3 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 concept of a software process and a software process model;
- understand a number of different software process models and when they might be used;
- understand, in outline, process models for software requirements engineering, software development, testing and evolution;
- have been introduced to CASE technology for software process support.

Contents

As I suggested in Chapter 1, a software process is a set of activities and associated results which lead to the production of a software product. These may involve the development of software from scratch although it is increasingly the case that new software is developed by extending and modifying existing systems.

Software processes are complex and, like all intellectual processes, are reliant on human judgement. Because of the need for judgement and creativity, attempts to automate software processes have met with limited success. CASE tools (discussed in section 3.7) can support some process activities but there is no possibility, at least in the next few years, of more extensive automation where software takes over creative design from the engineers involved in the software process.

One reason why there is limited scope for process automation is the immense diversity of software processes. There is no ideal process and different organisations have developed completely different approaches to software development. Processes have evolved to exploit the capabilities of the people in an organisation and the specific characteristics of the systems which are being developed. Therefore, even within the same company, there may be many different processes used for software development.

Although there are many different software processes, there are fundamental activities which are common to all software processes. These are:

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.

I present an overview of these activities in this chapter and discuss them in much more detail in later parts of the book.

Although there is no 'ideal' software process, there is a lot of scope for improving the software process in many organisations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering. Indeed, many organisations still rely on *ad hoc* processes and do not take advantage of software engineering methods in their software development.

Software process improvement can be implemented in a number of different ways. It may come about through process standardisation where the diversity in software processes in an organisation is reduced. This leads to improved communication, reduction in training time and makes automated process support more economic. Standardisation is also an essential first step in introducing new software engineering methods and techniques and good software engineering practice. I return to the topic of software process improvement in Chapter 25.

3.1 Software process models

As discussed in Chapter 1, a software process model is an abstract representation of a software process. Each process model represents a process from a particular perspective so only provides partial information about that process. 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 useful abstractions which can be used to explain different approaches to software development. For many large systems, of course, there is no single software process that is used. Different processes are used to develop different parts of the system.

The process models that I discuss in this chapter 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.
2. *Evolutionary development* This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from abstract specifications. This is then refined with customer input to produce a system which satisfies the customer's needs.
3. *Formal systems development* This approach is based on producing a formal mathematical system specification and transforming this specification, using mathematical methods, to construct a program. Verification of system components is carried out by making mathematical arguments that they conform to their specification.
4. *Reuse-based development* 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.

Processes based on the waterfall model and evolutionary development are widely used for practical systems development. Formal system development has been successfully used in a number of projects (Mills *et al.*, 1987; Linger, 1994) but processes based on this model are still only used in a few organisations. Informal reuse is common in many processes but most organisations do not explicitly orient their software development processes around reuse. However, this approach is likely to be very influential in the 21st century as assembling systems from reusable components is essential for rapid software development. I discuss software reuse in Chapter 14.

3.1.1 The 'waterfall' model

The first published model of the software development process was derived from other engineering processes (Royce, 1970). This is illustrated in Figure 3.1. Because of the cascade from one phase to another, this model is known as the 'waterfall model' or software life cycle. The principal stages of the model map onto 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 partitions the requirements to either hardware or software systems. It establishes 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 realised 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.

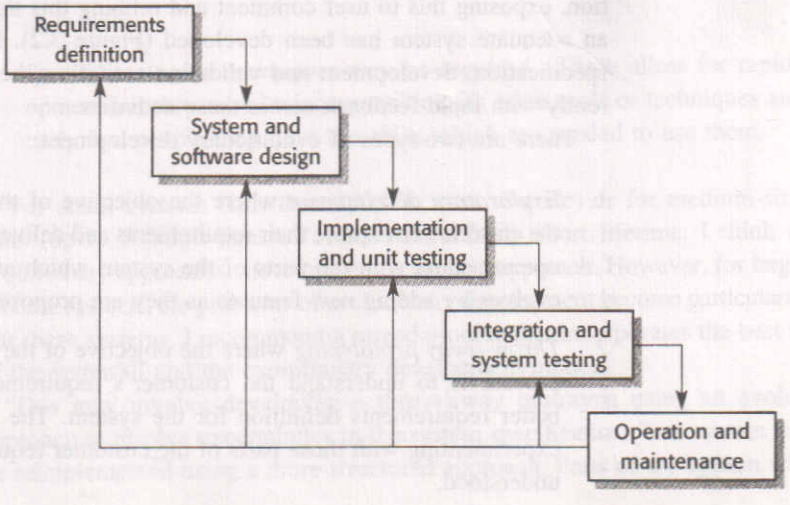


Figure 3.1 The software life cycle

In principle, the result of each phase is one or more documents which 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 a sequence of iterations of the development activities.

Because of the costs of producing and approving documents, iterations are 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 are 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 some or all previous process stages.

The problem with the waterfall model is its inflexible partitioning of the project into these distinct stages. Commitments must be made at an early stage in the process and this means that it is difficult to respond to changing customer requirements. Therefore, the waterfall model should only be used when the requirements are well understood. However, the waterfall model reflects engineering practice. Consequently, software processes based on this approach are still used for software development, particularly when this is part of a larger systems engineering project.

3.1.2 Evolutionary development

Evolutionary development is based on the idea of developing an initial implementation, exposing this to user comment and refining this through many versions until an adequate system has been developed (Figure 3.2). Rather than have separate specification, development and validation activities, these are carried out concurrently with rapid feedback across these activities.

There are two types of evolutionary development:

1. *Exploratory development* where the objective of the process is to work with the customer to explore their requirements and deliver a final system. The development starts with the parts of the system which are understood. The system evolves by adding new features as they are proposed by the customer.
2. *Throw-away prototyping* where the objective of the evolutionary development process is to understand the customer's requirements and hence develop a better requirements definition for the system. The prototype concentrates on experimenting with those parts of the customer requirements which are poorly understood.

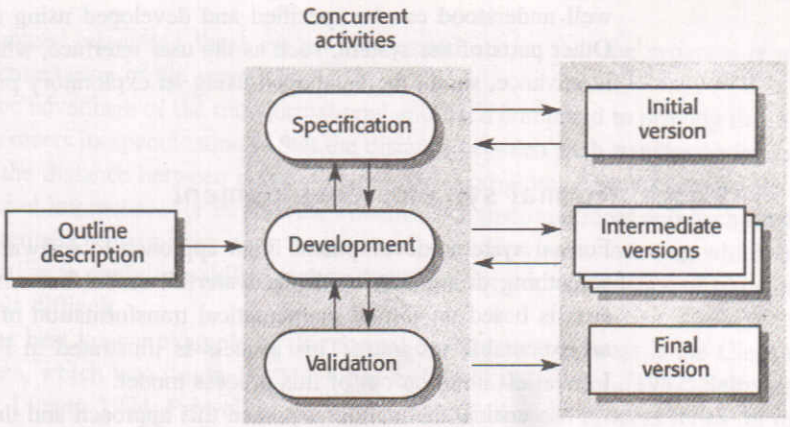


Figure 3.2 Evolutionary development

I cover evolutionary development processes and process support in Chapter 8 where various prototyping techniques are described.

An evolutionary approach to software development is often more effective than the waterfall approach in producing systems which meet the immediate needs of customers. The advantage of a software process which is based on an evolutionary approach is that the specification can be developed incrementally. As users develop a better understanding of their problem, this can be reflected in the software system. However, from an engineering and management perspective, it has three 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 which reflect every version of the system.

2. *Systems are often poorly structured* Continual change tends to corrupt the software structure. Incorporating software changes becomes increasingly difficult and costly.

3. *Special tools and techniques may be required* These allow for rapid development but they may be incompatible with other tools or techniques and relatively few people may have the skills which are needed to use them.

★ For small systems (less than 100,000 lines of code) or for medium-sized systems (up to 500,000 lines of code) with a fairly short lifetime, I think that the evolutionary approach to development is the best approach. However, for large, long-lifetime systems, the problems of evolutionary development become particularly acute. For these systems, I recommend a mixed process that incorporates the best features of the waterfall and the evolutionary development models.

This may involve developing a throw-away prototype using an evolutionary approach to resolve uncertainties in the system specification. This system may then be reimplemented using a more structured approach. Parts of the system which are

well understood can be specified and developed using a waterfall-based process. Other parts of the system, such as the user interface, which are difficult to specify in advance, should be developed using an exploratory programming approach.

3.1.3 Formal systems development

Formal systems development is an approach to software development which has something in common with the waterfall model but where the development process is based on formal mathematical transformation of a system specification to an executable program. This process is illustrated in Figure 3.3. For simplicity, I have left iteration out of this process model.

The critical distinctions between this approach and the waterfall model are:

1. The software requirements specification is refined into a detailed formal specification which is expressed in a mathematical notation.
2. The development processes of design, implementation and unit testing are replaced by a transformational development process where the formal specification is refined, through a series of transformations, into a program. This refinement process is illustrated in Figure 3.4.

In the transformation process, the formal mathematical representation of the system is systematically converted into a more detailed, but still mathematically correct, system representation. Each step adds detail until the formal specification is converted into an equivalent program. Transformations are sufficiently close that the effort of verifying the transformation is not excessive. It can therefore be

Figure 3.3 Formal systems development

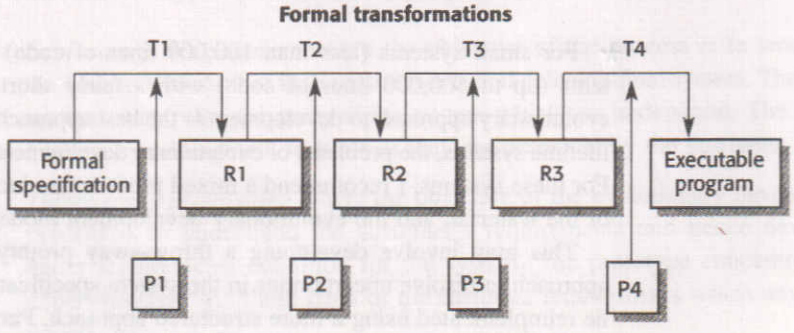


Figure 3.4 Formal transformation

guaranteed, assuming there are no verification errors, that the program is a true implementation of the specification.

★ The advantage of the transformational approach compared to proving that a program meets its specification is that the distance between each transformation is less than the distance between a specification and a program. Program proofs are very long and impractical for large-scale systems. A transformational approach made up of a sequence of smaller steps is more tractable. However, choosing which transformation to apply is a skilled task and proving the correspondence of transformations is difficult.

The best known example of this formal development process is the Cleanroom process, which was originally developed by IBM (Mills *et al.*, 1987; Selby *et al.*, 1987; Linger, 1994; Prowell *et al.*, 1999). The Cleanroom process relies on incremental development of the software and each stage is developed and its correctness demonstrated using a formal approach. There is no testing for defects in the process and the system testing is focused on assessing the system's reliability. I discuss this process in Chapter 19.

Both the Cleanroom approach and another approach to formal development based on the B method (Wordsworth, 1996) have been applied successfully. There were few defects in the delivered system and the development costs were not significantly different from the costs of other approaches. This approach is 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 which demonstrates to customers or certification bodies that the system does actually meet the safety or security requirements.

Outside of these specialised domains, processes based on formal transformations are not widely used. They require specialised expertise and, in reality, for the majority of systems this process does not offer significant cost or quality advantages over other approaches. The main reason for this is that system interaction is not amenable to formal specification and this takes up a very large part of the development effort for most software systems.

3.1.4 Reuse-oriented development

In the majority of software projects, there is some software reuse. This usually happens informally when people working on the project know of designs or code which is similar to that required. They look for these, modify them as required and incorporate them into their system. In the evolutionary approach, described in section 3.1.2, reuse is often seen as essential for rapid system development.

This informal reuse takes place irrespective of the generic process which is used. However, in the past few years, an approach to software development (component-based software engineering) which relies on reuse has emerged and is becoming increasingly widely used.

This reuse-oriented approach relies on a large base of reusable software components which can be accessed and some integrating framework for these components.

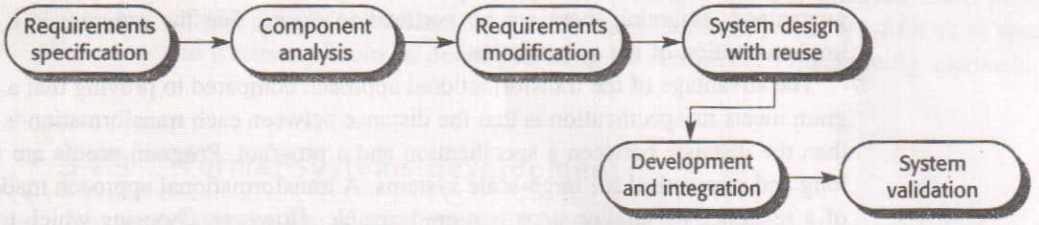


Figure 3.5 Reuse-oriented development

Sometimes, these components are systems in their own right (COTS or Commercial Off-The-Shelf systems) that may be used to provide specific functionality such as text formatting, numeric calculation, etc. The generic process model for reuse-oriented development is shown in Figure 3.5.

While the initial requirements specification stage and the validation stage are comparable with other 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 not an exact match and the components which may be used provide only some of the functionality required.
2. *Requirements modification* During this stage, the requirements are analysed using information about the components which 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 which are reused and organise 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 which cannot be bought in is developed and the components and COTS systems are integrated to create the system. System integration, in this model, may be part of the development process rather than a separate activity.

The reuse-oriented model has the obvious advantage that it reduces the amount of software to be developed and so reduces 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 which 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 organisation using these components.

3.2 Process iteration

All of the above process models have advantages and disadvantages. For most large systems, there is a need to use different approaches for different parts of the system, so a hybrid model must be used. Furthermore, there is also a need to support process iteration where parts of the process are repeated as system requirements evolve. The system design and implementation work must be reworked to implement the changed requirements.

In this section, I discuss two hybrid models which support different approaches to development and which have been explicitly designed to support process iteration. These are:

1. incremental development where the software specification, design and implementation is broken down into a series of increments which are developed in turn;
2. spiral development where the development of the system spirals outwards from an initial outline through to the final developed system.

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 organisations where the complete system specification is part of the contract for the system development. In the incremental approach, there is not a compete 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.

3.2.1 Incremental development

The waterfall model of development requires customers for a system to commit to a set of requirements before design begins and the designer to commit to particular design strategies before implementation. Changes to the requirements during development require rework of the requirements, design and implementation. However, the advantages of the waterfall model are that it is a simple management model and its separation of design and implementation should lead to robust systems which are amenable to change.

By contrast, an evolutionary approach to development allows requirements and design decisions to be delayed but also leads to software which may be poorly structured and difficult to understand and maintain. Incremental development is an in-between approach which combines the advantages of both of these models.

The incremental approach to development (Figure 3.6) was suggested by Mills (Mills *et al.*, 1980) as a means of reducing rework in the development process and giving customers some opportunities to delay decisions on their detailed requirements until they had some experience with the system.

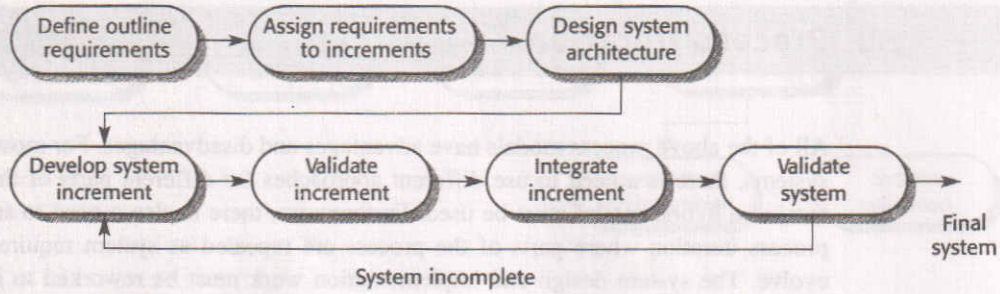


Figure 3.6 Incremental development

In an incremental development 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 subset of the system functionality. The allocation of services to increments depends on the service priority. The highest priority services are delivered first to the customer

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 using the most appropriate development process. During that 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 which helps them clarify their requirements for later increments and for later versions of the current increment. As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment. The common services may be implemented early in the process or may be implemented incrementally as functionality is required by an increment.

There is no need to use the same process for the development of each increment. Where the services in an increment have a well-defined specification, a waterfall model of development may be used for that increment. Where the specification is unclear, an evolutionary development model may be used.

This incremental development process has a number of advantages:

1. Customers do not have to wait until the entire system is delivered until they can gain value from it. The first increment satisfies their most critical requirements so the software can be immediately used.
2. Customers can use the early increments as a form of prototype and gain experience which informs the requirements for later system increments.
3. There is a lower risk of overall project failure. Although problems may be encountered in some increments, it is likely that some will be successfully delivered to the customer.

4. As the highest priority services are delivered first and later increments are integrated with them, it is inevitable that 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 some problems with incremental development. Increments should be relatively small (no more than 20,000 lines of code) and each increment should deliver some system functionality. It may therefore be difficult to map the customer's requirements onto increments of the right size. Furthermore, most systems require a set of basic facilities which are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it is difficult to identify common facilities that all increments require.

A recent evolution of this incremental approach called 'extreme programming' has been developed (Beck, 1999). This is based around the development and delivery of very small increments of functionality, customer involvement in the process, constant code improvement and egoless programming as discussed in Chapter 23. Beck's article includes several reports of the success of this approach but it is too early to say if this will become a mainstream approach to software development.

3.2.2 Spiral development

The spiral model of the software process (Figure 3.7) that was originally proposed by Boehm (1988) is now widely known. Rather than represent the software process as a sequence of activities with some backtracking from one activity to another, the process is represented as a spiral. 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 system requirements definition, the next loop with system design and so on.

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 then chosen. For example, if user interface risks are dominant, an appropriate development model might be evolutionary prototyping. If safety risks are the main consideration, development based on formal transformations may be the most appropriate and so on. The waterfall model may be the

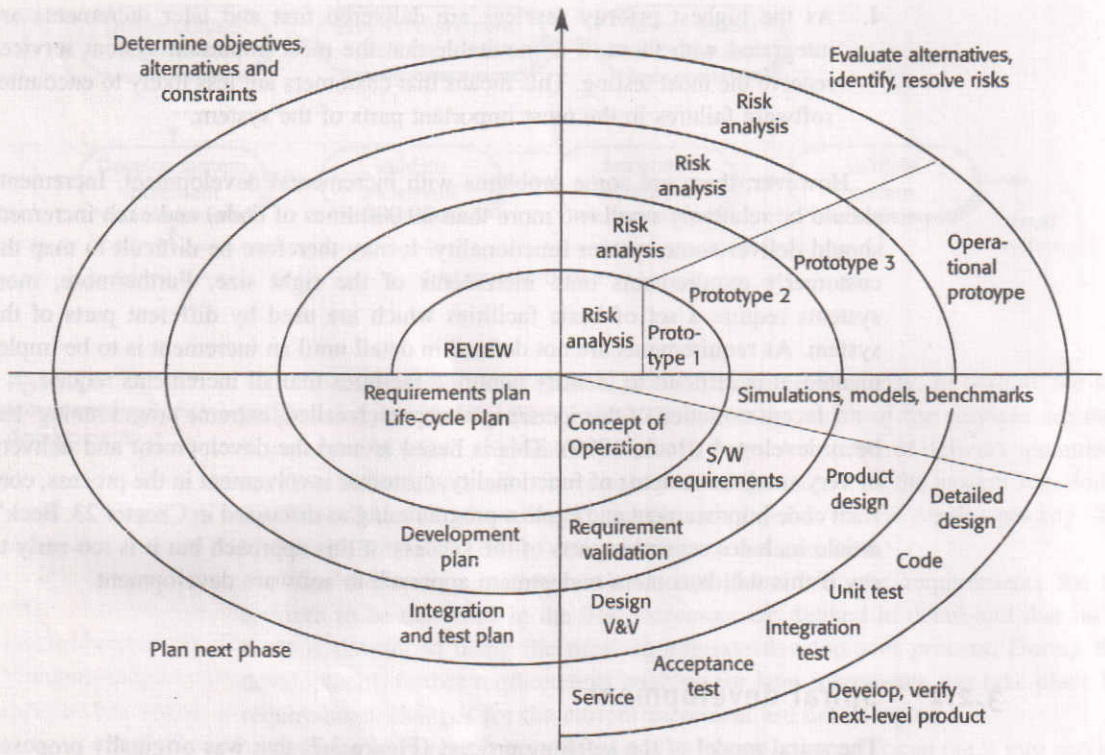


Figure 3.7 Boehm's spiral model of the software process (© 1988 IEEE)

most appropriate development model if the main identified risk is sub-system integration.

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 important distinction between the spiral model and other software process models is the explicit consideration of risk in the spiral model. Informally, risk is simply something which 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 result in project problems such as schedule and cost overrun, so risk minimisation is a very important project management activity. In Chapter 4, which covers project management, I have included a detailed discussion of risk and risk management.

A cycle of the spiral begins by elaborating objectives such as performance, functionality, etc. Alternative ways of achieving these objectives and the constraints imposed on each of these alternatives are then enumerated. Each alternative is assessed against each objective. This usually results in the identification of sources of project risk.

The next step is to evaluate these risks by activities such as more detailed analysis, prototyping, simulation, etc. Once risks have been assessed, some development is carried out and this is followed by a planning activity for the next phase of the process.

There are no fixed phases such as specification or design in the spiral model. The spiral model encompasses other process models. Prototyping may be used in one spiral to resolve requirements uncertainties and hence reduce risk. This may be followed by a conventional waterfall development. Formal transformation may be used to develop those parts of the system with high security requirements.

3.3 Software specification

In this section and in the following three sections I discuss the basic activities of software specification, development, validation and evolution. The first of these activities, software specification, is intended to establish what services are required from the system and the constraints on the system's operation and development. This activity is now often called requirements engineering. Requirements engineering is a 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 is shown in Figure 3.8. This process leads to the production of a requirements document which is the specification for the system. Requirements are usually presented at two levels of detail in this document. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

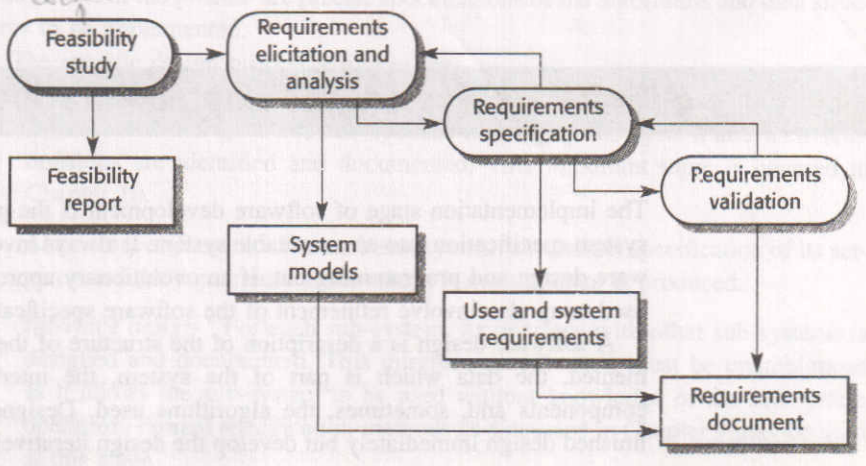


Figure 3.8
The requirements engineering process

There are four main phases 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 will decide if the proposed system will be cost-effective from a business point of view and if it can be developed given existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether 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, etc. This may involve the development of one or more different system models and prototypes. These help the analyst 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 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.

3.4 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 evolutionary 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 which is part of 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 through a number of

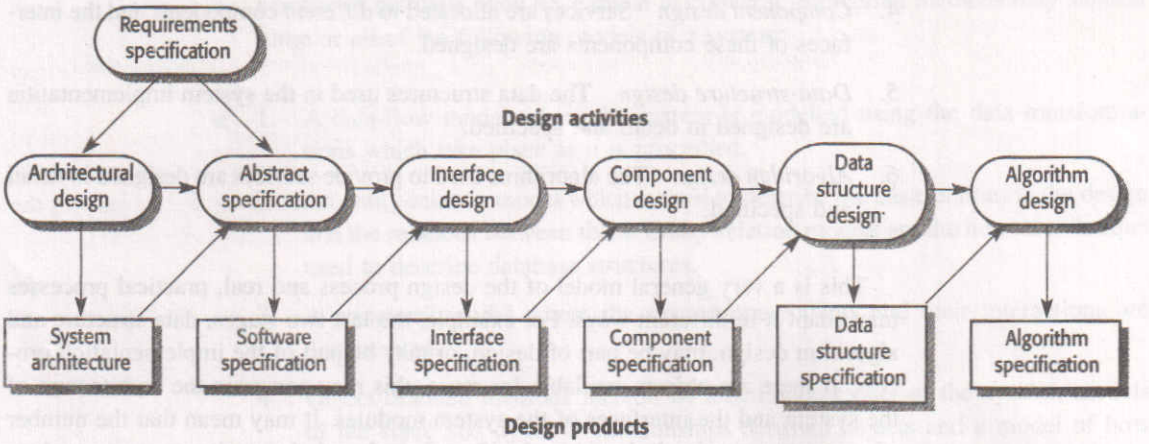


Figure 3.9 A general model of the design process

different versions. The design process involves adding formality and detail as the design is developed, with constant backtracking to correct earlier designs.

The design process may involve developing several models of the system at different levels of abstraction. As a design is decomposed, errors and omissions in earlier stages are discovered. These feed back to allow earlier design models to be improved. Figure 3.9 is a model of this process which shows the design descriptions produced at different stages of design. This 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.

A specification for the next stage is the output of each design activity. This specification may be an abstract, formal specification that is produced to clarify the requirements or it may be a specification of how part of the system is to be realised. As the design process continues, these specifications become more detailed. The final results of the process are precise specifications of the algorithms and data structures to be implemented.

The specific design process activities are:

1. *Architectural design* The sub-systems making up the system and their relationships are identified and documented. This important topic is covered in Chapter 10.
2. *Abstract specification* For each sub-system, an abstract specification of its services and the constraints under which it must operate is produced.
3. *Interface design* For each sub-system, its interface with other sub-systems is designed and documented. This interface specification must be unambiguous as it allows the sub-system to be used without knowledge of the sub-system operation. Formal specification methods as discussed in Chapter 9 may be used at this stage.

4. *Component design* Services are allocated to different components and the interfaces of these components are designed.
5. *Data structure design* The data structures used in the system implementation are designed in detail and specified.
6. *Algorithm design* The algorithms used to provide services are designed in detail and specified.

This is a very general model of the design process and real, practical processes may adapt it in different ways. For example, the last two stages, data structure and algorithm design, may be part of design, or may be part of the implementation process. If there are objects available for reuse, this may constrain the architecture of the system and the interfaces of the system modules. It may mean that the number of components to be designed is significantly reduced. An exploratory approach to design may be used and the system interfaces may be designed after the data structures have been specified.

3.4.1 Design methods

In many software development projects, software design is still an *ad hoc* process. Starting from a set of requirements, usually in natural language, an informal design is prepared. Coding commences and the design is modified as the system is implemented. There is little or no formal change control or design management. When the implementation stage is complete, the design has usually changed so much from its initial specification that the original design document is an incorrect and incomplete description of the system.

A more methodical approach to software design is proposed by 'structured methods' which are sets of notations and guidelines for software design. Examples of structured methods include Structured Design (Constantine and Yourdon, 1979), Structured Systems Analysis (Gane and Sarson, 1979), Jackson System Development (Jackson, 1983) and various approaches to object-oriented design (Robinson, 1992; Booch, 1994; Rumbaugh *et al.*, 1991; Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a, 1999b).

The use of structured methods normally involves producing graphical system models and results in large amounts of design documentation. CASE tools (see section 3.7) have been developed to support particular methods. Structured methods have been applied successfully in many large projects. They can deliver significant cost reductions because they use standard notations and ensure that standard design documentation is produced. No one method is demonstrably better or worse than other methods. The success or otherwise of methods often depends on their suitability for an application domain.

A structured method includes a design process model, notations to represent the design, report formats, rules and design guidelines. Although there are a large

number of methods, they have much in common. Structured methods may support some or all of the following models of a system:

1. A data-flow model where the system is modelled using the data transformations which take place as it is processed.
2. An entity-relation model which is used to describe the basic entities in the design and the relations between them. Entity-relation models are the normal technique used to describe database structures.
3. A structural model where the system components and their interactions are documented.
4. Object-oriented methods include an inheritance model of the system, models of the static and dynamic relationships between objects and a model of how objects interact with each other when the system is executing.

Particular methods supplement these with other system models such as state transition diagrams, entity life histories that show how each entity is transformed as it is processed and so on. Most methods suggest that a centralised repository for system information or a data dictionary should be used. I discuss a number of these approaches to system modelling in Chapter 7.

In practice, the guidance given by the methods is informal, so different designers will develop different designs. These 'methods' are really standard notations and embodiments of good practice. By following these methods and applying the guidelines, a reasonable design should emerge. Designer creativity is still required to decide on the system decomposition and to ensure that the design adequately captures the system specification. Empirical studies of designers (Bansler and Bødker, 1993) have shown that they rarely follow methods slavishly. They pick and choose from the guidelines depending on local circumstances.

3.4.2 Programming and debugging

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 designed in detail before any implementation begins, it is more normal for the later stages of design and program development to be interleaved. CASE 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 which is usually followed. Some programmers will start with components that they understand, develop these and then move on to less well-understood components. Others will take the opposite approach, leaving familiar components till last because they know

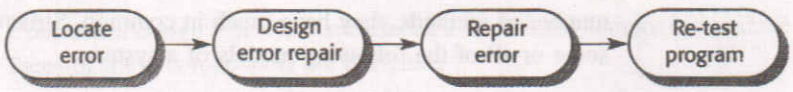


Figure 3.10 The debugging process

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.

Figure 3.10 illustrates a possible debugging process. Defects in the code must be located and the program modified to meet its requirements. Testing must then be repeated to ensure that the change has been made correctly. Thus the debugging process is part of both software development and software testing.

The debugger must generate hypotheses about the observable behaviour of the program, then test these hypotheses in the hope of finding the fault which caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localise the problem. Interactive debugging tools which show the intermediate values of program variables and a trace of the statements executed may be used to help the debugging process.

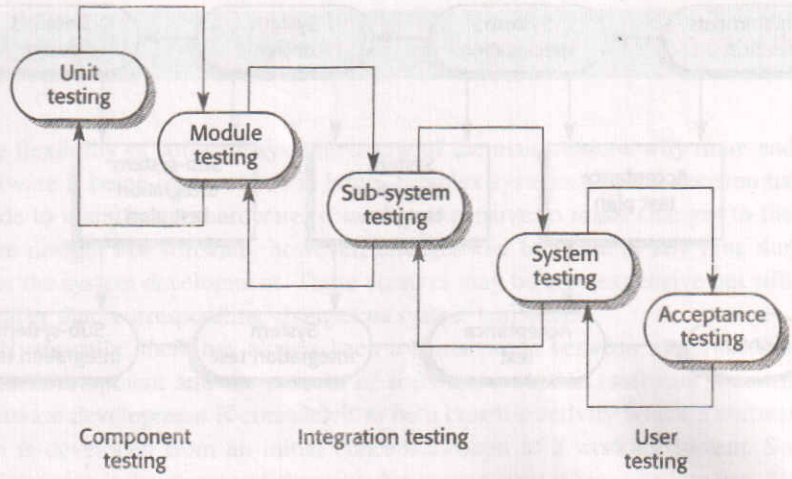
3.5 Software validation

Software validation or, more generally, verification and validation (V & V) is intended to show that a system conforms to its specification and that the system meets the expectations of the customer buying the system. It involves checking processes, such as inspections and reviews (see Chapter 19), at each stage of the software process from user requirements definition to program development. The majority of validation costs, however, are incurred after implementation when the operational system is tested (Chapter 20).

Except for small programs, systems should not be tested as a single, monolithic unit. Large systems are built out of sub-systems which are built out of modules which are composed of procedures and functions. The testing process should therefore proceed in stages where testing is carried out incrementally in conjunction with system implementation.

Figure 3.11 shows a five-stage testing process where system components are tested, the integrated system is tested and, finally, the system is tested with the customer's

Figure 3.11
The testing process



data. Ideally, component defects are discovered early in the process and interface problems 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 integration 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. *Unit testing* Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components.

2. *Module testing* A module is a collection of dependent components such as an object class, an abstract data type or some looser collection of procedures and functions. A module encapsulates related components, so can be tested without other system modules.

3. *Sub-system testing* This phase involves testing collections of modules which have been integrated into sub-systems. The most common problems which arise in large software systems are interface mismatches. The sub-system test process should therefore concentrate on the detection of module interface errors by rigorously exercising these interfaces.

4. *System testing* The sub-systems are integrated to make up the system. This process is concerned with finding errors that result from unanticipated interactions between sub-systems and sub-system interface problems. It is also concerned with validating that the system meets its functional and non-functional requirements and testing the emergent system properties.

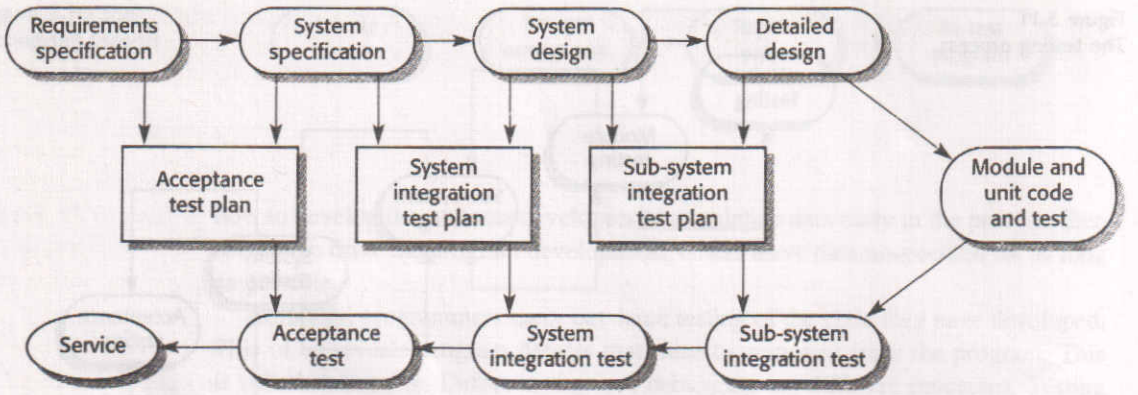


Figure 3.12 Testing phases in the software process

5. *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 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.

Unit testing and module testing are usually the responsibility of the programmers developing the component. 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 best and is therefore the best person to generate test data. Unit testing is part of the implementation process and it is expected that a component conforming to its specification will be delivered as part of that process.

Later stages of testing involve integrating work from a number of programmers and must be planned in advance. An independent team of testers should work from pre-formulated test plans which are developed from the system specification and design. Figure 3.12 illustrates how test plans are the link between testing and development activities.

Acceptance testing is sometimes called *alpha testing*. Bespoke 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 system 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 which 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.

3.6 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. For software, however, changes can be made at any time during or after the system development. These changes may be very expensive but still much cheaper than corresponding changes to system hardware.

Historically, there has always been a demarcation between the process of software development and the process of software evolution (software maintenance). Software development is considered to be a creative activity where a software system is developed from an initial concept through to a working system. Software maintenance is the process of changing that system once it has gone into use. Although the costs of 'maintenance' are often several times the initial development costs, maintenance processes are considered to be less challenging than original software development.

This demarcation is becoming increasingly irrelevant. Few software systems are now 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 where software is continually changed over its lifetime in response to changing requirements and customer needs. This evolutionary process is illustrated in Figure 3.13.

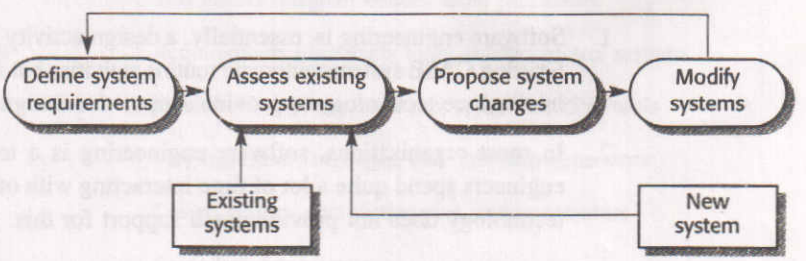


Figure 3.13 System evolution

3.7 Automated process support

Computer-aided Software Engineering (CASE) is the name given to software that is used to support software process activities such as requirements engineering, design,

program development and testing. CASE tools therefore include design editors, data dictionaries, compilers, debuggers, system building tools, etc.

CASE technology provides software process support by automating some process activities and by providing information about the software which is being developed. Examples of activities which can be automated using CASE include:

1. the development of graphical system models as part of the requirements specification or the software design;
2. understanding a design using a data dictionary which holds information about the entities and relations in a design;
3. the generation of user interfaces from a graphical interface description which is created interactively by the user;
4. program debugging through the provision of information about an executing program;
5. the automated translation of programs from an old version of a programming language such as COBOL to a more recent version.

CASE technology is now available for most routine activities in the software process. This has led to some improvements in software quality and productivity although these have been less than predicted by early advocates of CASE. They suggested that orders of magnitude improvement were likely if integrated CASE environments were used. In fact, the actual improvements which have been achieved are of the order of 40 per cent (Huff, 1992). Although this is significant, CASE has not revolutionised software engineering as was once predicted.

The improvements from the use of CASE are limited by two factors:

1. Software engineering is, essentially, a design activity based on creative thought. Existing CASE systems automate routine activities but attempts to harness artificial intelligence technology to provide support for design have not been successful.
2. In most organisations, software engineering is a team activity and software engineers spend quite a lot of time interacting with other team members. CASE technology does not provide much support for this.

Whether this situation will change in future is currently unclear. My view is that specific CASE products to support software design and software engineering teamwork are unlikely. However, generic design and cooperation support systems will be developed and adapted for use in the software process.

CASE technology is now mature and CASE tools and workbenches are available from a wide range of suppliers. However, rather than focus on any specific tools, I have simply presented an overview here with some discussion of specific support in related chapters of the book. In the book's web pages, I include links to more detailed material on CASE integration and links to CASE tool suppliers.

3.7.1 CASE classification

CASE classifications help us understand the different types of CASE tools and their role in supporting software process activities. There are various different ways of classifying CASE tools, each of which gives us a different perspective on these tools. In this section, I discuss CASE tools from three of these perspectives, namely:

1. *A functional perspective* where CASE tools are classified according to their specific function.
2. *A process perspective* where tools are classified according to the process activities which they support.
3. *An integration perspective* where CASE tools are classified according to how they are organised into integrated units which provide support for one or more process activities.

Figure 3.14 is a classification of CASE tools according to function. This table lists a number of different types of CASE tool and gives specific examples of each tool. This is not a complete list of CASE tools. Specialised tools such as tools to support reuse have not been included.

Figure 3.14 Functional classification of CASE tools

Tool type	Examples
Planning tools	PERT tools, estimation tools, spreadsheets
Editing tools	Text editors, diagram editors, word processors
Change management tools	Requirements traceability tools, change control systems
Configuration management tools	Version management systems, system building tools
Prototyping tools	Very high-level languages, user interface generators
Method-support tools	Design editors, data dictionaries, code generators
Language-processing tools	Compilers, interpreters
Program analysis tools	Cross-reference generators, static analysers, dynamic analysers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Re-engineering tools	Cross-reference systems, program re-structuring systems

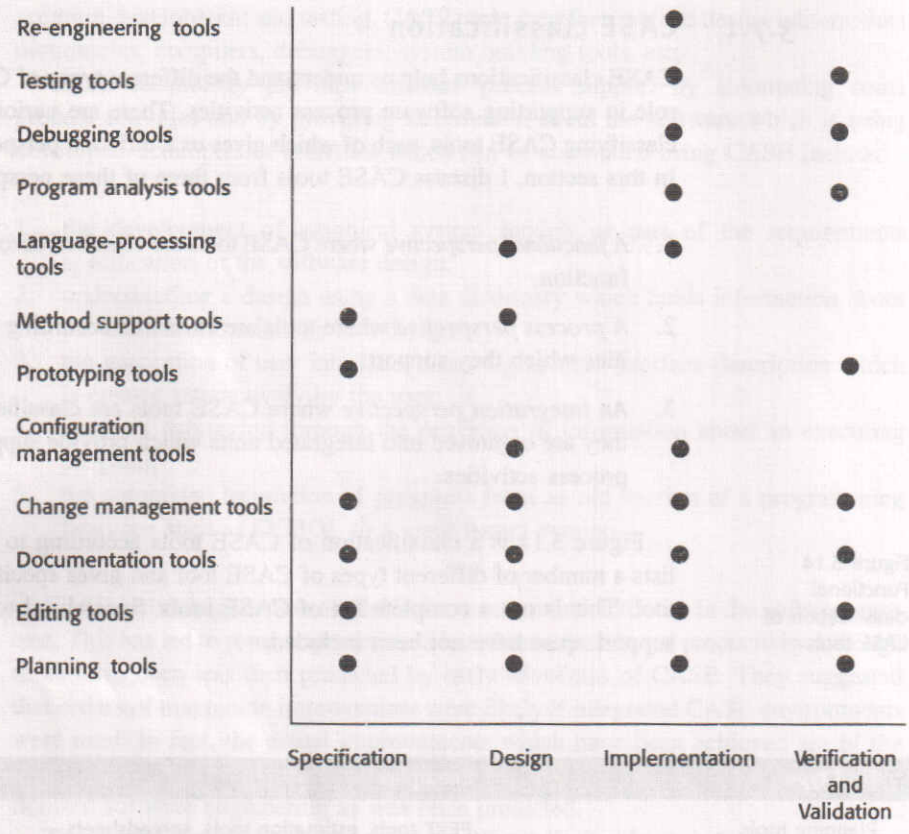


Figure 3.15 Activity-based classification of CASE tools

Figure 3.15 presents an alternative classification of CASE tools. It shows the process phases supported by a number of different types of CASE tool. Tools for planning and estimating, text editing, document preparation and configuration management may be used throughout the software process.

The breadth of support for the software process offered by CASE technology is another possible classification dimension. Fuggetta (1993) proposes that CASE systems should be classified into three categories:

1. *Tools* support individual process tasks such as checking the consistency of a design, compiling a program, comparing test results, etc. Tools may be general-purpose, stand-alone tools (e.g. a word-processor) or may be grouped into workbenches.
2. *Workbenches* support process phases or activities such as specification, design, etc. They normally consist of a set of tools with some greater or lesser degree of integration.
3. *Environments* support all or at least a substantial part of the software process. They normally include several different workbenches which are integrated in some way.

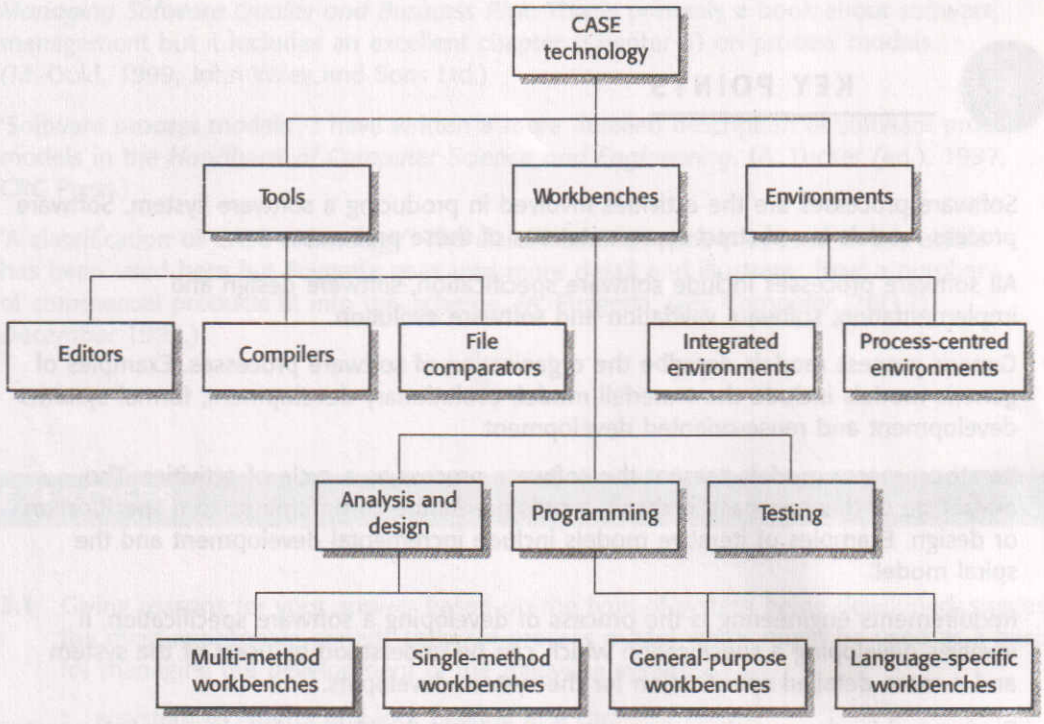


Figure 3.16 Tools, workbenches and environments

Figure 3.16 illustrates this classification and shows some examples of these different classes of CASE support. Of course, Figure 3.16 is simply an illustrative example; many types of tool and workbench have been left out of this diagram.

General-purpose tools are used at the discretion of the software engineer who makes decisions about when to apply them for process support. Workbenches, however, usually support some method which includes a process model and a set of rules/guidelines which apply to the software being developed. I have classified environments as integrated environments or process-centred environments. Integrated environments provide infrastructure support for data, control and presentation integration. Process-centred environments are more general. They include software process knowledge and a process engine which uses this process model to advise engineers on what tools or workbenches to apply and when they should be used.

In practice, the boundaries between these different classes are blurred. Tools may be sold as a single product but may embed support for different activities. For example, most word processors now provide a built-in diagram editor. CASE workbenches for design increasingly offer support for programming and testing so they are more akin to environments than specialised workbenches. It may therefore not always be easy to position a product using a classification. Nevertheless, it provides a useful first step to help understand the extent of process support which a tool is likely to provide.

KEY POINTS

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- All software processes include software specification, software design and implementation, software validation and software evolution.
- Generic process models describe the organisation of software processes. Examples of generic models include the waterfall model, evolutionary development, formal systems development and reuse-oriented development.
- Iterative process models present the software process as a cycle of activities. The advantage of this approach is that it avoids premature commitments to a specification or design. Examples of iterative models include incremental development and the spiral model.
- Requirements engineering is the process of developing a software specification. It involves developing a specification which can be understood by users of the system and a more detailed specification for 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 is concerned with modifying existing software systems to meet new requirements. This is becoming the normal approach to software development for small and medium-sized systems.
- CASE technology provides automated support for software processes. CASE tools support individual process activities; CASE workbenches support a set of related activities; CASE environments support all or most software process activities.

FURTHER READING

'Embracing change with extreme programming'. An interesting overview of this new incremental process that has been designed for use in conjunction with object-oriented development. (K. Beck, *IEEE Computer*, 32(10), October 1999.)

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, 1999, John Wiley and Sons Ltd.)

'Software process models'. I have written a more detailed description of software process models in the *Handbook of Computer Science and Engineering*. (A. Tucker (ed.), 1997, CRC Press.)

'A classification of CASE technology'. The classification scheme proposed in this article has been used here but Fuggetta goes into more detail and illustrates how a number of commercial products fit into this scheme. (A. Fuggetta, *IEEE Computer*, 26(12), December 1993.)

EXERCISES

3.1 Giving reasons for your answer based on the type of system being developed, suggest the most appropriate generic software process model which 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 system for railway passengers that finds train times from terminals installed in stations.

3.2 Explain why programs which are developed using evolutionary development are likely to be difficult to maintain.

3.3 Explain how both the waterfall model of the software process and the prototyping model can be accommodated in the spiral process model.

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

3.5 Describe the main activities in the software design process and the outputs of these activities. Using an entity-relation diagram, show possible relationships between the outputs of these activities.

3.6 What are the five components of a design method? Take any method which you know and describe its components. Assess the completeness of the method which you have chosen.

3.7 Design a process model for running system tests and recording their results.

3.8 Explain why a software system that is used in a real-world environment must change or become progressively less useful.

3.9 Suggest how a CASE technology classification scheme may be helpful to managers responsible for CASE system procurement.

3.10 Survey the tool availability in your local development environment and classify the tools according to the parameters (function, activity, breadth of support) suggested here.

3.11 Historically, the introduction of technology has caused profound changes in the labour market and, temporarily at least, displaced people from jobs. Discuss whether the introduction of CASE technology 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 resist the introduction of this technology passively and actively?