Introduction

Objectives

The objectives of this chapter are to introduce software engineering and to provide a framework for understanding the rest of the book. When you have read this chapter you will:

- understand what software engineering is and why it is important;
- understand that the development of different types of software systems may require different software engineering techniques;
- understand some ethical and professional issues that are important for software engineers;

1. We can’t run the modern world without software.
2. Software systems are abstract and intangible. They are not constrained by the properties of materials, governed by physical laws, or by manufacturing processes.
3. There are many different types of software systems, from simple embedded systems to complex, worldwide information systems.
4. There are still many reports of software projects going wrong and ‘software failures’. Software engineering is criticized as inadequate for modern software development.

However, **many of these so-called software failures are a consequence of two factors**:

1. **Increasing demands** As new software engineering techniques help us to build larger, more complex systems, the demands change**. Systems have to be built and delivered more quickly; larger, even more complex systems are required;** systems have to have new capabilities that were previously thought to be impossible. Existing software engineering methods cannot cope and new software engineering techniques have to be developed to meet new these new demands.
2. **Low expectations It** is relatively easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. **They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be**. **We need better software engineering education and training to address this problem**.

History of software engineering

The notion of ‘software engineering’ was first proposed in 1968 at a conference held to discuss what was then called the ‘software crisis’ (Naur and Randell, 1969). It became clear that individual approaches to program development did not scale up to large and complex software systems. These were unreliable, cost more than expected, and were delivered late.

Throughout the 1970s and 1980s, a variety of new software engineering techniques and methods were developed, such as structured programming, information hiding and object-oriented development. Tools and standard notations were developed and are now extensively used.

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

Software engineering is intended to support professional software development, rather than individual programming. It includes techniques that support program specification, design, and evolution, none of which are normally relevant for personal software development. To help you to get a broad view of what software engineering is about

Many people think that software is simply another word for computer programs. However, when we are talking about software engineering, software is not just the programs themselves but also all associated documentation and configuration data that is required to make these programs operate correctly. A professionally developed software system is often more than a single program. **The system usually consists of a number of separate programs and configuration files that are used to set up these programs. It may include system documentation, which describes the structure of the system; user documentation, which**

explains how to use the system, and websites for users to download recent product information.

This is one of the **important differences between professional and amateur software development**. If you are writing a program for yourself, no one else will use it and you don't have to worry about writing program guides, documenting the program design, etc. **However, if you are writing software that other people will use and other engineers will change then you usually have to provide additional information as well as the code of the program.**

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.
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 professionallymanaged 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.

There are two kinds of software products:

1. ***Generic products*** These are stand-alone systems that are produced by a development organization and **sold on the open market to any customer who is able to buy them**. Examples of this type of product include software for PCs such as databases, word processors, drawing packages, and project-management tools. It also includes so-called vertical applications designed for some specific purpose such as library information systems, accounting systems, or systems for maintaining dental records.
2. ***Customized (or bespoke) products*** These are systems that are commissioned by a particular customer. A software contractor develops the **software especially for that customer**. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

An important difference between these types of software is that,

- **In generic products**, the organization that develops the software controls the software specification.
- **For custom products**, the specification is usually developed and controlled by the organizationthat is buying the software. The software developers must work to that specification.

However, the distinction between these system product types is becoming increasingly blurred. More and more systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer. **Enterprise Resource Planning (ERP) systems**, such as the SAP system, are the best examples of this approach. Here, a large and complex system is adapted for a company by incorporating information about business rules and processes, reports required, and so on.

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, a banking system must be secure, an interactive game must be responsive, a telephone switching system must be reliable, and so on.
These can be generalized into the set of attributes shown in the table below, which I believe are the essential characteristics of a professional software system.

Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.

Software engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this definition, there are two key phrases:

1. ***Engineering discipline*** Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constraints so they look for solutions within these constraints.
2. ***All aspects of software production*** Software engineering i***s not just concerned with the technical processes*** of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software production.

Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project**. For most types of systems, the majority of costs are the costs of changing the software after it has gone into use**!!!!!!!!!

There are four fundamental activities that are common to all software processes. These activities are:

1. **Software specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. **Software development,** where the software is designed and programmed.
3. **Software validation,** where the software is checked to ensure that it is what the customer requires.
4. **Software evolution,** where the software is modified to reflect changing customer and market requirements.

There are many different types of software. There is no universal software engineering method or technique that is applicable for all of these.
However, there are three general issues that affect many different types of software:

1. ***Heterogeneity*** Increasingly, systems are required to operate as distributed systemsacross networks that include different types of computer and mobile devices. Aswell as running on general-purpose computers, software may also have to executeon mobile phones. You often have to integrate new software with older legacy systems written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope withthis heterogeneity.
2. ***Business and social change*** Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. Theyneed to be able to change their existing software and to rapidly develop new software. Many traditional software engineering techniques are time consuming and delivery of new systems often takes longer than planned. They need to evolve so that the time required for software to deliver value to its customers is reduced.
3. ***Security and trust*** As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface. We have to make sure that malicious users cannot attack our software and that information security is maintained.

Perhaps the most significant factor in determining which software engineering methods and techniques are most important is the type of application that is being developed. There are many different types of application including:

1. ***Stand-alone applications*** These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, etc.
2. **Interactive transaction-based applications** These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals. Obviously, these include web applications such as e-commerce applications where you can interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.
3. **Embedded control systems** These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system. Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.
4. **Batch processing systems** These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.
5. **Entertainment systems** These are systems that are primarily for personal use and which are intended to entertain the user. Most of these systems are games of one kind or another. The

quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.

6. **Systems for modeling and simulation** These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.
7. **Data collection systems** These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing. The software has to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location.
8. **Systems of systems** These are systems that are composed of a number of other software systems. Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

Nevertheless, **there are software engineering fundamentals that apply to all types of software system:**

1. They should be developed using a **managed and understood development process**. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed. Of course, different processes are used for different types of software.
2. **Dependability and performance** are important for all types of systems. Software should behave as expected, without failures and should be available for use when it is required. It should be safe in its operation and, as far as possible, should be secure against external attack. The system should perform efficiently and should not waste resources.
3. **Understanding and managing the software specification** and requirements (what the software should do) are important. You have to know what different customers and users of the system expect from it and you have to manage their expectations so that a useful system can be delivered within budget and to schedule.
4. You should make **as effective use as possible of existing resources**. This means that, where appropriate, you should reuse software that has already been developed rather than write new software.

The advent of the web, therefore, has led to a significant change in the way that business software is organized. Before the web, business applications were mostly monolithic, single programs running on single computers or computer clusters.
Communications were local, within an organization. Now, software is highly distributed, sometimes across the world. Business applications are not programmed from scratch but involve extensive reuse of components and programs.
This radical change in software organization has, obviously, led to changes in the ways that web-based systems are engineered.

For example:

1. Software reuse has become the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from pre-existing software components and systems.

It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems should be developed and delivered incrementally.

User interfaces are constrained by the capabilities of web browsers. Although technologies such as AJAX (Holdener, 2008) mean that rich interfaces can be created within a web browser, these technologies are still difficult to use. Web forms with local scripting are more commonly used.
Application interfaces on web-based systems are often poorer than the specially designed user interfaces on PC system products.

Software engineering ethics

1. ***Confidentiality*** You should normally respect the confidentiality of your employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
2. ***Competence*** You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
3. ***Intellectual property rights*** You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.
4. ***Computer misuse*** You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses or other malware).

Case studies

To illustrate software engineering concepts,we can use examples from three different types of systems. The reason why we have not used a single case study is that one of the key messages in this book is that software engineering practice depends on the type of systems being produced. I therefore choose an appropriate example when discussing concepts such as safety and dependability, system modeling, reuse, etc.

Three types of systems that can use as case studies are:
1. ***An embedded system*** This is a system where the software controls a hardwaredevice and is embedded in that device. Issues in embedded systems typically include physical size, responsiveness, power management, etc. The example of an embedded system that I use is a software system to control a medical device.
2. ***An information system*** This is a system whose primary purpose is to manage and provide access to a database of information. Issues in information systems include security, usability, privacy, and maintaining data integrity. The example of an information system that I use is a medical records system.
3. ***A sensor-based data collection system*** This is a system whose primary purpose is to collect data from a set of sensors and process that data in some way. The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability. The example of a data collection system that I use is a wilderness weather station.

1.3.1 ***An insulin pump control system***
1.3.2 ***A patient information system for mental health care***
1.3.3 ***A wilderness weather station***

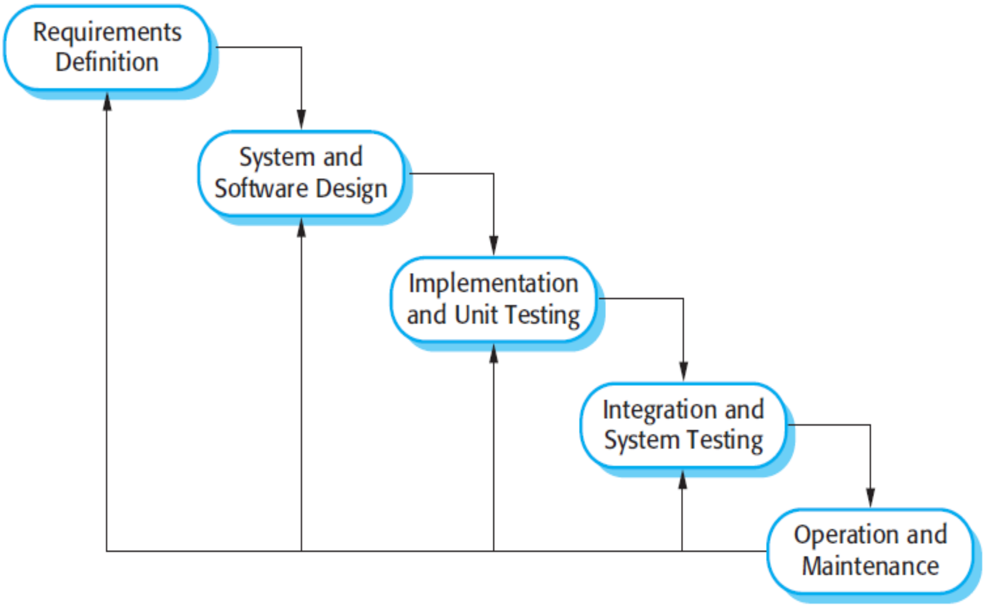
EXERCISES

1.1. Explain why professional software is not just the programs that are developed for a customer.
1.2. What is the most important difference between generic software product development and custom software development? What might this mean in practice for users of generic software products?
1.3. What are the four important attributes that all professional software should have? Suggest four other attributes that may sometimes be significant.
1.4. Apart from the challenges of heterogeneity, business and social change, and trust and security, identify other problems and challenges that software engineering is likely to face in the 21st century (Hint: think about the environment).
1.5. Based on your own knowledge of some of the application types discussed in section 1.1.2, explain, with examples, why different application types require specialized software engineering techniques to support their design and development.
1.6. Explain why there are fundamental ideas of software engineering that apply to all types of software systems.
1.7. Explain how the universal use of the Web has changed software systems.
1.8. Discuss whether professional engineers should be certified in the same way as doctors or lawyers.
1.9. For each of the clauses in the ACM/IEEE Code of Ethics shown in Figure 1.3, suggest an appropriate example that illustrates that clause.

1.10. To help counter terrorism, many countries are planning or have developed computer systems that track large numbers of their citizens and their actions. Clearly this has privacy implications. Discuss the ethics of working on the development of this type of system.

Software process models

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

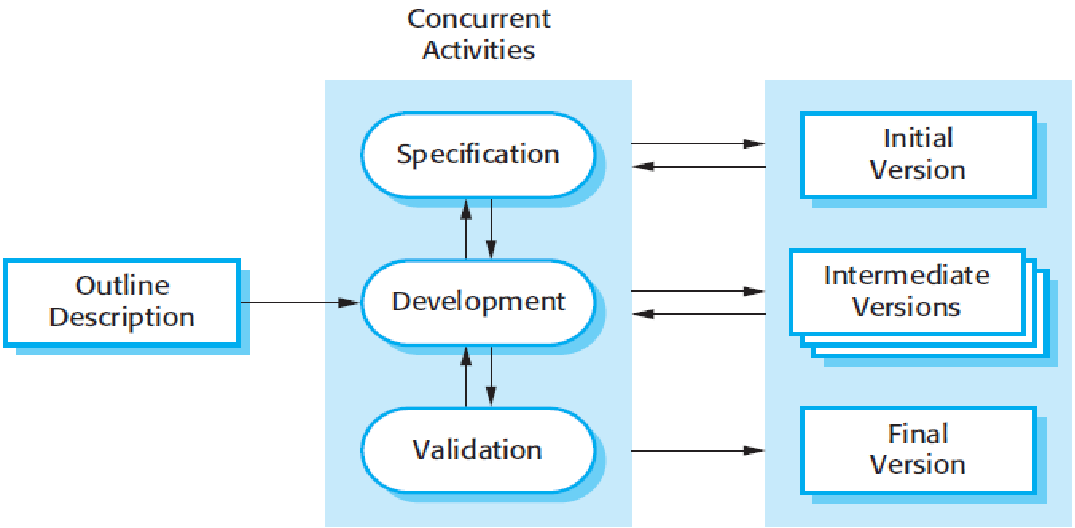
Cleanroom software engineering An example of a formal development process, originally developed by IBM, is the Cleanroom process. In the Cleanroom process each software increment is formally specified and this specification is transformed into an implementation. Software correctness is demonstrated using 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, **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.

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.



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.

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

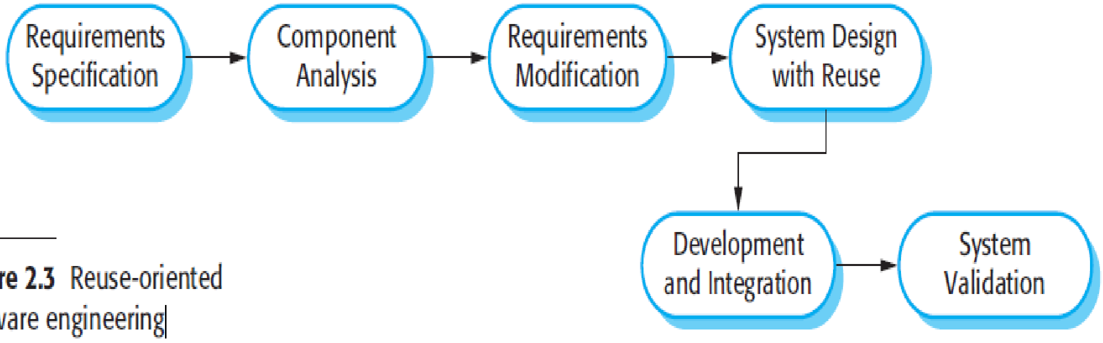

Requirements Specification
Component Analysis
Requirements Modification
System Design with Reuse
Development and Integration
System Validation

Figure 2.3 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 reuseoriented process are different**.

These stages are:

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.

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**

1. **of specification,**
2. **development,**
3. **validation,**
4. **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.

Software development tools:

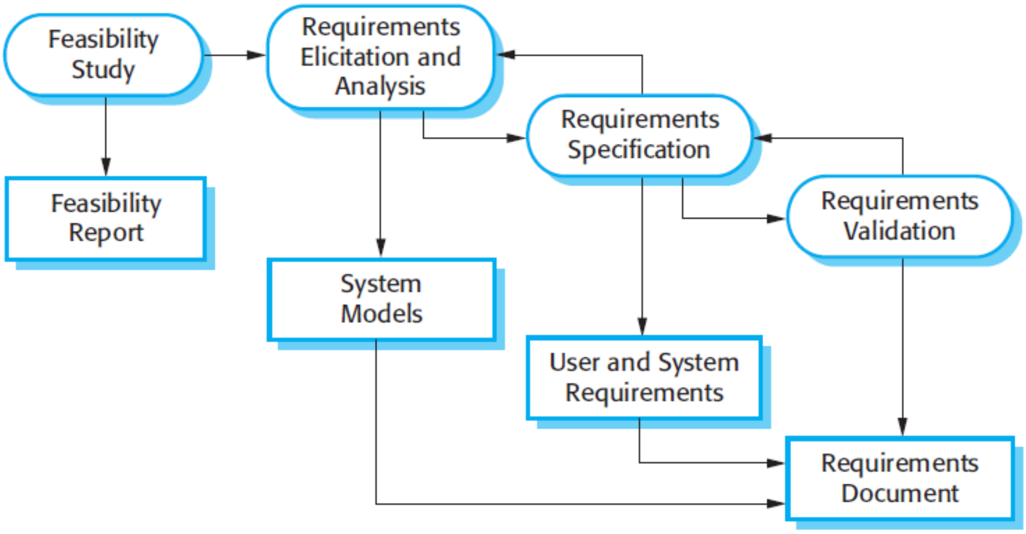
Software development tools (sometimes called Computer-Aided Software Engineering or CASE tools) are programs that are used to support software engineering process activities. These tools therefore include design editors, data dictionaries, compilers, debuggers, system building tools, etc.

Software tools provide process support by automating some process activities and by providing information about the software that is being developed.

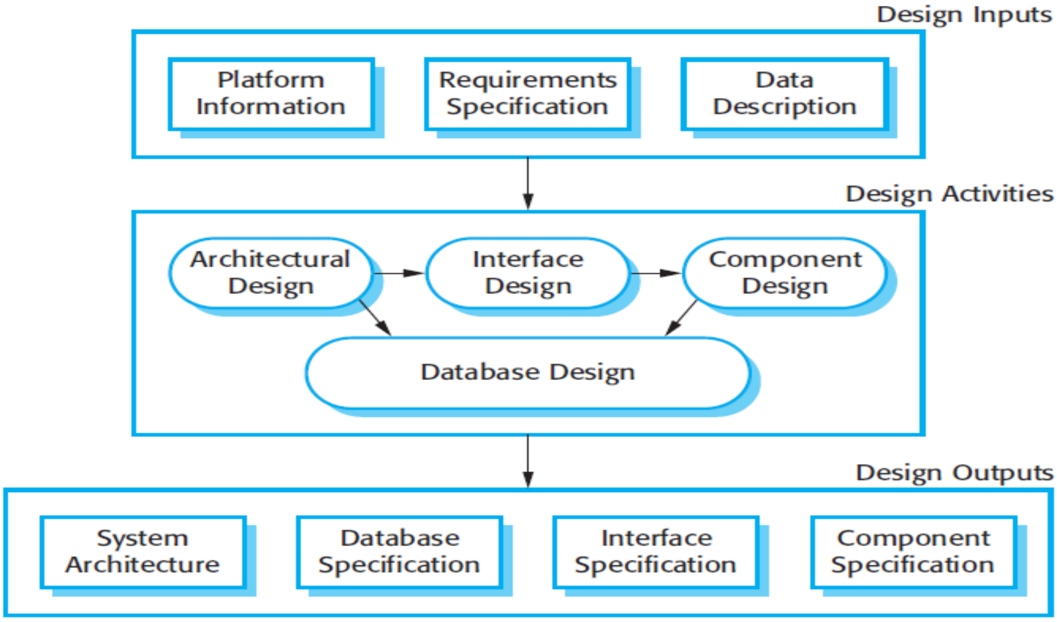
Examples of activities that can be automated include:

- **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**

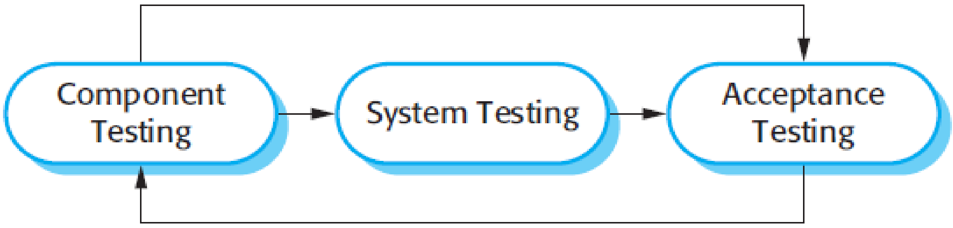
Tools may be combined within a framework called an Interactive Development Environment or IDE. This provides a common set of facilities that tools can use so that it is easier for tools to communicate and operate in an integrated way. The ECLIPSE IDE is widely used and has been designed to incorporate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



Software design and implementation



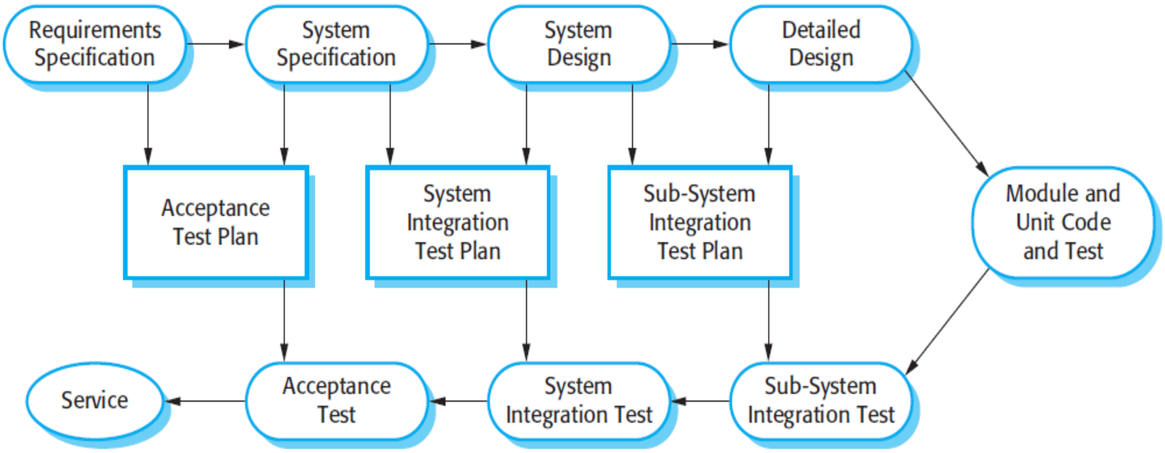
The diagram suggests that the stages of the design process are sequential. In fact, design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.



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.

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 subsystems 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.

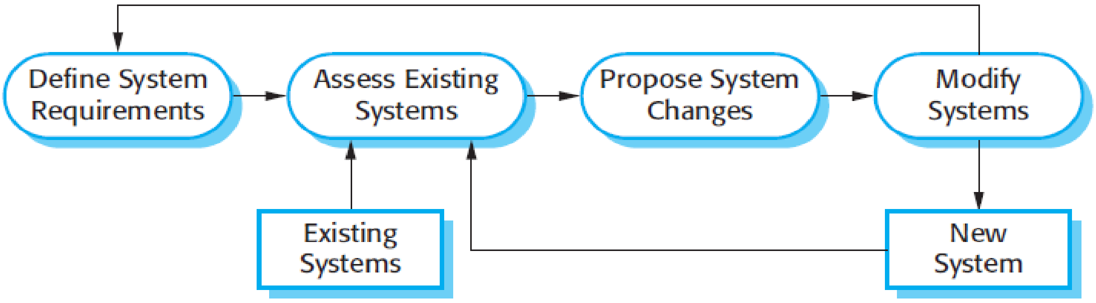

Requirements Specification
System Specification
System Design
Detailed Design
Acceptance Test Plan
System Integration Test Plan
Sub-System Integration Test Plan
Module and Unit Code and Test
Service
Acceptance Test
System Integration Test
Sub-System Integration Test

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.

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.



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.

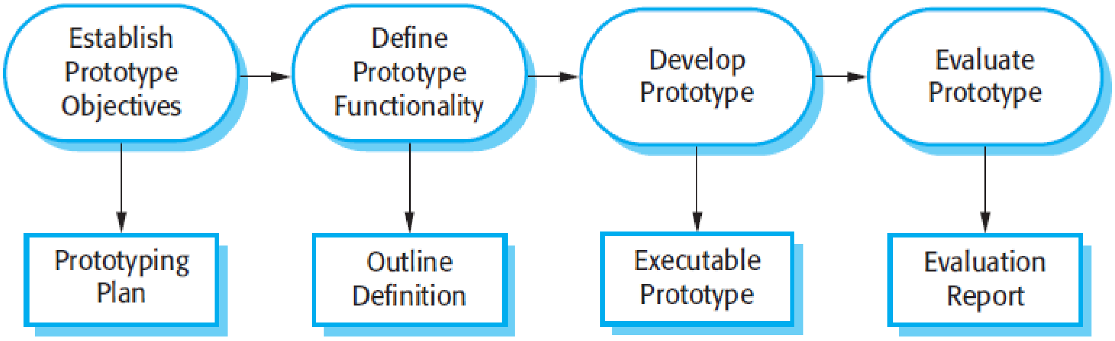
In this section, I discuss two ways of coping with change and changing system requirements. These are:
1. System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions. This supports change 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.

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 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.

Incremental delivery

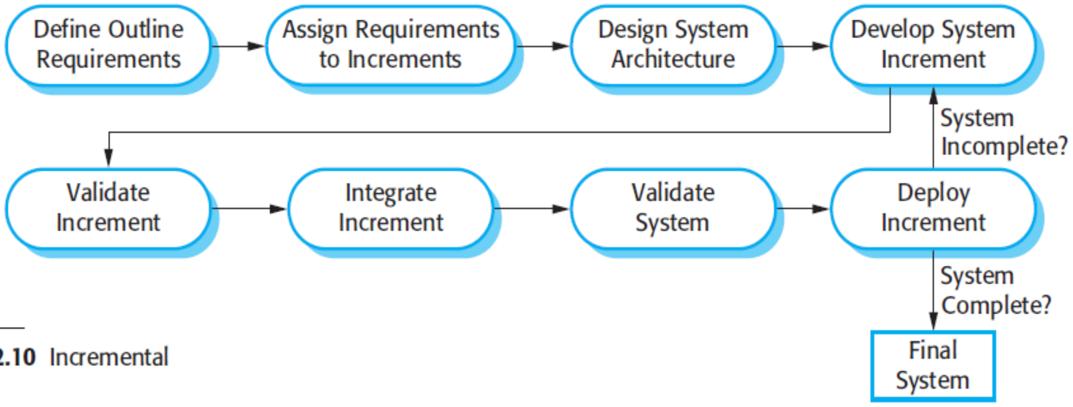


Figure 2.10 Incremental delivery

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.

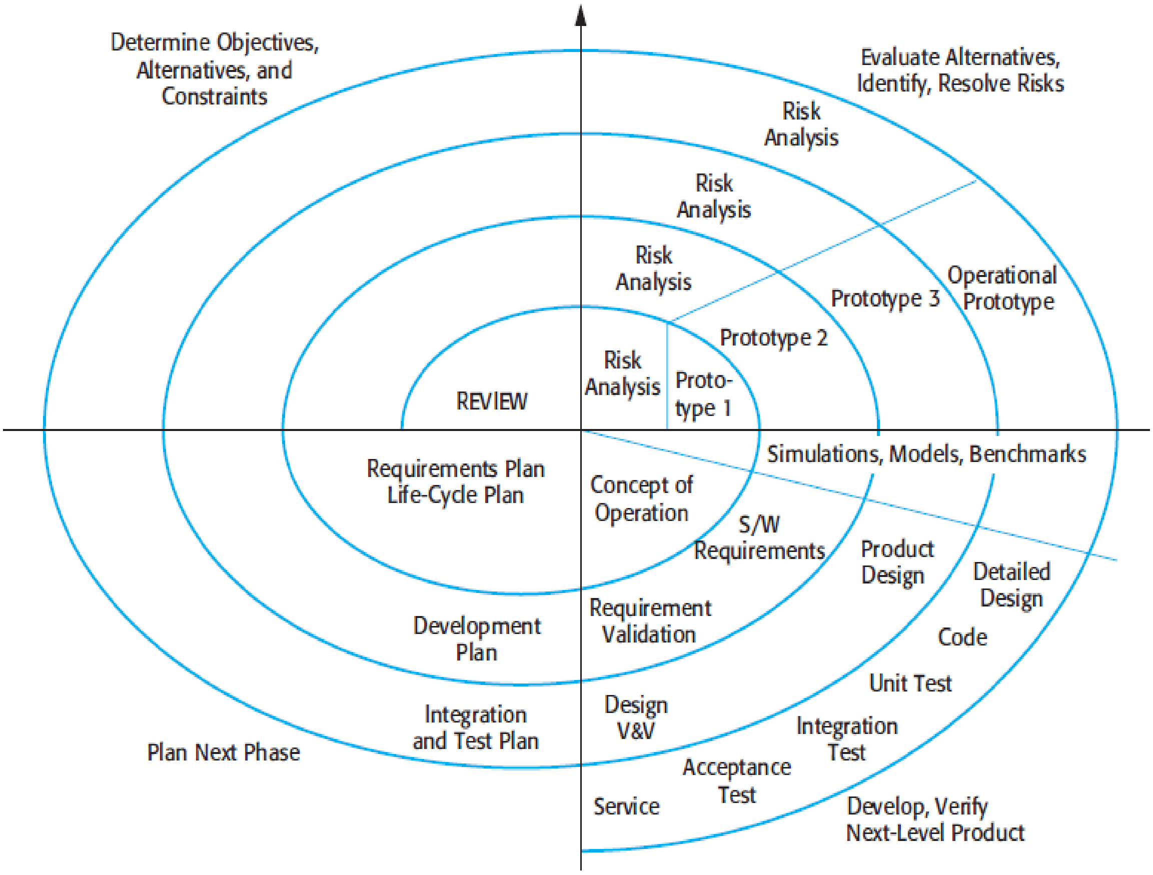
However, there are problems with incremental delivery:

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.

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 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 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).

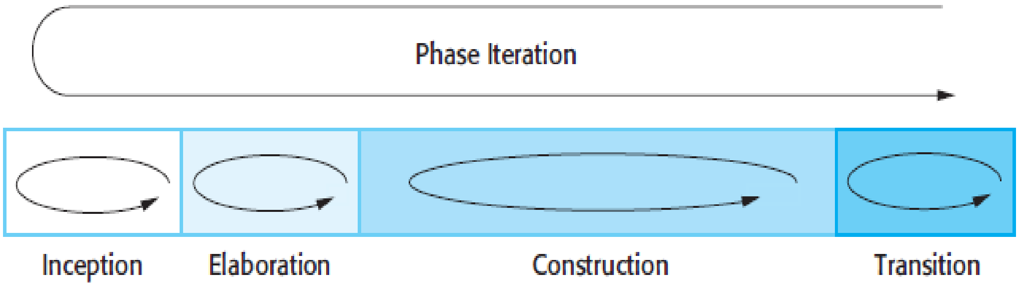
The RUP recognizes that conventional process models present a single view of the process. In contrast, the RUP is normally described from three perspectives:

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 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.

Static workflows in the Rational Unified Process

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.

. 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.

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.

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?

.