation is 'division of work', or limitation of the roles of a software engineer in the whole software development processes. Considering that in electronic engineering, an electronic engineer is not supposed to be specialised in all application areas of electric engineering: from low to high frequency circuits, from analogue to digital circuits, from real-time systems to home appliances. Similarly, a car engineer is not supposed to be experienced in all areas of car manufacturing and maintenance, such as mechanical structures, engines, transmissions, electronic systems, micro-controllers, petrel, lighting, safety facilities, and bodies of vehicles.

Therefore, 'division of work' is the key for organising an engineering discipline, which is so obvious and so often to be ignored in current software engineering practices. For large-scale software development, what we need is highly skilled software engineers who are competitive for one or limited roles, rather than a person with all-round skills in the software engineering processes. This is what we learnt from the universal principles of industry engineering.

The second fundamental issue we learnt is that, obviously, there are significant gaps in many important aspects of software engineering, such as:

- team perception
- programming for colleagues in related processes rather than for machines
- following common roles rather than personal hobby
- roles and best practices are explicitly described and regulated by processes rather than remaining as personal experience and private knowledge

- test and maintenance are carried out independently from original developers
- software engineers are prepared to fit in specific process rather than tend to be a master of all-round activities in development
- maximum reuse of available components and tools rather than tends to be self-sufficient.

The EMM model can be taken as a reference for analysing and organising software engineering for a maturing engineering discipline. By clarifying the current status of software engineering as a discipline in the age of art, responsibilities of software engineering researchers and practitioners are to push forward software engineering to a matured engineering discipline by applications of the generic engineering principles we gained from other engineering disciplines.

Note from Ruzanna Chitchyan: section 7 is not relevant, it is for specific articles that we are not reading.

~~7. What are Presented in this Volume on Comparative Software Engineering?~~

~~In the above sections, we presented the EMM model and contrasted characteristics of software engineering in the engineering and art ages. With these results as a framework, we present two sets of articles on "Pinnacles of Software Engineering Technologies" and "Approaches to Software Engineering" in this special volume of Annals of Software Engineering. As shown in Table 4, the papers are selected from contributions of 14 outstanding researchers and practitioners in software engineering among 34 submissions.~~

~~Table 4. The Structure of ASE Vol.10 on Comparative Software Engineering~~

~~No.~~	~~Pinnacles of Software Engineering Technologies~~	~~Approaches to Software Engineering~~
~~1~~	~~*D. Bjorner*~~ ~~Pinnacles of Software Engineering: 25 Years of Formal Methods~~	~~*W. Humphrey*~~ ~~Software – A Performing Science?~~
~~2~~	~~*F. Barbier and B. Henderson-Sellers*~~ ~~Object Modelling Languages: An Evaluation and Some Key Expectations for the Future~~	~~*A. Bryant*~~ ~~Metaphor, Myth and Mimicry: The Bases of Software Engineering~~
~~3~~	~~*A. Mili et al.*~~ ~~A Comparative Analysis of Hardware and Software Fault Tolerance: Impact on Software Reliability Engineering~~	~~*B. Beizer*~~ ~~Software is Different~~
~~4~~	~~*C. Rolland*~~ ~~From Conceptual Modelling to Requirement Engineering: an Engineering Approach~~	~~*C. Lewerentz and H. Rust*~~ ~~Are Software Engineers True Engineers?~~
~~5~~	~~*B. Boehm et al.*~~ ~~Software Development Cost Estimation Approaches – A Survey~~	~~*D.J. Ram et al.*~~ ~~An Approach for Pattern Oriented Software Development Based on a Design Handbook~~
~~6~~	~~*J. Tsai and K. Xu*~~ ~~A Comparative Study of Formal Analysis Techniques for Software Architecture Specifications~~	~~*G.A. King*~~ ~~Quality Technique Transfer: Manufacturing and Software~~
~~7~~	~~*D.C. Rine and N. Nada*~~ ~~Three Empirical Studies of a Software Reuse Reference Model~~	~~*B. Banerjee*~~ ~~Mapping Software: Are We Nearing Standardisation?~~

~~We hope that readers of Annals of Software Engineering will benefit from the papers presented in this Special Volume on Comparative Software Engineering and be able to share the cutting-edge research results with the authors. We would like to thank the reviewers for their enormous effort in helping to improve the quality of this special volume that covered a wide range of fundamentally significant issues in software engineering.~~

References

[1] Naur, P. and Randell, B. (eds.) (1969), *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO.

[2] Bauer, F. L. (1976), Software Engineering, in Ralston, A. and Meek, C. L. (eds.)*, Encyclopedia of Computer Science,* Petrocelli/Charter.

[3] McDermid, J. A., ed. (1991), *Software Engineer's Reference Book*, Butterworth-Heinemann Ltd., Oxford.

[5] Sommerville, I. (1996), *Software Engineering* (5th edition), Addison-Wesley, 1996.

[6] Wang, Y. and Graham, K. (2000), *Software Engineering Processes: Principles and Applications*, CRC Press, USA, ISBN: 0-8493-2366-5, pp. 1-746.

[7] Pressman, R. S. [1992], *Software Engineering: A Practitioner's Approach* (3rd ed.), McGraw-Hill International Editions, 1992.

[8] Hoare, C.A.R.(1975), Software Engineering, *Computer Bulletin*, Dec., pp.6-7.

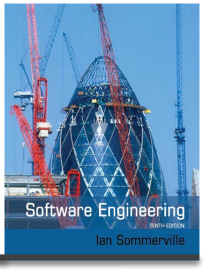
[9] Wasserman, A. (1996), Toward a Discipline of Software Engineering*, IEEE Software,* Nov., pp.23-31.

[10] Wang Y., Bryant A. and Wickberg, H. [1998], A Perspective on Education of the Foundations of Software Engineering, Proceedings of the 1st International Software Engineering Education Symposium (SEES'98), Scientific Publishers OWN, Poznan, pp.194-204.

[11] Marshall, A. (1938), *Principles of Economics*, The Macmillan Co., London.

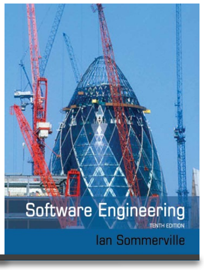
[12] Kuhn, T. (1970), *The Structure of Scientific Revolutions,* The Univ. of Chicago, Chicago.

Frequently asked questions about software engineering



Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

Frequently asked questions about software engineering



Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.
What differences has the web made to software engineering?	The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

2

Software processes

Objectives

The objective of this chapter is to introduce you to the idea of a software process—a coherent set of activities for software production. When you have read this chapter you will:

- understand the concepts of software processes and software process models;
- have been introduced to three generic software process models and when they might be used;
- know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;
- understand why processes should be organized to cope with changes in the software requirements and design;
- understand how the Rational Unified Process integrates good software engineering practice to create adaptable software processes.

Contents

2.1 Software process models
2.2 Process activities
2.3 Coping with change
2.4 The Rational Unified Process

A software process is a set of related activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. However, business applications are not necessarily developed in this way. New business software is now often developed by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components.

There are many different software processes but all must include four activities that are fundamental to software engineering:

1. *Software specification* The functionality of the software and constraints on its operation must be defined.
2. *Software design and implementation* The software to meet the specification must be produced.
3. *Software validation* The software must be validated to ensure that it does what the customer wants.
4. *Software evolution* The software must evolve to meet changing customer needs.

In some form, these activities are part of all software processes. In practice, of course, they are complex activities in themselves and include sub-activities such as requirements validation, architectural design, unit testing, etc. There are also supporting process activities such as documentation and software configuration management.

When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc., and the ordering of these activities. However, as well as activities, process descriptions may also include:

1. Products, which are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. Roles, which reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, programmer, etc.
3. Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. There is no ideal process and most organizations have developed their own software development processes. Processes have evolved to take advantage of the capabilities of the people in an organization and the specific characteristics of the systems that are being developed. For some systems, such as critical systems, a very structured development process is required. For business systems, with rapidly changing requirements, a less formal, flexible process is likely to be more effective.

Sometimes, software processes are categorized as either plan-driven or agile processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan. In agile processes, which I discuss in Chapter 3, planning is incremental and it is easier to change the process to reflect changing customer requirements. As Boehm and Turner (2003) discuss, each approach is suitable for different types of software. Generally, you need to find a balance between plan-driven and agile processes.

Although there is no 'ideal' software process, there is scope for improving the software process in many organizations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering. Indeed, many organizations still do not take advantage of software engineering methods in their software development.

Software processes can be improved by process standardization where the diversity in software processes across an organization is reduced. This leads to improved communication and a reduction in training time, and makes automated process support more economical. Standardization is also an important first step in introducing new software engineering methods and techniques and good software engineering practice. I discuss software process improvement in more detail in Chapter 26.

2.1 Software process models

As I explained in Chapter 1, a software process model is a simplified representation of a software process. Each process model represents a process from a particular perspective, and thus provides only partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. In this section, I introduce a number of very general process models (sometimes called 'process paradigms') and present these from an architectural perspective. That is, we see the framework of the process but not the details of specific activities.

These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

The process models that I cover here are:

1. *The waterfall model* This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on.

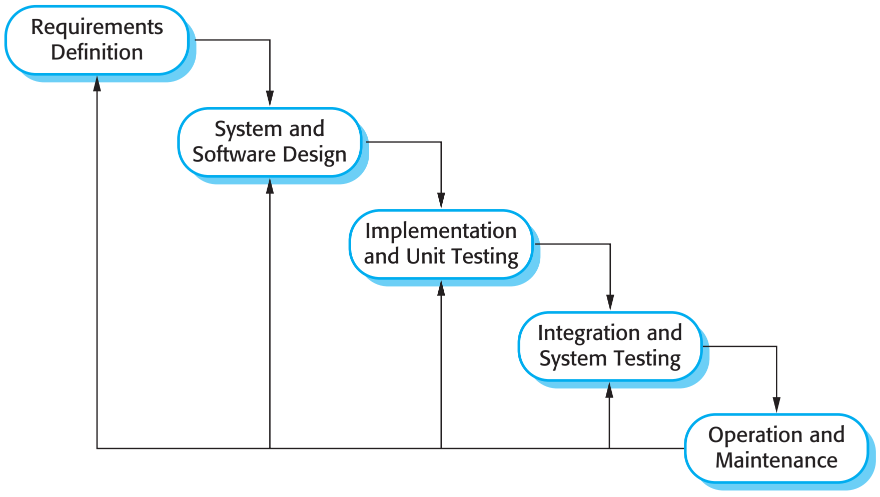

Requirements Definition
System and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Figure 2.1 The waterfall model

2. *Incremental development* This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.

3. *Reuse-oriented software engineering* This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

These models are not mutually exclusive and are often used together, especially for large systems development. For large systems, it makes sense to combine some of the best features of the waterfall and the incremental development models. You need to have information about the essential system requirements to design a software architecture to support these requirements. You cannot develop this incrementally. Sub-systems within a larger system may be developed using different approaches. Parts of the system that are well understood can be specified and developed using a waterfall-based process. Parts of the system which are difficult to specify in advance, such as the user interface, should always be developed using an incremental approach.

2.1.1 The waterfall model

The first published model of the software development process was derived from more general system engineering processes (Royce, 1970). This model is illustrated in Figure 2.1. Because of the cascade from one phase to another, this model is known as the 'waterfall model' or software life cycle. The waterfall model is an example of a plan-driven process—in principle, you must plan and schedule all of the process activities before starting work on them.

The principal stages of the waterfall model directly reflect the fundamental development activities:

1. *Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. *System and software design* The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
3. *Implementation and unit testing* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. *Operation and maintenance* Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase is one or more documents that are approved ('signed off'). The following phase should not start until the previous phase has finished. In practice, these stages overlap and feed information to each other. During design, problems with requirements are identified. During coding, design problems are found and so on. The software process is not a simple linear model but involves feedback from one phase to another. Documents produced in each phase may then have to be modified to reflect the changes made.

Because of the costs of producing and approving documents, iterations can be costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored, or programmed around. This premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

Cleanroom software engineering

An example of a formal development process, originally developed by IBM, is the Cleanroom process. In the Cleanroom process each software increment is formally specified and this specification is transformed into an implementation. Software correctness is demonstrated using a formal approach. There is no unit testing for defects in the process and the system testing is focused on assessing the system's reliability.

The objective of the Cleanroom process is zero-defects software so that delivered systems have a high level of reliability.

http://www.SoftwareEngineering-9.com/Web/Cleanroom/

The waterfall model is consistent with other engineering process models and documentation is produced at each phase. This makes the process visible so managers can monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.

In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development. However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Based on the assumption that your mathematical transformations are correct, you can therefore make a strong argument that a program generated in this way is consistent with its specification.

Formal development processes, such as that based on the B method (Schneider, 2001; Wordsworth, 1996) are particularly suited to the development of systems that have stringent safety, reliability, or security requirements. The formal approach simplifies the production of a safety or security case. This demonstrates to customers or regulators that the system actually meets its safety or security requirements.

Processes based on formal transformations are generally only used in the development of safety-critical or security-critical systems. They require specialized expertise. For the majority of systems this process does not offer significant cost-benefits over other approaches to system development.

2.1.2 Incremental development

Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed (Figure 2.2). Specification, development, and

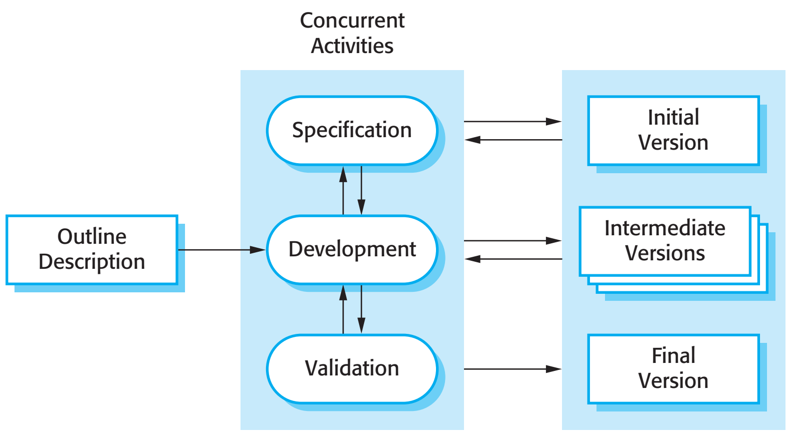


Figure 2.2 Incremental development

validation activities are interleaved rather than separate, with rapid feedback across activities.

Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems. Incremental development reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the customer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

Incremental development has three important benefits, compared to the waterfall model:

1. The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.

2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented. Customers find it difficult to judge progress from software design documents.

3. More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Problems with incremental development

Although incremental development has many advantages, it is not problem-free. The primary cause of the difficulty is the fact that large organizations have bureaucratic procedures that have evolved over time and there may be a mismatch between these procedures and a more informal iterative or agile process.

Sometimes these procedures are there for good reasons—for example, there may be procedures to ensure that the software properly implements external regulations (e.g., in the United States, the Sarbanes-Oxley accounting regulations). Changing these procedures may not be possible so process conflicts may be unavoidable.

http://www.SoftwareEngineering-9.com/Web/IncrementalDev/

Incremental development in some form is now the most common approach for the development of application systems. This approach can be either plan-driven, agile, or, more usually, a mixture of these approaches. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified but the development of later increments depends on progress and customer priorities.

From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
2. System structure tends to degrade as new increments are added. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. Large systems need a stable framework or architecture and the responsibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than developed incrementally.

You can develop a system incrementally and expose it to customers for comment, without actually delivering it and deploying it in the customer's environment. Incremental delivery and deployment means that the software is used in real, operational processes. This is not always possible as experimenting with new software can disrupt normal business processes. I discuss the advantages and disadvantages of incremental delivery in Section 2.3.2.

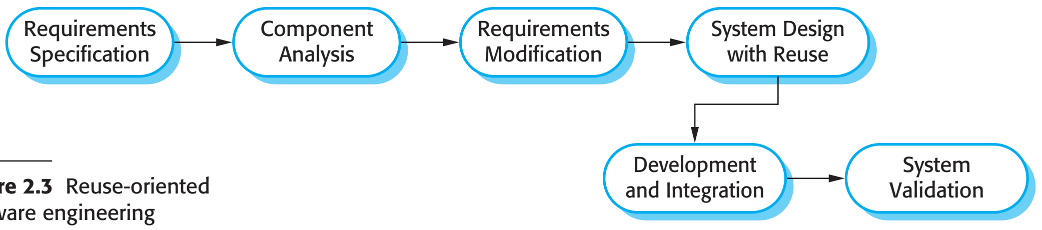

Requirements Specification
Component Analysis
Requirements Modification
System Design with Reuse
Development and Integration
System Validation

Figure 2.3 Reuse-oriented software engineering

2.1.3 Reuse-oriented software engineering

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incorporate them into their system.

This informal reuse takes place irrespective of the development process that is used. However, in the 21st century, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a large base of reusable software components and an integrating framework for the composition of these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as word processing or a spreadsheet.

A general process model for reuse-based development is shown in Figure 2.3. Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuse-oriented process are different. These stages are:

1. *Component analysis* Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.
2. *Requirements modification* During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.
3. *System design with reuse* During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater for this. Some new software may have to be designed if reusable components are not available.
4. *Development and integration* Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

There are three types of software component that may be used in a reuse-oriented process:

1. Web services that are developed according to service standards and which are available for remote invocation.
2. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
3. Stand-alone software systems that are configured for use in a particular environment.

Reuse-oriented software engineering has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable and this may lead to a system that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

Software reuse is very important and I have dedicated several chapters in the third part of the book to this topic. General issues of software reuse and COTS reuse are covered in Chapter 16, component-based software engineering in Chapters 17 and 18, and service-oriented systems in Chapter 19.

2.2 Process activities

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. Software developers use a variety of different software tools in their work. Tools are particularly useful for supporting the editing of different types of document and for managing the immense volume of detailed information that is generated in a large software project.

The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved. How these activities are carried out depends on the type of software, people, and organizational structures involved. In extreme programming, for example, specifications are written on cards. Tests are executable and developed before the program itself. Evolution may involve substantial system restructuring or refactoring.

2.2.1 Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a

Software development tools

Software development tools (sometimes called Computer-Aided Software Engineering or CASE tools) are programs that are used to support software engineering process activities. These tools therefore include design editors, data dictionaries, compilers, debuggers, system building tools, etc.

Software tools provide process support by automating some process activities and by providing information about the software that is being developed. Examples of activities that can be automated include:

- The development of graphical system models as part of the requirements specification or the software design
- The generation of code from these graphical models
- The generation of user interfaces from a graphical interface description that is created interactively by the user
- Program debugging through the provision of information about an executing program
- The automated translation of programs written using an old version of a programming language to a more recent version

Tools may be combined within a framework called an Interactive Development Environment or IDE. This provides a common set of facilities that tools can use so that it is easier for tools to communicate and operate in an integrated way. The ECLIPSE IDE is widely used and has been designed to incorporate many different types of software tools.

http://www.SoftwareEngineering-9.com/Web/CASE/

particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation.

The requirements engineering process (Figure 2.4) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

There are four main activities in the requirements engineering process:

1. *Feasibility study* An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

2. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.

3. *Requirements specification* Requirements specification is the activity of translating the information gathered during the analysis activity into a document that

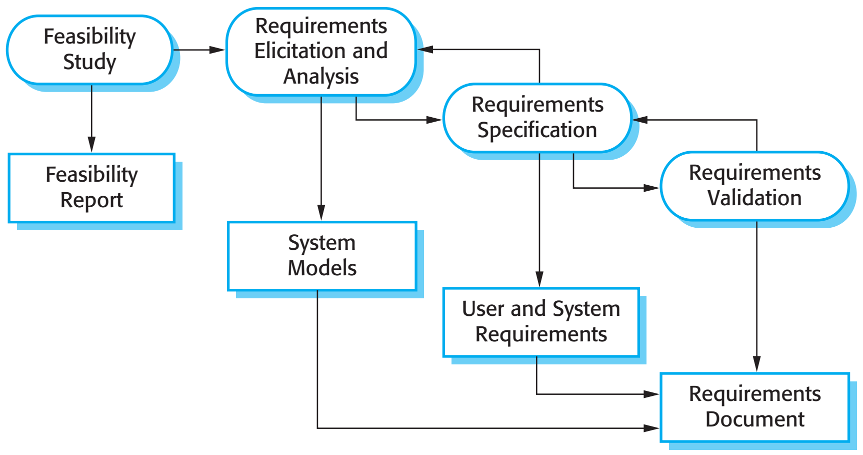


Figure 2.4 The requirements engineering process

defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.

4. *Requirements validation* This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

Of course, the activities in the requirements process are not simply carried out in a strict sequence. Requirements analysis continues during definition and specification and new requirements come to light throughout the process. Therefore, the activities of analysis, definition, and specification are interleaved. In agile methods, such as extreme programming, requirements are developed incrementally according to user priorities and the elicitation of requirements comes from users who are part of the development team.

2.2.2 Software design and implementation

The implementation stage of software development is the process of converting a system specification into an executable system. It always involves processes of software design and programming but, if an incremental approach to development is used, may also involve refinement of the software specification.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design iteratively. They add formality and detail as they develop their design with constant backtracking to correct earlier designs.

Figure 2.5 is an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.

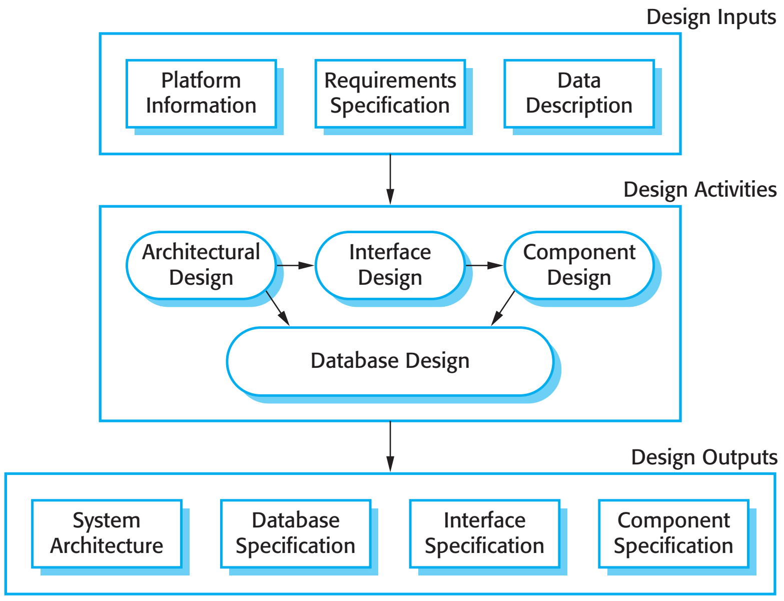


Figure 2.5 A general model of the design process

The diagram suggests that the stages of the design process are sequential. In fact, design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.

Most software interfaces with other software systems. These include the operating system, database, middleware, and other application systems. These make up the 'software platform', the environment in which the software will execute. Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with the software's environment. The requirements specification is a description of the functionality the software must provide and its performance and dependability requirements. If the system is to process existing data, then the description of that data may be included in the platform specification; otherwise, the data description must be an input to the design process so that the system data organization to be defined.

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require timing design but may not include a database so there is no database design involved. Figure 2.5 shows four activities that may be part of the design process for information systems:

1. *Architectural design,* where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.

2. *Interface design,* where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.

Structured methods

Structured methods are an approach to software design in which graphical models that should be developed as part of the design process are defined. The method may also define a process for developing the models and rules that apply to each model type. Structured methods lead to standardized documentation for a system and are particularly useful in providing a development framework for less-experienced and less-expert software developers.

http://www.SoftwareEngineering-9.com/Web/Structured-methods/

3. *Component design,* where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.

4. *Database design,* where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

These activities lead to a set of design outputs, which are also shown in Figure 2.5. The detail and representation of these vary considerably. For critical systems, detailed design documents setting out precise and accurate descriptions of the system must be produced. If a model-driven approach is used, these outputs may mostly be diagrams. Where agile methods of development are used, the outputs of the design process may not be separate specification documents but may be represented in the code of the program.

Structured methods for design were developed in the 1970s and 1980s and were the precursor to the UML and object-oriented design (Budgen, 2003). They rely on producing graphical models of the system and, in many cases, automatically generating code from these models. Model-driven development (MDD) or model-driven engineering (Schmidt, 2006), where models of the software are created at different levels of abstraction, is an evolution of structured methods. In MDD, there is greater emphasis on architectural models with a separation between abstract implementation-independent models and implementation-specific models. The models are developed in sufficient detail so that the executable system can be generated from them. I discuss this approach to development in Chapter 5.

The development of a program to implement the system follows naturally from the system design processes. Although some classes of program, such as safety-critical systems, are usually designed in detail before any implementation begins, it is more common for the later stages of design and program development to be interleaved. Software development tools may be used to generate a skeleton program from a design. This includes code to define and implement interfaces, and, in many cases, the developer need only add details of the operation of each program component.

Programming is a personal activity and there is no general process that is usually followed. Some programmers start with components that they understand, develop these, and then move on to less-understood components. Others take the opposite

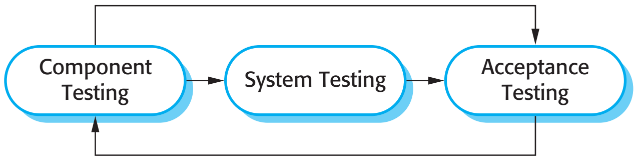


Figure 2.6 Stages of testing

approach, leaving familiar components till last because they know how to develop them. Some developers like to define data early in the process then use this to drive the program development; others leave data unspecified for as long as possible.

Normally, programmers carry out some testing of the code they have developed. This often reveals program defects that must be removed from the program. This is called debugging. Defect testing and debugging are different processes. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.

When you are debugging, you have to generate hypotheses about the observable behavior of the program then test these hypotheses in the hope of finding the fault that caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debugging tools, which show the intermediate values of program variables and a trace of the statements executed, may be used to support the debugging process.

2.2.3 Software validation

Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. Because of the predominance of testing, the majority of validation costs are incurred during and after implementation.

Except for small programs, systems should not be tested as a single, monolithic unit. Figure 2.6 shows a three-stage testing process in which system components are tested then the integrated system is tested and, finally, the system is tested with the customer's data. Ideally, component defects are discovered early in the process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

The stages in the testing process are:

1. *Development testing* The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions

or object classes, or may be coherent groupings of these entities. Test automation tools, such as JUnit (Massol and Husted, 2003), that can re-run component tests when new versions of the component are created, are commonly used.

2. *System testing* System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multi-stage process where components are integrated to form sub-systems that are individually tested before these sub-systems are themselves integrated to form the final system.

3. *Acceptance testing* This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable.

Normally, component development and testing processes are interleaved. Programmers make up their own test data and incrementally test the code as it is developed. This is an economically sensible approach, as the programmer knows the component and is therefore the best person to generate test cases.

If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment. In extreme programming, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.

When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans. An independent team of testers works from these pre-formulated test plans, which have been developed from the system specification and design. Figure 2.7 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development (turn it on its side to see the V).

Acceptance testing is sometimes called 'alpha testing'. Custom systems are developed for a single client. The alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implementation of the requirements.

When a system is to be marketed as a software product, a testing process called 'beta testing' is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the system builders. After this feedback, the system is modified and released either for further beta testing or for general sale.

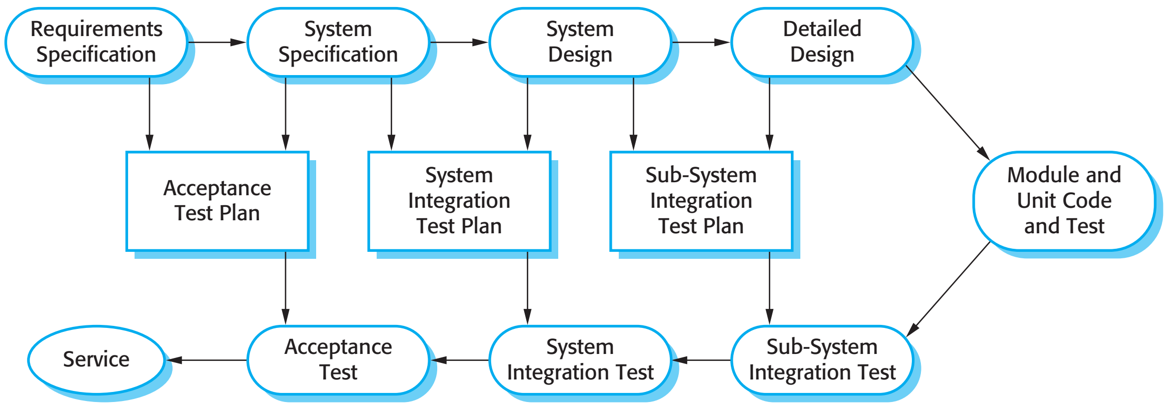

Requirements Specification
System Specification
System Design
Detailed Design
Acceptance Test Plan
System Integration Test Plan
Sub-System Integration Test Plan
Module and Unit Code and Test
Service
Acceptance Test
System Integration Test
Sub-System Integration Test

Figure 2.7 Testing phases in a plan-driven software process

2.2.4 Software evolution

The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. Although the costs of maintenance are often several times the initial development costs, maintenance processes are sometimes considered to be less challenging than original software development.

This distinction between development and maintenance is increasingly irrelevant. Hardly any software systems are completely new systems and it makes much more sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.8) where software is continually changed over its lifetime in response to changing requirements and customer needs.

2.3 Coping with change

Change is inevitable in all large software projects. The system requirements change as the business procuring the system responds to external pressures and management priorities change. As new technologies become available, new design and implementation possibilities emerge. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed.

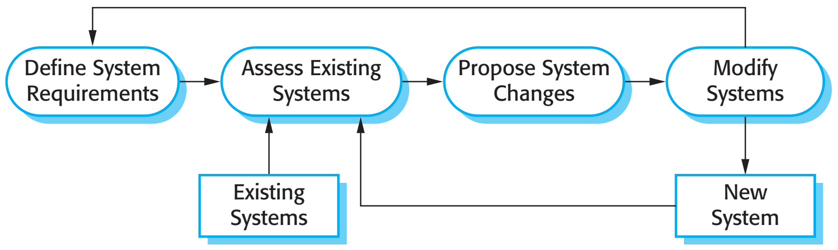


Figure 2.8 System evolution

Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new requirements, change any programs that have been developed, and re-test the system.

There are two related approaches that may be used to reduce the costs of rework:

1. Change avoidance, where the software process includes activities that can anticipate possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.

2. Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost. This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.

In this section, I discuss two ways of coping with change and changing system requirements. These are:

1. System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions. This supports change avoidance as it allows users to experiment with the system before delivery and so refine their requirements. The number of requirements change proposals made after delivery is therefore likely to be reduced.

2. Incremental delivery, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low cost.

The notion of refactoring, namely improving the structure and organization of a program, is also an important mechanism that supports change tolerance. I discuss this in Chapter 3, which covers agile methods.

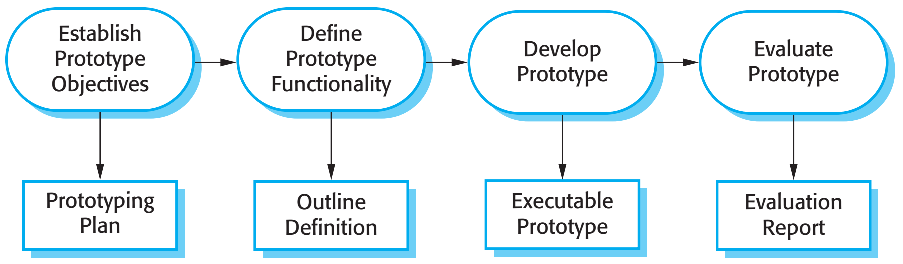


Figure 2.9 The process of prototype development

2.3.1 Prototyping

A prototype is an initial version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions. Rapid, iterative development of the prototype is essential so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.

A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore particular software solutions and to support user interface design.

System prototypes allow users to see how well the system supports their work. They may get new ideas for requirements, and find areas of strength and weakness in the software. They may then propose new system requirements. Furthermore, as the prototype is developed, it may reveal errors and omissions in the requirements that have been proposed. A function described in a specification may seem useful and well defined. However, when that function is combined with other functions, users often find that their initial view was incorrect or incomplete. The system specification may then be modified to reflect their changed understanding of the requirements.

A system prototype may be used while the system is being designed to carry out design experiments to check the feasibility of a proposed design. For example, a database design may be prototyped and tested to check that it supports efficient data access for the most common user queries. Prototyping is also an essential part of the user interface design process. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements. Therefore, rapid prototyping with end-user involvement is the only sensible way to develop graphical user interfaces for software systems.

A process model for prototype development is shown in Figure 2.9. The objectives of prototyping should be made explicit from the start of the process. These may be to develop a system to prototype the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the feasibility of the application to managers. The same prototype cannot meet all objectives. If the objectives are left unstated, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Provision must be made during this stage for user training and the prototype objectives should be used to derive a plan for evaluation. Users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirements errors and omissions.

A general problem with prototyping is that the prototype may not necessarily be used in the same way as the final system. The tester of the prototype may not be typical of system users. The training time during prototype evaluation may be insufficient. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with better response in the final system, they may use it in a different way.

Developers are sometimes pressured by managers to deliver throwaway prototypes, particularly when there are delays in delivering the final version of the software. However, this is usually unwise:

1. It may be impossible to tune the prototype to meet non-functional requirements, such as performance, security, robustness, and reliability requirements, which were ignored during prototype development.
2. Rapid change during development inevitably means that the prototype is undocumented. The only design specification is the prototype code. This is not good enough for long-term maintenance.
3. The changes made during prototype development will probably have degraded the system structure. The system will be difficult and expensive to maintain.
4. Organizational quality standards are normally relaxed for prototype development.

Prototypes do not have to be executable to be useful. Paper-based mock-ups of the system user interface (Rettig, 1994) can be effective in helping users refine an interface design and work through usage scenarios. These are very cheap to develop and can be constructed in a few days. An extension of this technique is a Wizard of Oz prototype where only the user interface is developed. Users interact with this interface but their requests are passed to a person who interprets them and outputs the appropriate response.

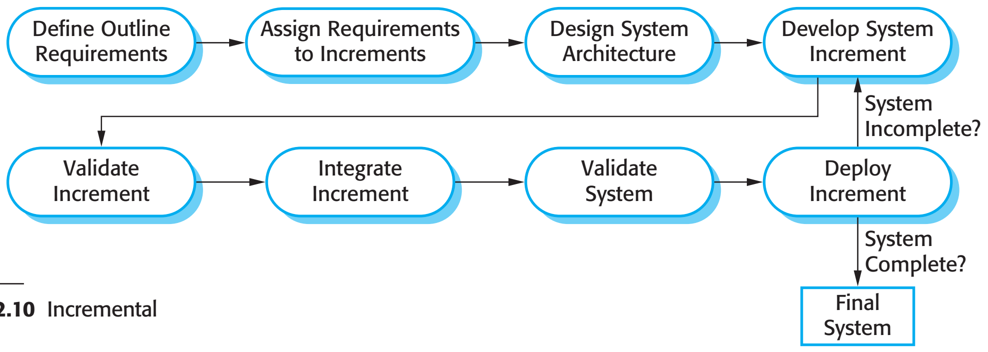


Figure 2.10 Incremental delivery

2.3.2 Incremental delivery

Incremental delivery (Figure 2.10) is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in an operational environment. In an incremental delivery process, customers identify, in outline, the services to be provided by the system. They identify which of the services are most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a sub-set of the system functionality. The allocation of services to increments depends on the service priority, with the highest-priority services implemented and delivered first.

Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail and that increment is developed. During development, further requirements analysis for later increments can take place but requirements changes for the current increment are not accepted.

Once an increment is completed and delivered, customers can put it into service. This means that they take early delivery of part of the system functionality. They can experiment with the system and this helps them clarify their requirements for later system increments. As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment.

Incremental delivery has a number of advantages:

1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system so there is no re-learning when the complete system is available.

2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.

3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.

4. As the highest-priority services are delivered first and increments then integrated, the most important system services receive the most testing. This means

that customers are less likely to encounter software failures in the most important parts of the system.

However, there are problems with incremental delivery:

1. Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
2. Iterative development can also be difficult when a replacement system is being developed. Users want all of the functionality of the old system and are often unwilling to experiment with an incomplete new system. Therefore, getting useful customer feedback is difficult.
3. The essence of iterative processes is that the specification is developed in conjunction with the software. However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

There are some types of system where incremental development and delivery is not the best approach. These are very large systems where development may involve teams working in different locations, some embedded systems where the software depends on hardware development and some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system.

These systems, of course, suffer from the same problems of uncertain and changing requirements. Therefore, to address these problems and get some of the benefits of incremental development, a process may be used in which a system prototype is developed iteratively and used as a platform for experiments with the system requirements and design. With the experience gained from the prototype, definitive requirements can then be agreed.

2.3.3 Boehm's spiral model

A risk-driven software process framework (the spiral model) was proposed by Boehm (1988). This is shown in Figure 2.11. Here, the software process is represented as a spiral, rather than a sequence of activities with some backtracking from one activity to another. Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model combines change avoidance with change tolerance. It assumes that

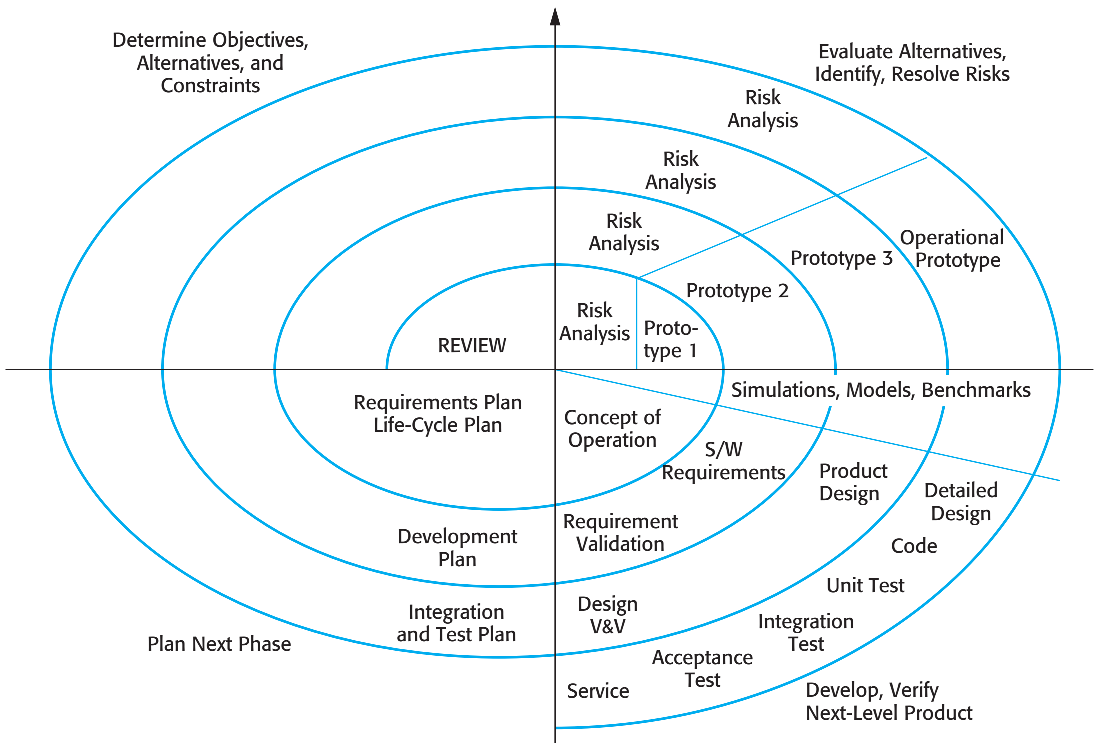


Figure 2.11 Boehm's spiral model of the software process (©IEEE 1988)

changes are a result of project risks and includes explicit risk management activities to reduce these risks.

Each loop in the spiral is split into four sectors:

1. *Objective setting* Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.

2. *Risk assessment and reduction* For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3. *Development and validation* After risk evaluation, a development model for the system is chosen. For example, throwaway prototyping may be the best development approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-system integration, the waterfall model may be the best development model to use.

4. *Planning* The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is its explicit recognition of risk. A cycle of the spiral begins by elaborating objectives such as performance and functionality. Alternative ways of achieving these objectives, and dealing with the constraints on each of them, are then enumerated. Each alternative is assessed against each objective and sources of project risk are identified. The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping, and simulation.

Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code. Risks lead to proposed software changes and project problems such as schedule and cost overrun, so risk minimization is a very important project management activity. Risk management, an essential part of project management, is covered in Chapter 22.

2.4 The Rational Unified Process

The Rational Unified Process (RUP) (Krutchen, 2003) is an example of a modern process model that has been derived from work on the UML and the associated Unified Software Development Process (Rumbaugh, et al., 1999; Arlow and Neustadt, 2005). I have included a description here, as it is a good example of a hybrid process model. It brings together elements from all of the generic process models (Section 2.1), illustrates good practice in specification and design (Section 2.2) and supports prototyping and incremental delivery (Section 2.3).

The RUP recognizes that conventional process models present a single view of the process. In contrast, the RUP is normally described from three perspectives:

1. A dynamic perspective, which shows the phases of the model over time.

2. A static perspective, which shows the process activities that are enacted.

3. A practice perspective, which suggests good practices to be used during the process.

Most descriptions of the RUP attempt to combine the static and dynamic perspectives in a single diagram (Krutchen, 2003). I think that makes the process harder to understand, so I use separate descriptions of each of these perspectives.

The RUP is a phased model that identifies four discrete phases in the software process. However, unlike the waterfall model where phases are equated with process activities, the phases in the RUP are more closely related to business rather than technical concerns. Figure 2.11 shows the phases in the RUP. These are:

1. *Inception* The goal of the inception phase is to establish a business case for the system. You should identify all external entities (people and systems) that will

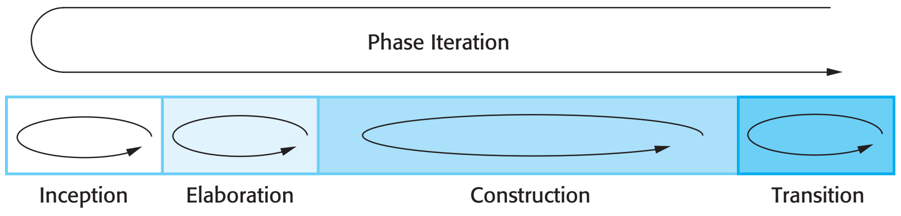


Figure 2.12 Phases in the Rational Unified Process

interact with the system and define these interactions. You then use this information to assess the contribution that the system makes to the business. If this contribution is minor, then the project may be cancelled after this phase.

2. *Elaboration* The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks. On completion of this phase you should have a requirements model for the system, which may be a set of UML use-cases, an architectural description, and a development plan for the software.

3. *Construction* The construction phase involves system design, programming, and testing. Parts of the system are developed in parallel and integrated during this phase. On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.

4. *Transition* The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment. This is something that is ignored in most software process models but is, in fact, an expensive and sometimes problematic activity. On completion of this phase, you should have a documented software system that is working correctly in its operational environment.

Iteration within the RUP is supported in two ways. Each phase may be enacted in an iterative way with the results developed incrementally. In addition, the whole set of phases may also be enacted incrementally, as shown by the looping arrow from Transition to Inception in Figure 2.12.

The static view of the RUP focuses on the activities that take place during the development process. These are called workflows in the RUP description. There are six core process workflows identified in the process and three core supporting workflows. The RUP has been designed in conjunction with the UML, so the workflow description is oriented around associated UML models such as sequence models, object models, etc. The core engineering and support workflows are described in Figure 2.13.

The advantage in presenting dynamic and static views is that phases of the development process are not associated with specific workflows. In principle at least, all of the RUP workflows may be active at all stages of the process. In the early phases of the process, most effort will probably be spent on workflows such as business modelling and requirements and, in the later phases, in testing and deployment.

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models, and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users, and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Figure 2.13 Static workflows in the Rational Unified Process

The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development. Six fundamental best practices are recommended:

1. *Develop software iteratively* Plan increments of the system based on customer priorities and develop the highest-priority system features early in the development process.

2. *Manage requirements* Explicitly document the customer's requirements and keep track of changes to these requirements. Analyze the impact of changes on the system before accepting them.

3. *Use component-based architectures* Structure the system architecture into components, as discussed earlier in this chapter.

4. *Visually model software* Use graphical UML models to present static and dynamic views of the software.

5. *Verify software quality* Ensure that the software meets the organizational quality standards.

6. *Control changes to software* Manage changes to the software using a change management system and configuration management procedures and tools.

The RUP is not a suitable process for all types of development, e.g., embedded software development. However, it does represent an approach that potentially combines the three generic process models discussed in Section 2.1. The most important innovations in the RUP are the separation of phases and workflows, and the recognition that deploying software in a user's environment is part of the process. Phases are dynamic and have goals. Workflows are static and are technical activities that are not associated with a single phase but may be used throughout the development to achieve the goals of each phase.

KEY POINTS

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- General process models describe the organization of software processes. Examples of these general models include the waterfall model, incremental development, and reuse-oriented development.
- Requirements engineering is the process of developing a software specification. Specifications are intended to communicate the system needs of the customer to the system developers.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system. Systematic design methods may be used as part of this transformation.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. Changes are continuous and the software must evolve to remain useful.
- Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design. Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- The Rational Unified Process is a modern generic process model that is organized into phases (inception, elaboration, construction, and transition) but separates activities (requirements, analysis, and design, etc.) from these phases.

FURTHER READING

Managing Software Quality and Business Risk. This is primarily a book about software management but it includes an excellent chapter (Chapter 4) on process models. (M. Ould, John Wiley and Sons Ltd, 1999.)

Process Models in Software Engineering. This is an excellent overview of a wide range of software engineering process models that have been proposed. (W. Scacchi, *Encyclopaedia of Software Engineering,* ed. J.J. Marciniak, John Wiley and Sons, 2001.) http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf.

The Rational Unified Process—An Introduction (3rd edition). This is the most readable book available on the RUP at the time of this writing. Krutchen describes the process well, but I would like to have seen more on the practical difficulties of using the process. (P. Krutchen, Addison-Wesley, 2003.)

EXERCISES

2.1. Giving reasons for your answer based on the type of system being developed, suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems:

A system to control anti-lock braking in a car

A virtual reality system to support software maintenance

A university accounting system that replaces an existing system

An interactive travel planning system that helps users plan journeys with the lowest environmental impact

2.2. Explain why incremental development is the most effective approach for developing business software systems. Why is this model less appropriate for real-time systems engineering?

2.3. Consider the reuse-based process model shown in Figure 2.3. Explain why it is essential to have two separate requirements engineering activities in the process.

2.4. Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process.

2.5. Describe the main activities in the software design process and the outputs of these activities. Using a diagram, show possible relationships between the outputs of these activities.

2.6. Explain why change is inevitable in complex systems and give examples (apart from prototyping and incremental delivery) of software process activities that help predict changes and make the software being developed more resilient to change.

2.7. Explain why systems developed as prototypes should not normally be used as production systems.

2.8. Explain why Boehm's spiral model is an adaptable model that can support both change avoidance and change tolerance activities. In practice, this model has not been widely used. Suggest why this might be the case.

2.9. What are the advantages of providing static and dynamic views of the software process as in the Rational Unified Process?

2.10. Historically, the introduction of technology has caused profound changes in the labor market and, temporarily at least, displaced people from jobs. Discuss whether the introduction of extensive process automation is likely to have the same consequences for software engineers. If you don't think it will, explain why not. If you think that it will reduce job opportunities, is it ethical for the engineers affected to passively or actively resist the introduction of this technology?

REFERENCES

Arlow, J. and Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Boston: Addison-Wesley.

Boehm, B. and Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley.

Boehm, B. W. (1988). 'A Spiral Model of Software Development and Enhancement'. *IEEE Computer*, **21** (5), 61–72.

Budgen, D. (2003). *Software Design (2nd Edition)*. Harlow, UK.: Addison-Wesley.

Krutchen, P. (2003). *The Rational Unified Process—An Introduction (3rd Edition)*. Reading, MA: Addison-Wesley.

Massol, V. and Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.

Rettig, M. (1994). 'Practical Programmer: Prototyping for Tiny Fingers'. *Comm. ACM*, **37** (4), 21–7.

Royce, W. W. (1970). 'Managing the Development of Large Software Systems: Concepts and Techniques'. IEEE WESTCON, Los Angeles CA: 1–9.

Rumbaugh, J., Jacobson, I. and Booch, G. (1999). *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley.

Schmidt, D. C. (2006). 'Model-Driven Engineering'. *IEEE Computer*, **39** (2), 25–31.

Schneider, S. (2001). *The B Method*. Houndmills, UK: Palgrave Macmillan.

Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.