

**KANTIPUR ENGINEERING COLLEGE**

# Software Engineering

---

**CT 601**

**Compiled by: Dr. Sunil Chaudhary  
2/11/2018**

This is a compiled note for the Software Engineering courses as per the syllabus of Institute of Engineering, Nepal.

# SOFTWARE ENGINEERING

Software Engineering is an engineering discipline which is concerned with all aspects of software production from the early stages of system requirements through to maintaining the system after it has gone into use, for instance, requirement engineering, development, testing, implementation, documentation, and maintenance.

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.

*"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software"*—IEEE Standard Glossary of Software Engineering Terminology

*"An engineering discipline that is concerned with all aspects of software production"*— Ian Sommerville

# 1. SOFTWARE PROCESS AND REQUIREMENTS

## 1.1. Software Crisis

If software developers are asked to certify that the developed software is bug free, no software would have ever been released. Hence, “*software crisis*” has become a fixture of everyday life. Many well published failures have had not only major economic impact but also become the cause of death of many human beings. Some of the failures are, for example, Y2K problem and Ariane-5 space rocket.

The Standish Group (Industry analysis firm) classified software projects into three classes and the percentage of projects that fall into them are mentioned in Figure 1.1:

- **Successful:** All features within time and budget constraints
- **Challenged:** Late, over budget, and/or with reduced functionality
- **Cancelled** before completion or never used

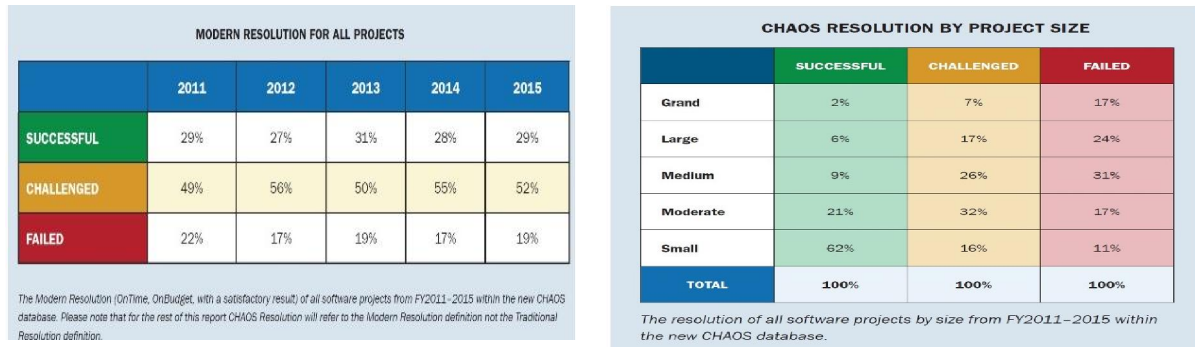


Figure 1.1: Chaos report [2]

Findings from some other similar studies are mentioned in Table 1.1.

| Studies  | Results  |
|--|--|
| El Emam and Koru [3]   | About 20% of delivered projects were unsuccessful, and about 30% were challenged.                                |
| Bull Survey 1998 [4]   | 75% Missed Deadlines, 55% Exceeded Budget and 37% Not Meeting Requirements                                       |
| KPMG Report 1999 (IT projects in 1450 leading public and private institutions in Canada) [5] | 39% Success Rate. Of the Failed Projects 87% Overran Schedule, 56% Over Budget and 45% Not Meeting Requirements. |
| Computer Weekly 2003 [6]   | 16% Success Rate   |
| Miller Report 2007 (163 IT projects in UK)[7]  | 44% Success Rate   |

Table 1.1: Software projects survey report

Next there is *Anti-Chaos* [8], who does not support the Chaos report primarily due to the following shortcomings of the Chaos report:

- Categories are not really well defined (is a fully functional project that is 10% late really challenged?)
- Sources are not disclosed (which projects were surveyed?)
- Methodology is not disclosed (what methodology was used for the study?)
- Reporting is sloppy (not clear if “189% cost overruns” means 89% over original estimate or 189% over original estimate)

The three main software crises are mentioned in Table 1.2:

|            | First Software Crisis   | Second Software Crisis   | Third Software Crisis  |
|------------|---|--|--|
| Time frame | 60s and 70s   | 80s and 90s  | 2010 to??  |
| Problem    | ->Assembly Language Programming<br>->Computers could handle larger more complex programs<br>->Needed to get <b>Abstraction</b> and <b>Portability</b> without losing <b>Performance</b> | ->Inability to build and maintain complex and robust applications requiring multi-million lines of code developed by hundreds of programmers<br>->Computers could handle larger more complex programs<br>->Needed to get <b>Composability</b> , <b>Malleability</b> and <b>Maintainability</b> | ->Sequential performance is left behind by <b>Moore's law</b>  |
| Solution   | ->High-level languages for von-Neumann machines, FORTRAN and C<br>->Provided "common machine language" for uniprocessors  | ->Object Oriented Programming ,e.g., C++, C# and Java<br>->Better tools, e.g., Component libraries, Purify<br>->Better software engineering methodology, e.g., Design patterns, specification, testing, code reviews   | ->Needed continuous and reasonable performance improvements to support new features, and larger datasets<br>-> While sustaining <b>portability</b> , <b>malleability</b> and <b>maintainability</b> without unduly increasing complexity faced by the programmer<br>->Critical to keep-up with the current rate of evolution in software |

Table 1.2: Software crises [9]

Software failures are as 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.
- **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.

## 1.2 Software Characteristics

According to I. Sommerville [1], the software characteristics are:

- **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.

### 1.3 Software Quality Attributes

Software quality is the degree to which software possesses a desired combination of attributes [10]. There are different schools/opinions/traditions concerning the properties of critical systems and the best methods to develop them [11]. According to ISO 9126 [12], the software quality attributes are (mentioned in detail in Table 1.3):

| The full table of Characteristics and Sub-characteristics for the ISO 9126-1 Quality Model is:- |                     |   |
|---|---------------------|---|
| Characteristics   | Sub-characteristics | Definitions   |
| <b>Functionality</b>  | Suitability         | This is the essential Functionality characteristic and refers to the appropriateness (to specification) of the functions of the software.   |
|   | Accurateness        | This refers to the correctness of the functions; an ATM may provide a cash dispensing function but is the amount correct?   |
|   | Interoperability    | A given software component or system does not typically function in isolation. This sub-characteristic concerns the ability of a software component to interact with other components or systems. |
|   | Compliance          | Where appropriate certain industry (or government) laws and guidelines need to be complied with, i.e. SOX. This sub-characteristic addresses the compliant capability of software.                |
|   | Security            | This sub-characteristic relates to unauthorized access to the software functions.   |
| <b>Reliability</b>  | Maturity            | This sub-characteristic concerns frequency of failure of the software.  |
|   | Fault tolerance     | The ability of software to withstand (and recover) from component, or environmental, failure.   |
|   | Recoverability      | Ability to bring back a failed system to full operation, including data and network connections.  |
| <b>Usability</b>  | Understandability   | Determines the ease of which the systems functions can be understood, relates to user mental models in Human Computer Interaction methods.  |
|   | Learnability        | Learning effort for different users, i.e. novice, expert, casual etc.   |
|   | Operability         | Ability of the software to be easily operated by a given user in a given environment.   |
| <b>Efficiency</b>   | Time behavior       | Characterizes response times for a given thru put, i.e. transaction rate.   |
|   | Resource behavior   | Characterizes resources used, i.e. memory, CPU, disk and network usage.   |
| <b>Maintainability</b>  | Analyzability       | Characterizes the ability to identify the root cause of a failure within the software.  |
|   | Changeability       | Characterizes the amount of effort to change a system.  |
|   | Stability           | Characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes.   |
|   | Testability         | Characterizes the effort needed to verify (test) a system change.   |
| <b>Portability</b>  | Adaptability        | Characterizes the ability of the system to change to new specifications or operating environments.  |
|   | Installability      | Characterizes the effort required to install the software.  |
|   | Conformance         | Similar to compliance for functionality, but this characteristic relates to portability. One example would be Open SQL conformance which relates to portability of database used.                 |
|   | Replaceability      | Characterizes the <i>plug and play</i> aspect of software components, that is how easy is it to exchange a given software component within a specified environment.                               |

Table 1.3: Software quality attributes [12]

- **Functionality:** Functionality is the essential purpose of any product or service. It refers to the product features or list of functions that the software is expected to perform. For software, the list

of functions can go on and on but the main point to note is that functionality is expressed as a totality of essential functions that the software product provides. It is also important to note that the presence or absence of these functions in a software product can be verified as either existing or not, in that it is a Boolean (either a yes or no answer).

- **Reliability:** Once a software system is functioning, as specified, and delivered the reliability characteristic defines the capability of the system to maintain its service provision under defined conditions for defined periods of time. One aspect of this characteristic is *fault tolerance* that is the ability of a system to withstand component failure. For example if the network goes down for 20 seconds then comes back the system should be able to recover and continue functioning.
- **Usability:** Usability only exists with regard to functionality and refers to the ease of use for a given function. For example a function of an ATM machine is to dispense cash as requested. Placing common amounts on the screen for selection, i.e. \$20.00, \$40.00, \$100.00 etc, does not impact the function of the ATM but addresses the Usability of the function. The ability to learn how to use a system (learnability) is also a major sub-characteristic of usability.
- **Efficiency:** This characteristic is concerned with the system resources used when providing the required functionality. The amount of disk space, memory, network etc. provides a good indication of this characteristic. As with a number of these characteristics, there are overlaps. For example the usability of a system is influenced by the system's Performance, in that if a system takes 3 hours to respond the system would not be easy to use although the essential issue is a performance or efficiency characteristic.
- **Maintainability:** The ability to identify and fix a fault within a software component is what the maintainability characteristic addresses. In other software quality models this characteristic is referenced as supportability. Maintainability is impacted by code readability or complexity as well as modularization. Anything that helps with identifying the cause of a fault and then fixing the fault is the concern of maintainability. Also the ability to verify (or test) a system, i.e. testability, is one of the sub-characteristics of maintainability.
- **Portability:** This characteristic refers to how well the software can adopt to changes in its environment or with its requirements. The sub-characteristics of this characteristic include adaptability. Object oriented design and implementation practices can contribute to the extent to which this characteristic is present in a given system.

## SOFTWARE QUALITY ATTRIBUTES TRADE-OFFS

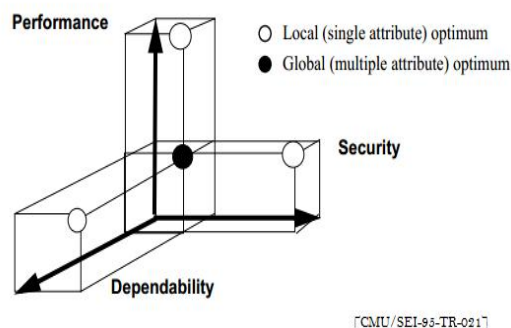


Figure 1.2: Software quality attributes trade-offs [11]

Calculating and understanding the value of a single overall metric for software quality may be more trouble than it is worth. The major problem is that many of the individual characteristics of quality are in conflict; added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness and conflict with legibility, as illustrated in Figure 1.2.

Designers need to analyze trade-offs between multiple conflicting attributes to satisfy user requirements. The ultimate goal is the ability to quantitatively evaluate and trade off multiple quality attributes to arrive at a better overall system. We should not look for a single, universal metric, but rather for quantification of individual attributes and for trade-off between these different metrics, starting with a description of the software architecture.

#### 1.4. Software Process Model

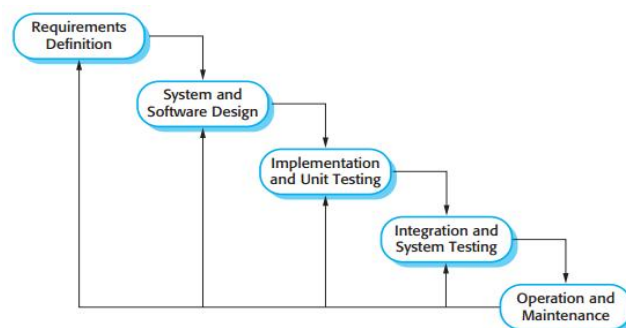
A software process is a sequence of activities that leads to the production of a software product. The four fundamental activities that is common to all software processes [1]:

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

There is no ideal process and most organizations have developed their own software development processes to take advantage of the capabilities of the people in an organization and the specific characteristics of the systems that are being developed.

A (software/system) process model is a description of the sequence of activities carried out in a software engineering project, and the relative order of these activities. For some systems, such as critical systems, a very structured development process is required. For business systems, with rapidly changing requirements, a less formal, flexible process is likely to be more effective.

#### THE WATERFALL MODEL



I. Sommerville, "Software Engineering", Ninth Edition, Pearson Education, Inc., 2011.

Figure 1.3: Waterfall model [1]

In principle, the waterfall model is a linear sequential life cycle model. It cascaded from one phase to another phase giving it name the “**waterfall model**” or software life cycle, as shown in Figure 1.3. This model is an example of a plan-driven process, that is, you must plan and schedule all of the process activities before starting work on them. Each phase must be completed before the next phase can begin and there is no overlapping in the phases. Typically, the outcome of one phase acts as the input for the next phase sequentially.

But, in practice, software development process is not a simple linear model but involves feedback from one phase to another. Depending on the budget (cost), a few numbers of iterations are normal. Then, proceed to the consecutive phase.

- **Requirements definition**
  - *Requirements elicitation*: Define the system (services, constraints, and goals) in terms understood by the system users
  - *Requirements analysis*: Technical specifications of the system in terms understood by designer and developer
- **System and software design**
  - Specify hardware and system requirements
  - Define overall system architecture
- **Implementation and unit testing**
  - System is developed in small programs called units/ components
  - Each unit is tested for its functionality (referred as Unit Testing) to validate its correctness
- **Integration and system testing**
  - Units or components developed in the implementation phase are integrated
  - Post integration, the complete software system is tested to ensure that it meets the requirements
- **Operation and maintenance**
  - Software system is installed/deployed and put into practical use
  - Correct or fix the issues/errors as they are discovered after the release of software system
  - Include the requirements that were omitted from the original requirements as they are discovered, or add new functionalities as their need are identified

### **Waterfall Model Application**

- When the requirements are very well known, clear and fixed; that unlikely to change radically during the system development.
- Required technology is understood.
- Ample resources with required expertise are available.
- A variant of waterfall model is Formal development process. In this, a mathematical model of the system specification is created. This model is suitable for the development of system that has stringent safety, reliability, or security requirements.

### **Advantages of Waterfall Model**

- Simple and easy to understand and use.



- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time. Phases do not overlap.
- Works well for smaller projects where requirements are very well understood.
- Process and results are well documented.

### Disadvantages of Waterfall Model

- Not suitable for the projects where requirements are at a moderate to high risk of changing.
- Difficult to accommodate the changes that were not well-thought out in the concept stage but discovered during the testing phase
- No working software is produced until late during the life cycle.
- Very less customer interaction is involved during the development of the product
- Integration is done at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early
- Once the product is developed and if any failure occurs then the cost of fixing such issues are very high, because it will need to update everywhere from document till the logic

### INCREMENTAL DEVELOPMENT

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

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

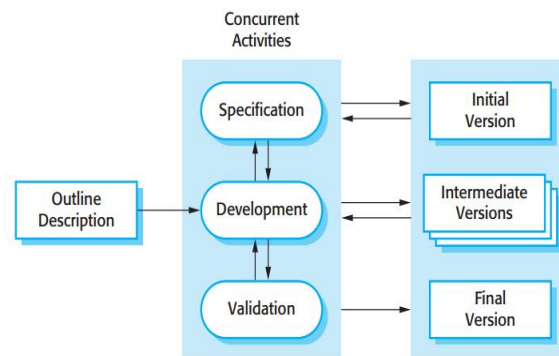


Figure 1.4: Incremental model [1]

Incremental software development, shown in Figure 1.4, which is a fundamental part of *agile approaches*, is suitable for the projects where *system requirements rapidly during the development*. Incremental development reflects the way that we solve the problems, i.e., move toward a solution of a problem is a series of steps, backtracking when we realized that we have made a mistake.

Customers can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current incremental has to be changed and possibly, new functionality defined for later increments.

### Advantages of Incremental Development

- The cost of accommodating changing customer requirements is reduced.
- It is easier to get customer feedback on the development work that has been done.
- More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included

### Disadvantages of Incremental Development

- Not suitable for large, complex, long-lifetime systems, where different teams develop different parts of the system.
- Incremental delivery and deployment is not always possible as experimenting with new software can disrupt normal business processes.
- Not suitable for a large system development. A large system needs 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.

## REUSE ORIENTED SOFTWARE ENGINEERING

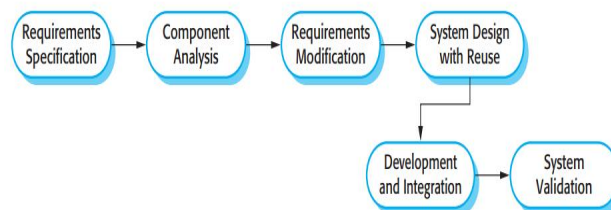


Figure 1.5: Reuse oriented software engineering [1]

When people working on the project know of designs or code that is similar to what is required, there is some software reuse. People look for design or code that is similar to what is required, modify them as needed, and incorporate them into their system.

Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages, shown in Figure 1.5, in a reuse oriented process are different.

- **Component analysis:** Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.
- **Requirements modification:** During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available

components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.

- ***System design with reuse:*** During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater for this. Some new software may have to be designed if reusable components are not available.
- ***Development and integration:*** Software that cannot be externally procured is developed, and the components and COTS (commercial off-the-shelf systems) systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

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

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

### **Advantages of Reuse Oriented Software Engineering**

- 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

### **Disadvantages of Reuse Oriented Software Engineering**

- Requirements compromises are inevitable and this may lead to a system that does not meet the real needs of users.
- 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

## **1.5. Process Iteration**

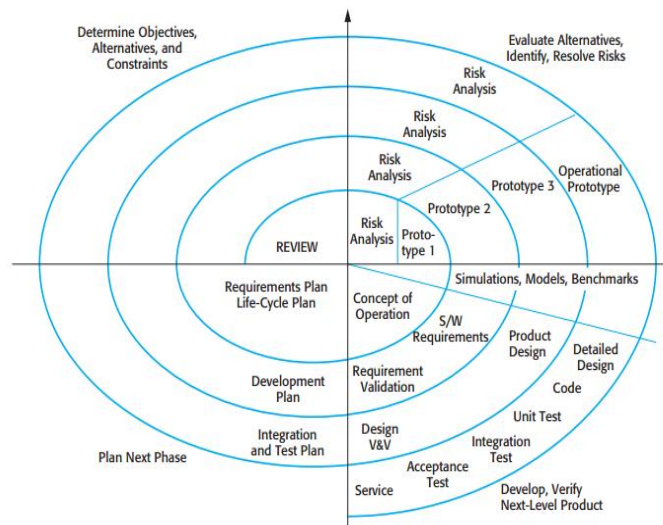
Changes are usually unavoidable in all large software projects. The system requirements change as organization continuously responds to the changing environment and conditions. Management priorities may change. Due to the quick progress in technologies, designs and implementation will change. This means that the process activities are regularly repeated as the system is reworked in response to change requirements.

The following two process models have been designed to support process iteration:

- ***Incremental development:*** The software specification, design and implementation are broken down into a series of increments that are each developed in turn. (Refer the above section for Incremental Development)
- ***Spiral development:*** The development of the system spirals outwards from an initial outline through to the final developed system

## SPIRAL DEVELOPMENT

A spiral development is risk driven software development process. It assumes that changes are as a result of project risks, thus, includes explicit risk management activities to reduce these risks. In this, software development process is represented as a spiral rather than a sequence of activities. There is some backtracking from one activity to another. Each spiral represents a phase of the software development process. Each loop in the spiral is divided into four sectors. Each phase repeatedly passes through the sectors. The spiral model is similar to the incremental model, with more emphasis placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation, shown in Figure 1.6. A software project repeatedly passes through these phases in iterations (called Spirals in this model).



I. Sommerville, 'Software Engineering', Ninth Edition, Pearson Education, Inc., 2011.

Figure 1.6: Boehm's spiral model of software process [1]

- **Objective setting**
  - Specific objectives for that phase of the project are defined
  - Constraints on the process and the products are identified and a detailed management plan is drawn up
  - Project risks are identified
  - Alternative strategies, depending on these risks, may be planned
- **Risk assessment and reduction**
  - Identified project risks are analyzed
  - Measures are adopted to reduce /minimize the risk
- **Development and validation**
  - Development
- **Planning**
  - Project is reviewed and decision is made whether to continue with a further loop of the spiral
  - Plans are drawn up for the next phase of the project, if necessary (if next loop is needed)

## **Spiral Model Application**

- For medium to high-risk projects (risk evaluation is important)
- Users are unsure of their needs (i.e., requirements are complicated and require continuous clarification)
- Suitable for new product line
- Significant changes are expected (research and exploration)

## **Advantages of Spiral Model**

- High amount of risk analysis hence, avoidance of risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional functionality can be added at a later date.
- Software is produced early in the software lifecycle

## **Disadvantages of Spiral Model**

- Can be a costly model to use; spiral may go infinitely
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects

### **1.6. 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. The four basic activities are [1]:

- Software specification
- Software design and implementation
- Software validation
- Software evolution

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.

## **SOFTWARE SPECIFICATION**

A software requirement is defined as a condition to which a system must comply. Software specification or requirements management is the process of understanding and defining what functional and non-functional requirements are required for the system and identifying the constraints on the system's operation and development. The requirements engineering process results in the production of a software requirements document that is the specification for the system.

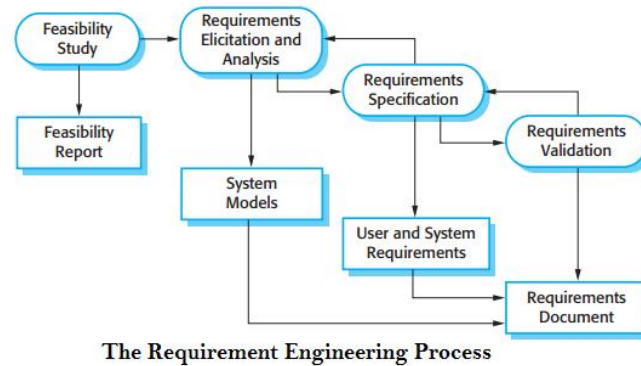


Figure 1.7: The requirement engineering process [1]

There are four main phases in the requirements engineering process, shown in Figure 1.7:

- **Feasibility study:** In this study an estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and whether it can be developed within existing budgetary constraints.
- **Requirements elicitation and analysis:** This is the process of deriving the system requirements through observation of existing systems, discussions with potential users, requirements workshop, storyboarding, etc.
- **Requirements specification:** This 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 (functional) requirements and system (non-functional) requirements.
- **Requirements validation:** It is determined whether the requirements defined are complete. This activity also checks the requirements for consistency.

The activities in the requirement process are not simply carried out in a strict sequence. The activities of analysis, definition, and specification are interleaved.

## SOFTWARE DESIGN AND IMPLEMENTATION

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

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

The design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.

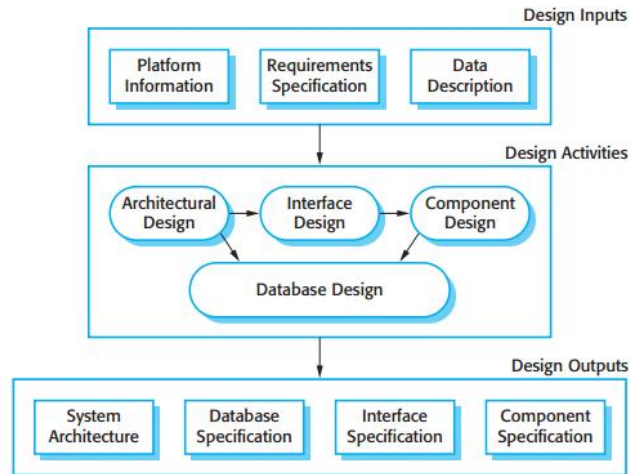


Figure 1.8: A general model of the design process [1]

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

The four activities that may be part of the design process for information systems, shown in Figure 1.8, are:

- **Architectural design**, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
- **Interface design**, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
- **Component design**, where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
- **Database design**, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.



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

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

Programming is a personal activity and there is no general process that is usually followed. Some programmers start with components that they understand, develop these, and then move on to less-understood components. Others take the opposite approach, leaving familiar components till last because they know how to develop them. Some developers like to define data early in the process then use this to drive the program development; others leave data unspecified for as long as possible.

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

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

## SOFTWARE VALIDATION

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

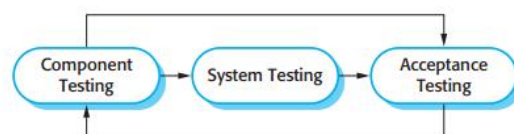


Figure 1.9: Stages of testing [1]



The stages in the testing process, also shown in Figure 1.9, are:

- **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, that can re-run component tests when new versions of the component are created, are commonly used.
- **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.
- **Acceptance testing.** This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable.

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

If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment.

When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans. The testing phases of plan-driven software are shown in Figure 1.10. An independent team of testers works from these pre-formulated test plans, which have been developed from the system specification and design.

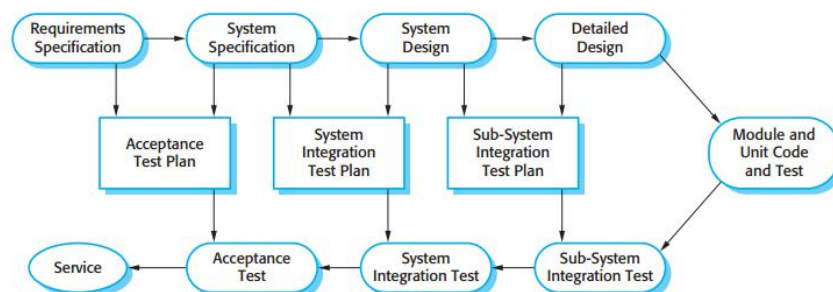


Figure 1.10: Testing phases in a plan-driven software process [1]

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

When a system is to be marketed as a software product, a testing process called 'beta testing' is often used. *Beta testing* involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the system builders. After this feedback, the system is modified and released either for further beta testing or for general sale. Alpha and beta testing are a kind of acceptance testing. The differences between alpha and beta testing are mentioned in Table 1.4.

| Alpha testing   | Beta testing  |
|---|---|
| There is a dedicated test team for alpha testing. It is done onsite therefore developers as well as business analyst are involved with the testing team. It is not open for market or public. | Beta testing is done by customers or end users at their own site.                                   |
| It is done for software application, project and product.   | It is usually done for software product like operating system, write or paint utilities, games etc. |
| It is done before the launch of software product into the market  | It is done at the time of software product marketing.   |

Table 1.4: Differences between alpha and beta testing

## 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.

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 where software is continually changed over its lifetime in response to changing requirements and customer needs, as shown in Figure 1.11.

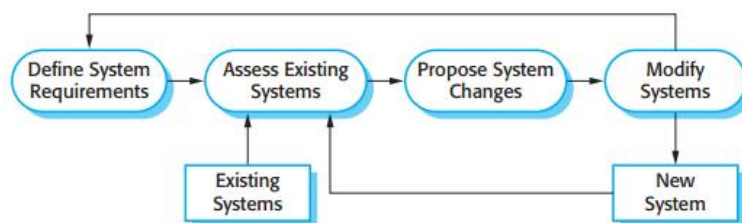


Figure 1.11: System evolution [1]

### 1.7. Computer Aided Software Engineering (CASE)

CASE means development and maintenance of software projects with help of various automated software tools. The goal of introducing CASE tools is the reduction of the time and cost of software development and the enhancement of the quality of the systems developed.

Software development tools (sometimes called 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 (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.

The improvements from the use of software tools are limited by three factors:

- For large projects, the major problems are caused by poorly defined requirements and requirements that change to reflect changing business needs. These lead to extensive rework of the software. These result in unforeseen costs and delays to the software development. As this is a business rather than a technical problem, tool support cannot reduce these costs.
- Software engineering is, essentially, a design activity based on creative thought. Software tools automate routine activities but attempts to harness artificial intelligence technology to provide support for design have not been successful.
- In most organizations, software engineering is a team activity and software engineers spend quite a lot of time interacting with other team members. Although tools for collaboration are improving, they are still fairly limited and do not support creative processes in an effective way.

## **1.8. Functional and Non-Functional Requirements**

Software requirements may be functional or non-functional

### **FUNCTIONAL REQUIREMENTS**

Functional requirements are the statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do. They describe the functionality of the system that can be modeled with use cases.

These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user

requirements, functional requirements are usually described in an abstract way that can be understood by system users.

However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail. Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.

In principle, the functional requirements specification of a system should be both *complete* and *consistent*. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory definitions.

## NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services. They describe system properties related to e.g. system performance, usability, security, maintainability...

Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

NFR classification and examples

- **Product requirements**
  - Requirements which specify that the delivered product must behave in a particular way, e.g., R78: The time period between motion detection and start of alarm signal shall be less than 0.5 seconds.
- **Organizational requirements**
  - Requirements which are a consequence of organizational policies and procedures, e.g., the signal power line transmission shall use the standard X10.
- **External requirements**
  - Requirements which arise from factors which are external to the system and its development process, e.g., the data recorded in the video surveillance shall not be disclosed to the public.

In reality, the distinction between different types of requirement is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a nonfunctional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

## 1.9. User Requirements

User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

They, often referred to as user needs, describe what the user does with the system, such as what activities that users must be able to perform. User requirements are generally documented in a User Requirements Document (URD) using narrative text. User requirements are generally signed off by the user and used as the primary input for creating system requirements.

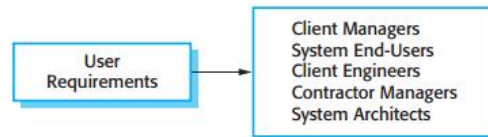


Figure 1.12: Reader of user requirements [1]

An important and difficult step of designing a software product is determining what the user actually wants it to do. This is because the user is often not able to communicate the entirety of their needs and wants, and the information they provide may also be incomplete, inaccurate and self-conflicting. The responsibility of completely understanding what the customer wants falls on the business analyst. This is why user requirements are generally considered separately from system requirements. The business analyst carefully analyzes user requirements and carefully constructs and documents a set of high quality system requirements ensuring that the requirements meet certain quality characteristics.

## 1.10. System Requirements

System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

They are the building blocks developers use to build the system. These are the traditional "*shall*" statements that describe what the system "shall do." System requirements are classified as either functional or non-functional requirements. A functional requirement specifies something that a user needs to perform their work. For example, a system may be required to enter and print cost estimates; this is a functional requirement. Non-functional requirements specify all the remaining requirements not covered by the functional requirements. Non-functional requirements are sometimes called quality of service requirements. The plan for implementing functional requirements is detailed in the system design. The plan for implementing supplemental requirements is detailed in the system architecture.

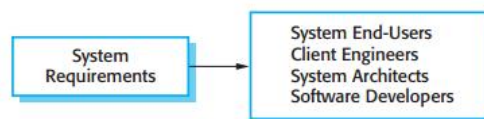


Figure 1.13: Reader of system requirements [1]

### 1.11. Interface Specification

Almost all software systems must operate with existing systems that have already been implemented and installed in an environment. If the new system and the existing systems must work together, the interfaces of existing systems have to be precisely specified.

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

There are three types of interface that may have to be defined:

- **Procedural interfaces** where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Application Programming Interfaces (APIs).
- **Data structures** that are passed from one sub-system to another. Graphical data models are the best notations for this type of description. Example: program descriptions in Java or C++ can be generated automatically from these descriptions.
- **Representations** of data (such as the ordering of bits) that have been established for an existing sub-system. These interfaces are most common in embedded, real-time system.

The *procedural interface* offered by a print server manages a queue of requests to print files on different printers. Users may examine the queue associated with a printer and may remove their print jobs from that queue. They may also switch jobs from one printer to another.

### 1.12. The Software Requirements Documents

The software requirements document (sometimes called the software requirements specification or SRS) is the official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. Sometimes, the user and system requirements are integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented in a separate document. Requirements documents are essential when an outside contractor is developing the software system.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software, shown in Figure 1.14.

The diversity of possible users means that the requirements document has to be a compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers, and including information about possible system evolution. Information on anticipated changes can help system designers avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements. The structure and chapters in software requirement document are mentioned in Table 1.5.

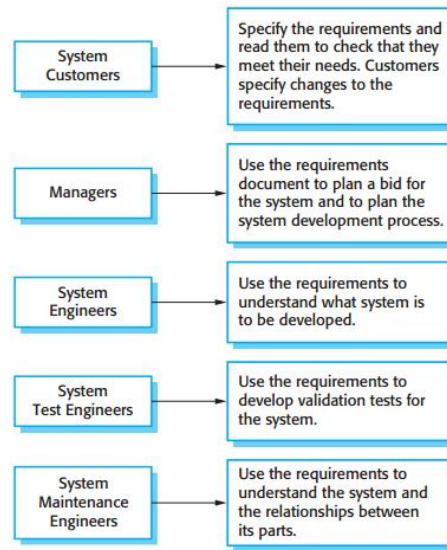


Figure 1.14: Users of software requirements document [1]

| Chapter                           | Description   |
|-----------------------------------|---|
| Preface                           | This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.   |
| Introduction                      | This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.  |
| Glossary                          | This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.   |
| User requirements definition      | Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.  |
| System architecture               | This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.  |
| System requirements specification | This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.  |
| System models                     | This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.  |
| System evolution                  | This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.  |
| Appendices                        | These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data. |
| Index                             | Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.   |

Table 1.5: The structure of a requirements document [1]

### 1.13. Feasibility Study

An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study



should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

The aims of a feasibility study are to find out whether the system is worth implementing and if it can be implemented, given the existing budget and schedule. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it is worth carrying on with the requirements engineering and system development process.

If a system does not support the business objectives, it has no real value to the business. While this may seem obvious, many organizations develop systems which do not contribute to their objectives either because they don't have a clear statement of these objectives, because they fail to define the business requirements for the system or because other political or organization factors influence the system procurement.

Carrying out a feasibility study involves information assessment, information collection and report writing. The information assessment phase identifies the information that is required to answer the three questions set out above. Once the information has been identified, you should question information sources to discover the answers to these questions. Some examples of possible questions that might ask:

- How would the organization cope if this system was not implemented?
- What are the problems with current processes and how would a new system help alleviate these problems?
- What direct contribution will the system make to the business objectives and requirements?
- Can information be transferred to and from other organizational systems?
- Does the system require technology that has not previously been used in the organization?
- What must be supported by the system and what need not be supported?

Information sources may be the managers of departments where the system will be used, software engineers who are familiar with the type of system that is proposed, technology experts, end-users of the system, etc. Normally, you should try and complete a feasibility study in two or three weeks.

When the information is available, you then write the feasibility study report. You should make a recommendation about whether or not the system development should continue. In the report, you may propose changes to the scope, budget and schedule of the system and suggest further high-level requirements for the system.

#### **1.14. Requirements Elicitation and Analysis**

In requirement elicitation and analysis, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

Requirements elicitation and analysis may involve a variety of different kinds of people in an organization. A system stakeholder is anyone who should have some direct or indirect influence on the system requirements. Stakeholders include end users who will interact with the system and anyone else in an organization who will be affected by it. Other system stakeholders might be engineers who are



developing or maintaining other related systems, business managers, domain experts, and trade union representatives.

Each organization will have its own version or instantiation of this general model depending on local factors such as the expertise of the staff, the type of system being developed, the standards used, etc.

The process activities, also shown in Figure 1.15, are:

1. **Requirements discovery.** This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity. There are several complementary techniques that can be used for requirements discovery.
2. **Requirements classification and organization.** This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.
3. **Requirements prioritization and negotiation.** Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.
4. **Requirements specification.** The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.

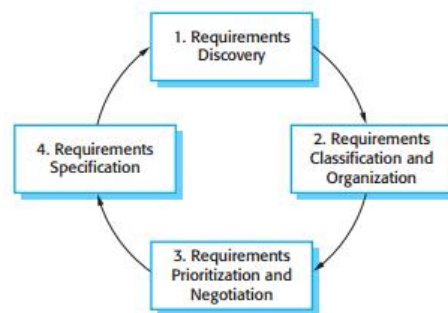


Figure 1.15: The requirements elicitation and analysis process [1]

Figure 1.15, above shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with the requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document is complete.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

- Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
- Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
- Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
- Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
- The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Inevitably, different stakeholders have different views on the importance and priority of requirements and, sometimes, these views are conflicting. During the process, you should organize regular stakeholder negotiations so that compromises can be reached. It is impossible to completely satisfy every stakeholder but if some stakeholders feel that their views have not been properly considered then they may deliberately attempt to undermine the requirement engineering process.

### 1.15. Requirements Validation and Management

#### REQUIREMENT VALIDATION

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address wicked' problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing. The system requirements must then also evolve, shown in Figure 1.16, to reflect this changed problem view.

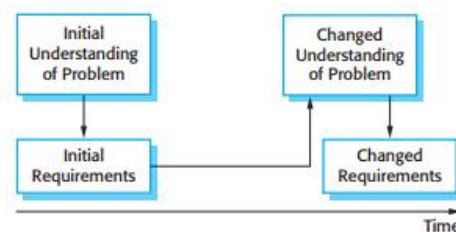


Figure 1.16: Requirements evolution [1]

Requirements validation is the process of checking that requirements actually define the system that the customer really wants. It overlaps with analysis as it is concerned with finding problems with the requirements. Requirements validation is important because errors in a requirements document can lead to

extensive rework costs when these problems are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed. Furthermore the system must then be re-tested.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

- **Validity checks.** A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.
- **Consistency checks.** Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
- **Completeness checks.** The requirements document should include requirements that define all functions and the constraints intended by the system user.
- **Realism checks.** Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.
- **Verifiability.** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

There are a number of requirements validation techniques that can be used individually or in conjunction with one another:

1. **Requirements reviews.** The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. **Prototyping.** In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
3. **Test-case generation.** Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

## REQUIREMENT MANAGEMENT

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address ‘wicked’ problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be

incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing.

Once a system has been installed and is regularly used, new requirements inevitably emerge. It is hard for users and system customers to anticipate what effects the new system will have on their business processes and the way that work is done. Once end users have experience of a system, they will discover new needs and priorities.

There are several reasons why change is inevitable:

- ***The business and technical environment of the system always changes after installation.*** New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- ***The people who pay for a system and the users of that system are rarely the same people.*** System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery; new features may have to be added for user support if the system is to meet its goals.
- ***Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.*** The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements management is the process of understanding and controlling changes to system requirements. You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements. The formal process of requirements management should start as soon as a draft version of the requirements document is available. However, you should start planning how to manage changing requirements during the requirements elicitation process.

### **Requirement Management Planning**

Planning is an essential first stage in the requirements management process. The planning stage establishes the level of requirements management detail that is required. During the requirements management stage, you have to decide on:

1. ***Requirements identification.*** Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
2. ***A change management process.*** This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. ***Traceability policies.*** These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.

4. **Tool support.** Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements management needs automated support and the software tools for this should be chosen during the planning phase. You need tool support for:

1. **Requirements storage.** The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
2. **Change management.** The process of change management (Figure 1.17) is simplified if active tool support is available.
3. **Traceability management.** As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, it may not be necessary to use specialized requirements management tools. The requirements management process may be supported using the facilities available in word processors, spreadsheets, and PC databases. However, for larger systems, more specialized tool support is required.

### Requirement Change Management

Requirements change management should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

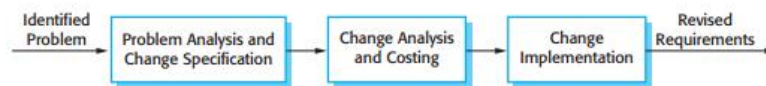


Figure 1.17: Requirement change management [1]

There are three principal stages to a change management process:

1. **Problem analysis and change specification.** The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
2. **Change analysis and costing.** The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
3. **Change implementation.** The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you

can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. You should try to avoid this as it almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document or to add information to the requirements document that is inconsistent with the implementation.

Agile development processes, such as extreme programming, have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped.

## REFERENCES

- [1] I. Sommerville , 2011. Software Engineering. Ninth Edition, Addison-Wesley.
- [2] <https://www.infoq.com/articles/standish-chaos-2015>
- [3] K. Emam and A.G. Koru, 2008. A Replicated Survey of IT Software Project Failures. IEEE Software, Volume: 25, Issue: 5, Sept.-Oct. 2008.
- [4] S. Cavell, 1998. The Bull Report. Cited from: [http://www.it-cortex.com/Stat\\_Failure\\_Rate.htm](http://www.it-cortex.com/Stat_Failure_Rate.htm)
- [5] B. Whittaker, 1999. What went wrong? Unsuccessful information technology projects, KPMG, MCB University Press, Information Management & Computer Security, 7(1) P23-29
- [6] C. Sauer & C. Cuthbertson, 2003. The State of IT Project Management in 2002–2003, Computer Weekly
- [7] K. Miller, R. Dawson & M. Bradley, 2007. IT Project Success –The Computing Holy Grail. In proceedings of the SQM2007.
- [8] D. Feitelson, 2009. Software Crisis. Cited from: <http://www.cs.huji.ac.il/~feit/sem/se09/7-crisis.pdf>
- [9] S. Amarasinghe, 2017. Third Software Crisis: The Multi-core Problem. Cited from: <http://cis.poly.edu/cs614/NewdirectionsSaman1206.pdf>
- [10] IEEE Computer Society, 2009. IEEE Standard for a Software Quality Metrics Methodology. Cited from: [https://cow.ceng.metu.edu.tr/Courses/download\\_courseFile.php?id=2681](https://cow.ceng.metu.edu.tr/Courses/download_courseFile.php?id=2681)
- [11] M. Barbacci et al., 1995. Quality Attributes. SEI, Carnegie Mellon University.

[12] <http://www.sqa.net/iso9126.html>

## 2. SYSTEM MODELS

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification.

Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system's structure and operation. You may develop models of both the existing system and the system to be developed:

1. ***Models of the existing system are used during requirements engineering.*** They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
2. ***Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders.*** Engineers use these models to discuss design proposals and to document the system for implementation. In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

The most important aspect of a system model is that it leaves out detail. A model is an abstraction of the system being studied rather than an alternative representation of that system. Ideally, a representation of a system should maintain all the information about the entity being represented. An abstraction deliberately simplifies and picks out the most salient characteristics.

You may develop different models to represent the system from different perspectives. For example:

- An ***external perspective***, where you model the context or environment of the system.
- An ***interaction perspective*** where you model the interactions between a system and its environment or between the components of a system.
- A ***structural perspective***, where you model the organization of a system or the structure of the data that is processed by the system.
- A ***behavioral perspective***, where you model the dynamic behavior of the system and how it responds to events.

The UML has many diagram types and so supports the creation of many different types of system model. The five diagram types of UML diagram that could represent the essentials of a system are:

1. **Activity diagrams**, which show the activities involved in a process or in data processing.
2. **Use case diagrams**, which show the interactions between a system and its environment.
3. **Sequence diagrams**, which show interactions between actors and the system and between system components.
4. **Class diagrams**, which show the object classes in the system and the associations between these classes.
5. **State diagrams**, which show how the system reacts to internal and external events.

## 2.1. Context Models

At an early stage in the specification of a system, you should decide on the system boundaries. This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system's environment. You may decide that automated support for some business processes should be implemented but others should be manual processes or supported by different systems. You should look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

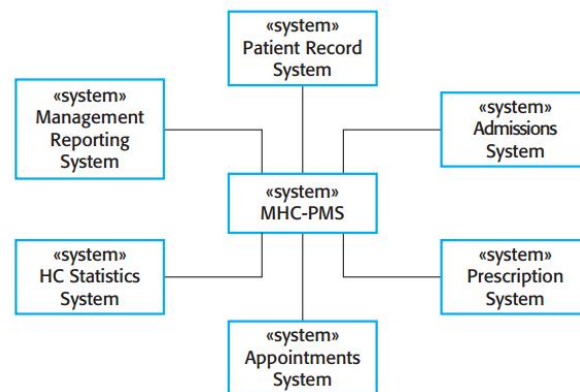


Figure 2.1: The context of the Mental Health Care-Patient Management System (MHC-PMS) [1]

For example, in developing the specification for the patient information system for mental healthcare (Figure 2.1), you have to decide:

- Whether the system should focus exclusively on collecting information about consultations (using other systems to collect personal information about patients)?
- Or whether it should collect personal patient information?

The advantage of relying on other systems for patient information is that you avoid duplicating data. The major disadvantage, however, is that using other systems may make it slower to access information.



The definition of a system boundary is not a value-free judgment. Social and organizational concerns may mean that the position of a system boundary may be determined by non-technical factors. For example, a system boundary may be deliberately positioned so that the analysis process can all be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; it may be positioned so that the system cost is increased and the system development division must therefore expand to design and implement the system.

Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity.

In Figure 2.1, MHC-PMS is connected to an appointments system and a more general patient record system with which it shares data. The system is also connected to systems for management reporting and hospital bed allocation and a statistics system that collects information for research. Finally, it makes use of a prescription system to generate prescriptions for patients' medication.

Context models normally show that the environment includes several other automated systems. However, they do not show the types of relationships between the systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network or not connected at all. They might be physically co-located or located in separate buildings. All of these relations may affect the requirements and design of the system being defined and must be taken into account.

Therefore, simple context models are used along with other models, such as business process models (activity diagram). These describe human and automated processes in which particular software systems are used.

### **2.1.1. ACTIVITY DIAGRAM (AD)**

Activity diagrams represent the dynamics of the system. They are flow charts that are used to show the workflow of a system. They show:

- The flow of control from activity to activity in the system
- What activities can be done in parallel
- Alternate paths through the flow

They can show the flow across use cases or within a use case.

### **Basic Symbols and Notations**

The basic symbols and notations of AD are shown in Figure 2.2

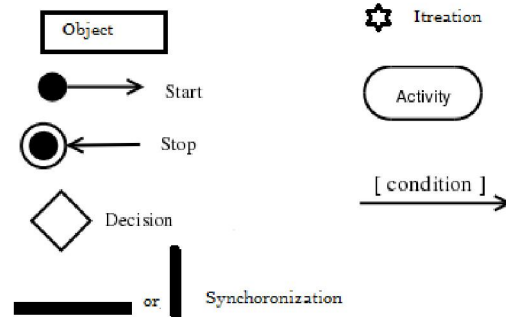
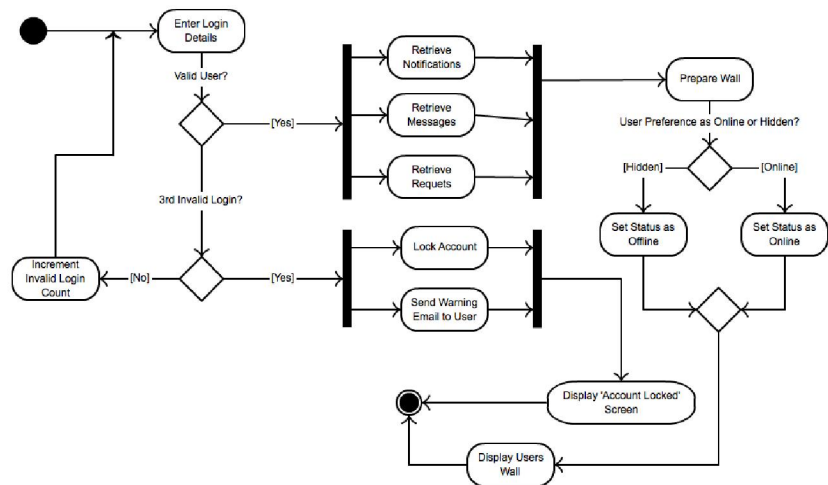


Figure 2.2: Basic symbols/notations of AD [1]

- **Start Point:** A small filled circle followed by an arrow represents the start point for any activity diagram.
- **Stop Point:** An encircled black circle represents the end node.
- **Activity:** The core symbol of AD is an activity represented by a rounded rectangle. An activity is some task which needs to be done. Each activity can be followed by another activity (sequencing). Triggers from the activity may be guarded as in state diagrams.
- **Action Flow:** Action flows, also called edges and paths, illustrate the transitions from one action state to another. They are usually drawn with an arrowed line.
- **Decisions:** A diamond represents a decision with alternate paths. When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities. The outgoing alternates should be labelled with a condition or guard expression. You can also label one of the paths "else."
- **Synchronization:** A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram. A join node joins multiple concurrent flows back into a single outgoing flow. A fork and join node used together are often referred to as synchronization.
- **Iteration** is represented by a \* (asterisk) on the trigger

For example, the AD for Facebook login is shown in Figure 2.3.



[online diagramming & design] [creately.com](https://creately.com)

Figure 2.3: AD for Facebook login

## 2.2. Behavioral Models

Behavioral models are models of the dynamic behavior of the system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. You can think of these stimuli as being of two types:

1. **Data.** Some data arrives that has to be processed by the system.
2. **Events.** Some event happens that triggers system processing. Events may have associated data but this is not always the case.

Many business systems are data processing systems that are primarily driven by data. They are controlled by the data input to the system with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, a phone billing system will accept information about calls made by a customer, calculate the costs of these calls, and generate a bill to be sent to that customer. By contrast, real-time systems are often event driven with minimal data processing. For example, a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone or the pressing of keys on a handset by capturing the phone number, etc.

### 2.2.1. DATA-DRIVEN MODELING

Data-driven models show the sequence of actions involved in processing input data and generating an associated output. They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system. That is, they show the entire sequence of actions that take place from an input being processed to the corresponding output, which is the system’s response.

Data-flow models are useful because tracking and documenting how the data associated with a particular process moves through the system helps analysts and designers understand what is going on. Data-flow diagrams are simple and intuitive and it is usually possible to explain them to potential system users who can then participate in validating the model.

The UML does not support data-flow diagrams as they were originally proposed and used for modeling data processing. The reason for this is that DFDs focus on system functions and do not recognize system objects. However, because data-driven systems are so common in business, UML 2.0 introduced activity diagrams, which are similar to data-flow diagrams. An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams.

#### 2.2.1.1. DATA FLOW DIAGRAM (DFD)

A DFD maps out the flow of information for any process or system. It uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination. Data flowcharts can range from simple, even hand-drawn process overviews, to in-depth, multi-level DFDs that dig progressively deeper into how the data is handled. They can be used to analyze an existing system or model a new one.

#### Basic Symbols and Notations

Using any convention's DFD rules or guidelines, the symbols (shown in Figure 2.4) depict the four components of data flow diagrams.

1. **External entity:** an outside system that sends or receives data, communicating with the system being diagrammed. They are the sources and destinations of information entering or leaving the system. They might be an outside organization or person, a computer system or a business system. They are also known as terminators, sources and sinks or actors. They are typically drawn on the edges of the diagram.
2. **Process:** any process that changes the data, producing an output. It might perform computations, or sort data based on logic, or direct the data flow based on business rules. A short label is used to describe the process, such as "Submit payment."
3. **Data store:** files or repositories that hold information for later use, such as a database table or a membership form. Each data store receives a simple label, such as "Orders."
4. **Data flow:** the route that data takes between the external entities, processes and data stores. It portrays the interface between the other components and is shown with arrows, typically labeled with a short data name, like "Billing details."









| Notation        | Yourdon and Coad  | Gane and Sarson  |
|-----------------|---|--|
| External Entity |    |    |
| Process         |   |   |
| Data Store      |  |  |
| Data Flow       |  |  |

Figure 2.4: Symbols/Notations of DFD [2]

### DFD Rules and Tips

- Each process should have at least one input and an output.
- Each data store should have at least one data flow in and one data flow out.
- Data stored in a system must go through a process.
- All processes in a DFD go to another process or a data store.

### DFD Layers and Levels

A data flow diagram can dive into progressively more detail by using levels and layers, zeroing in on a particular piece. DFD levels are numbered 0, 1 or 2, and occasionally go to even Level 3 or beyond. The necessary level of detail depends on the scope of what you are trying to accomplish.

- **DFD Level 0 is also called a Context Diagram.** It's a basic overview of the whole system or process being analyzed or modeled. It's designed to be an at-a-glance view, showing the system as a single high-level process, with its relationship to external entities. It should be easily

understood by a wide audience, including stakeholders, business analysts, data analysts and developers. The DFD level 0 for food ordering system is shown in Figure 2.5.

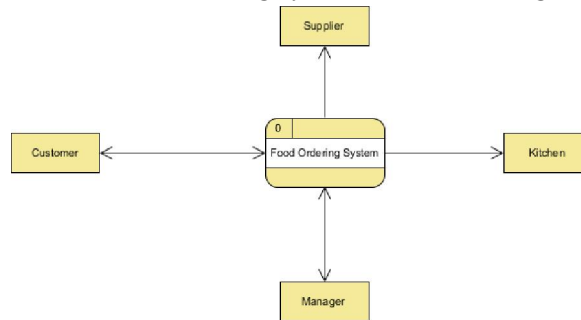


Figure 2.5: DFD of food ordering system (Level 0)

- **DFD Level 1** provides a more detailed breakout of pieces of the Context Level Diagram. You will highlight the main functions carried out by the system, as you break down the high-level process of the Context Diagram into its sub-processes. The DFD level 1 for food ordering system is shown in Figure 2.6.

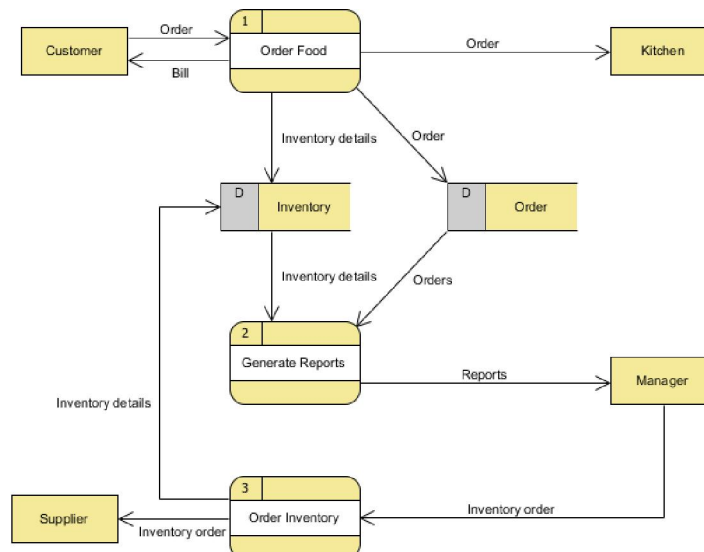


Figure 2.6: DFD of food ordering system (Level 1)

- **DFD Level 2** then goes one step deeper into parts of Level 1. It may require more text to reach the necessary level of detail about the system's functioning.
- **Progression to Levels 3, 4 and beyond** is possible, but going beyond Level 3 is uncommon. Doing so can create complexity that makes it difficult to communicate, compare or model effectively.

## DFD Balancing

The concept of balancing states that all the incoming flows to a process and all the outgoing flows from a process in the parent diagram should be preserved at the next level of decomposition. Process decomposition lets you organize your overall DFD in a series of levels so that each level provides successively more detail about a portion of the level above it. The goal of the balancing feature is to check

your system internal consistency, which is particularly useful as different levels of expertise are generally involved in a project.

### Rules of DFD Balancing

- The matters that must be focused in DFD that has more than one level:
  - Must be found balance input and output between one level and next level
  - Balance between level 0 and level 1 are seen in input/output from data flow to or from terminal in level 0, while balance between level 1 and level 2 are seen in input/output from data flow to/from process concerned
  - Data flow name, data storage and terminal in every level must same if the object is same

#### 2.2.1.2. SEQUENCE DIAGRAM

Sequence diagram describes the flow of messages, events, and actions between objects. It shows concurrent processes and activations, and time sequences that are not easily depicted in other diagrams. It is typically used during analysis and design to document and understand the logical flow of your system.

#### Basic Symbols and Notations

- **Participant:** object or entity that acts in the diagram (diagram starts with an unattached "found message" arrow)
- **Message:** communication between participant objects
- **The axes** in a sequence diagram: (horizontal: which object/participant is acting, vertical: time (down -> forward in time))

The sequence diagram for telephone call is shown in Figure 2.7.

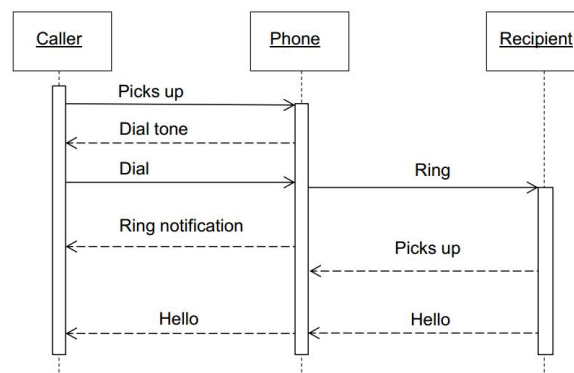


Figure 2.7: Sequence diagram for telephone call

#### 2.2.2. EVENT-DRIVEN MODELING

Event-driven modeling shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another. For example, a system controlling a valve may move from a state 'Valve open'

to a state ‘*Valve closed*’ when an operator command (the stimulus) is received. This view of a system is particularly appropriate for real-time systems.

### 2.2.2.1. STATE DIAGRAMS

**State diagrams** show system states and events that cause transitions from one state to another. They do not show the flow of data within the system but may include additional information on the computations carried out in each state. Some basic notations of state diagram are shown in Figure 2.8, and some examples of state diagrams are shown in Figure 2.9 and 2.10.

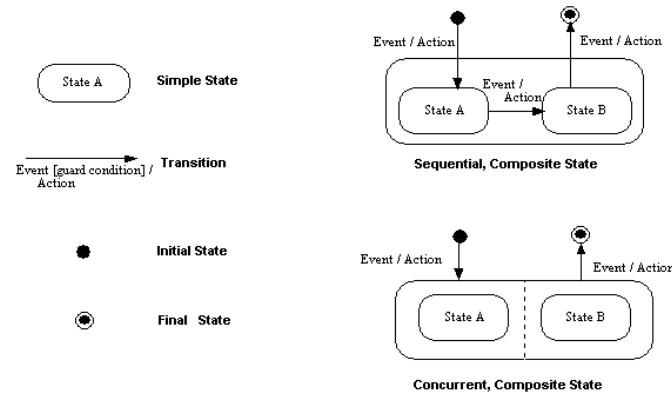


Figure 2.8: Basic notations of state diagram

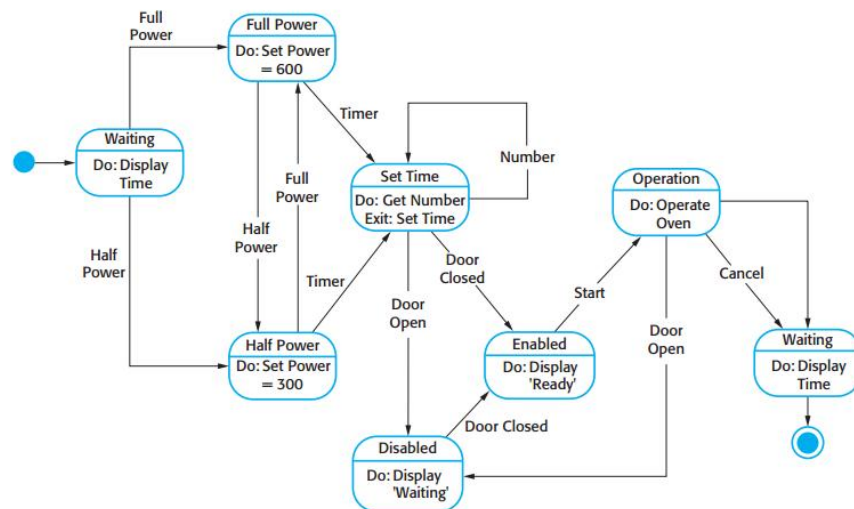


Figure 2.9: State diagram for a micro oven [1]

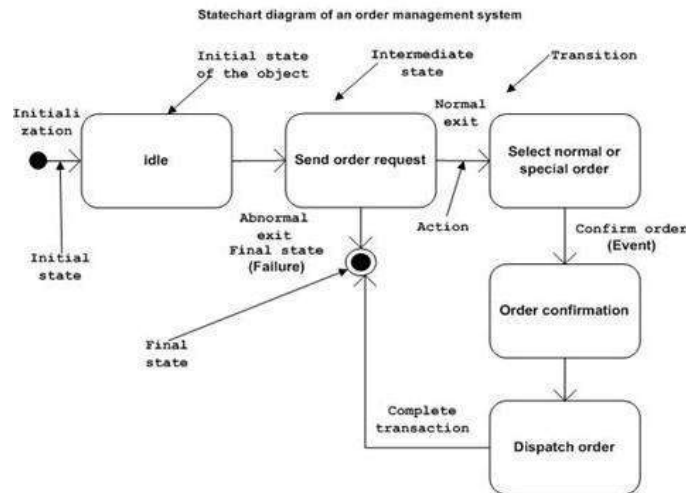


Figure 2.10: State-chart Diagram of an order management system [4]

Before drawing a State chart diagram we should clarify the following points –

- Identify the important objects to be analyzed.
- Identify the states.
- Identify the events.

## 2.3. Data and Object Models

### 2.3.1. DATA MODEL

A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to properties of the real world entities. For instance, a data model may specify that the data element representing a car be composed of a number of other elements which, in turn, represent the color and size of the car and define its owner.

A data model instance may be one of three kinds according to ANSI in 1975:

- **Conceptual data model:** describes the semantics of a domain, being the scope of the model. For example, it may be a model of the interest area of an organization or industry. This consists of entity classes, representing kinds of things of significance in the domain, and relationship assertions about associations between pairs of entity classes. A conceptual schema specifies the kinds of facts or propositions that can be expressed using the model. In that sense, it defines the allowed expressions in an artificial 'language' with a scope that is limited by the scope of the model.
- **Logical data model:** describes the semantics, as represented by a particular data manipulation technology. This consists of descriptions of tables and columns, object oriented classes, and XML tags, among other things.
- **Physical data model:** describes the physical means by which data are stored. This is concerned with partitions, CPUs, tablespaces, and the like.

Example, ER diagram is shown in Figure 2.11.



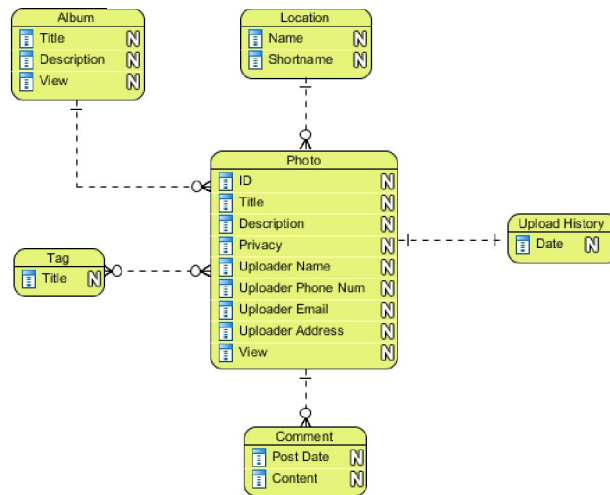


Figure 2.11: ER Diagram (Conceptual data model)

### 2.3.2. OBJECT MODEL

The properties of objects in general in a specific computer programming language, technology, notation or methodology that uses them. Examples are the object models of Java, the Component Object Model (COM), or Object-Modeling Technique (OMT). Such object models are usually defined using concepts such as class, generic function, message, inheritance, polymorphism, and encapsulation.

There are different types of Objects

- **Entity Objects:** Represent the persistent information tracked by the system (Application domain objects, also called “Business objects”)
- **Boundary Objects:** Represent the interaction between the user and the system
- **Control Objects:** Represent the control tasks performed by the system.

An object model consists of the following important features:

- **Object Reference:** Objects can be accessed via object references. To invoke a method in an object, the object reference and method name are given, together with any arguments.
- **Interfaces:** An interface provides a definition of the signature of a set of methods without specifying their implementation. An object will provide a particular interface if its class contains code that implements the method of that interface. An interface also defines types that can be used to declare the type of variables or parameters and return values of methods.
- **Actions:** An action in object-oriented programming (OOP) is initiated by an object invoking a method in another object. An invocation can include additional information needed to carry out the method. The receiver executes the appropriate method and then returns control to the invoking object, sometimes supplying a result.
- **Exceptions:** Programs can encounter various errors and unexpected conditions of varying seriousness. During the execution of the method many different problems may be discovered. Exceptions provide a clean way to deal with error conditions without complicating the code. A

block of code may be defined to throw an exception whenever particular unexpected conditions or errors arise. This means that control passes to another block of code that catches the exception.

## 2.4. Structured Methods

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design or dynamic models, which show the organization of the system when it is executing. These are not the same things—the dynamic organization of a system as a set of interacting threads may be very different from a static model of the system components.

You create structural models of a system when you are discussing and designing the system architecture. Architectural design is a particularly important topic in software engineering and UML component, package, and deployment diagrams may all be used when presenting architectural models.

### 2.4.1. CLASS DIAGRAMS

Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. Loosely, an object class can be thought of as a general definition of one kind of system object. An association is a link between classes that indicates that there is a relationship between these classes. Consequently, each class may have to have some knowledge of its associated class.

When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, a doctor, etc. As an implementation is developed, you usually need to define additional implementation objects that are used to provide the required system functionality.

Class diagrams in the UML can be expressed at different levels of detail. When you are developing a model, the first stage is usually to look at the world, identify the essential objects, and represent these as classes. The simplest way of writing these is to write the class name in a box. You can also simply note the existence of an association by drawing a line between classes.

#### Notations of Class Diagram

*The class shape itself consists of a rectangle with three rows. The top row contains the name of the class, the middle row has the attributes of the class, and the bottom section expresses the methods or operations that the class may utilize.* In a diagram, classes and subclasses are grouped together to show the static relationship between each object. The class diagram for a banking application is shown in Figure 2.12.

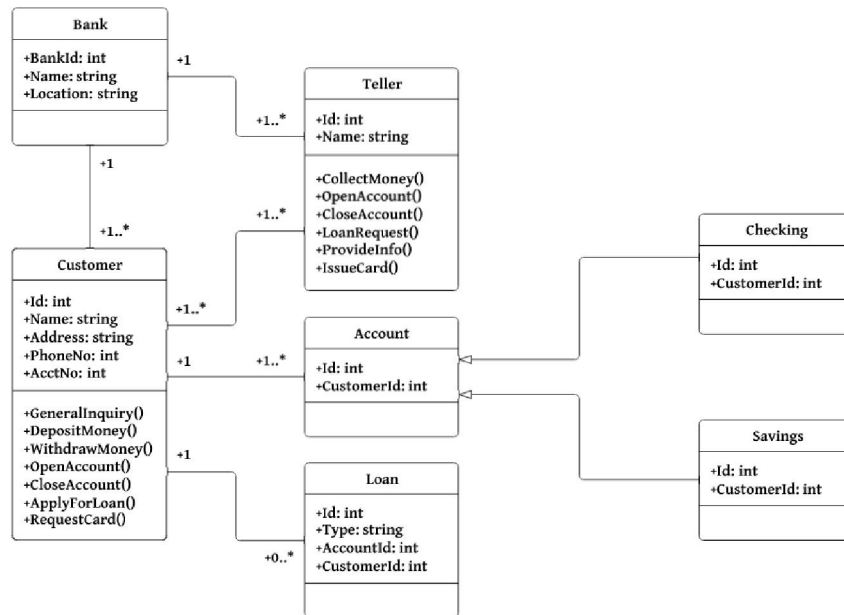


Figure 2.12: Class diagram for banking application

The class diagram is composed of three parts, shown in Figure 2.13:

1. **Upper Section** - Name of the class - This section is always required whether you are talking about the classifier or an object.
2. **Middle Section** - Attributes of the class - The attributes describe the variables that describe the qualities of the class. This is only required when describing a specific instance of a class.
3. **Bottom Section** - Class operations (methods) - Displayed in list format, each operation takes up its own line. The operations describe how a class can interact with data.

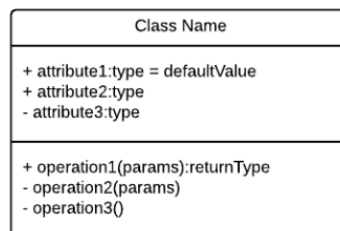


Figure 2.13: Class shape

## Member Access Modifier

All classes have different access levels depending on the access modifier (visibility). Here are the following access levels with their corresponding symbols:

- Public (+)
- Private (-)
- Protected (#)

- Package (~)
- Derived (/)
- Static (underlined)

## Member Scope

There are two scopes for members: classifiers and instances. Classifiers are static members while instances are the specific instances of the class.

## Object / Class Interactions in Class Diagrams

Interactions between objects and classes are an integral part of class diagrams.

**Inheritance** is when a child object assumes all the characteristics of its parent object. For example, as shown in Figure 2.14, if we had the object vehicle, a child class Go-go mobile would inherit all the attributes (speed, numbers of passengers, fuel) and methods (go(), stop(), changeDirection()) of the parent class in addition to the specific attributes(modelType, # of doors, autoMaker) and methods of its own class (Radio(), windshieldWiper(), ac/heat()). *Inheritance is shown in a class diagram by using a solid line with a closed, hollow arrow.*

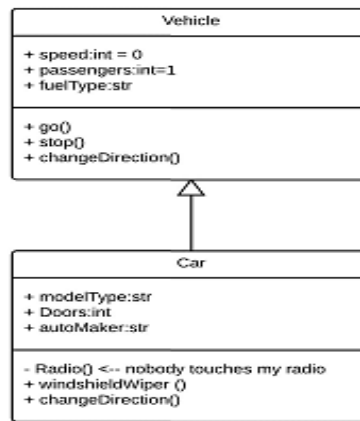


Figure 2.14: Inheritance

**Bidirectional** associations are the default associations between two classes and are represented by a straight line between two classes as shown in Figure 1.15. Both classes are aware of each other and of their relationship with each other. In the example above, the Go-go mobile class and RoadTrip class are interrelated. At one end of the line the Go-go mobile takes on the association of "assignedCar" with the multiplicity value of 0..1 which means that when the instance of RoadTrip exists, it can either have one instance of Go-go mobile associated with it or no Go-go mobiles associated with it. In this case, a separate Caravan class with a multiplicity value of 0..\* is needed to demonstrate that a RoadTrip could have multiple instances of Go-go mobiles associated with it. Since one Go-go mobile instance could have multiple "getRoadTrip" associations-- in other words, one go-go mobile could go on multiple road trips-- the multiplicity value is set to 0..\*

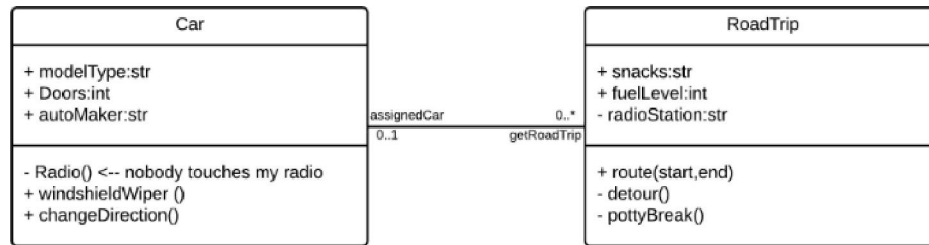


Figure 2.15: Bidirectional association

A **unidirectional** association is drawn as an unbroken line with an open arrowhead pointing from the knowing class to the known class, as shown in Figure 2.16. In this case, on your road trip through Arizona you might run across a speed trap where a speed cam records your driving activity, but you won't know about it until you get notification in the mail. It isn't drawn in the image but in this case the multiplicity value would be `0..*` depending on how many times you drive by the speed cam.

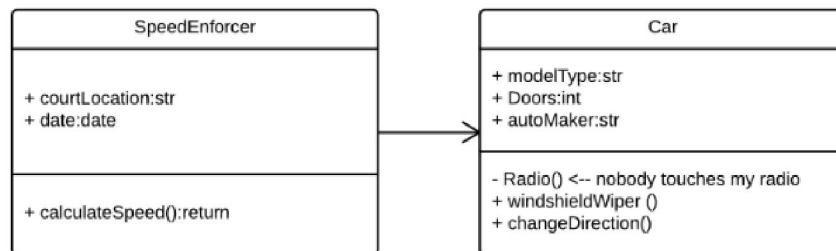


Figure 2.16: Unidirectional association

## REFERENCESS

- [1] I. Sommerville , 2011. Software Engineering. Ninth Edition, Addison-Wesley.
- [2] <https://www.lucidchart.com/pages/data-flow-diagram>
- [3] <https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/DFDs/>
- [4] [https://www.tutorialspoint.com/uml/uml\\_statechart\\_diagram.htm](https://www.tutorialspoint.com/uml/uml_statechart_diagram.htm)
- [5] <https://www.lucidchart.com/pages/uml/class-diagram>

### 3. ARCHITECTURAL DESIGN

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system. It is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

Figure 3.1, shows an abstract model of the architecture for a packing robot system that shows the components that have to be developed. This robotic system can pack different kinds of object. It uses a vision component to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor. The architectural model shows these components and the links between them.

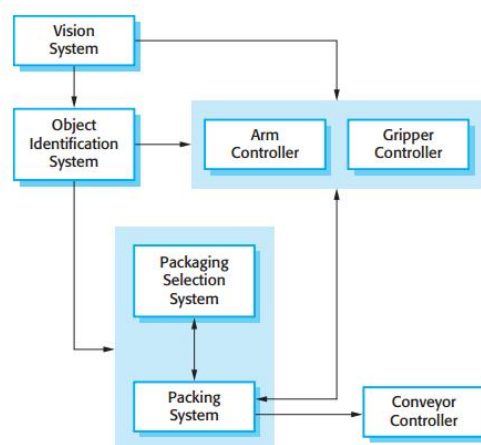


Figure 3.1: The architecture of a packing robot control system [1]

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not include any design information. This is unrealistic except for very small systems. Architectural decomposition is usually necessary to structure and organize the specification. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or sub-systems. You can then use this decomposition to discuss the requirements and features of the system with stakeholders.

You can design software architectures at two levels of abstraction, which are:

1. ***Architecture in the small*** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
2. ***Architecture in the large*** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system. Individual components implement the functional system requirements. The nonfunctional requirements depend on the system architecture—the way in which these components are organized and communicate. In many systems, non-functional requirements are also influenced by individual components, but there is no doubt that the architecture of the system is the dominant influence.

The three advantages of explicitly designing and documenting software architecture:

1. ***Stakeholder communication.*** The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. ***System analysis.*** Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, and maintainability.
3. ***Large-scale reuse.*** A model of system architecture is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. It may be possible to develop product-line architectures where the same architecture is reused across a range of related systems.

System architectures are often modeled using simple block diagrams, as in Figure 3.1. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to sub-components. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows.

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand. Block diagrams are an appropriate way of describing the system architecture during the design process, as they are a good way of supporting communications between the people involved in the process. In many projects, these are often the only architectural documentation that exists. *However, if the architecture of a system is to be*

*thoroughly documented then it is better to use a notation with well-defined semantics for architectural description.*

The apparent contradictions between practice and architectural theory arise because there are two ways in which an architectural model of a program is used:

1. ***As a way of facilitating discussion about the system design.*** A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so managers can start assigning people to plan the development of these systems.
2. ***As a way of documenting an architecture that has been designed.*** The aim here is to produce a complete system model that shows the different components in a system, their interfaces, and their connections. The argument for this is that such a detailed architectural description makes it easier to understand and evolve the system.

### 3.1. Architectural Design Decisions

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process, the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. It is therefore useful to think of *architectural design as a series of decisions to be made rather than a sequence of activities.*

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the following fundamental questions about the system:

1. Is there a generic application architecture that can act as a template for the system that is being designed?
2. How will the system be distributed across a number of cores or processors?
3. What architectural patterns or styles might be used?
4. What will be the fundamental approach used to structure the system?
5. How will the structural components in the system be decomposed into subcomponents?
6. What strategy will be used to control the operation of the components in the system?
7. What architectural organization is best for delivering the non-functional requirements of the system?
8. How will the architectural design be evaluated?
9. How should the architecture of the system be documented?

Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain. For example, application product lines are applications that are built around a core-architecture with variants that satisfy specific customer requirements. When designing a system-architecture, you have to decide what your system and broader



application classes have in common, and decide how much knowledge from these application architectures *you can reuse*.

For embedded systems and systems designed for personal computers, there is usually only a single processor and you will not have to design a distributed architecture for the system. However, most large systems are now distributed systems in which the system software is distributed across many different computers. The choice of distribution architecture is a key decision that affects the performance and reliability of the system.

The architecture of a software system may be based on a particular architectural pattern or style. An architectural pattern is a description of a system organization, such as a ***client-server organization*** or a ***layered architecture***. Architectural patterns capture the essence of an architecture that has been used in different software systems. You should be aware of common patterns, where they can be used, and their strengths and weaknesses when making decisions about the architecture of a system. You have to choose the most appropriate structure, such as client-server or layered structuring that will enable you to meet the system requirements. To decompose structural system units, you decide on the strategy for decomposing components into sub-components. The approaches that you can use allow different types of architecture to be implemented. Finally, in the control modeling process, you make decisions about how the execution of components is controlled. You develop a general model of the control relationships between the various parts of the system.

Because of the close relationship between non-functional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the non-functional system requirements:

- ***Performance*** If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, fine-grain components, which reduces the number of component communications. You may also consider run-time system organizations that allow the system to be replicated and executed on different processors.
- ***Security*** If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.
- ***Safety*** If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.
- ***Availability*** If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. For example, fault-tolerant system architectures are high-availability systems.
- ***Maintainability*** If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

Obviously there is potential conflict between some of these architectures. For example, using large components improves performance and using small, fine-grain components improves maintainability. If both performance and maintainability are important system requirements, then some compromise must be found. This can sometimes be achieved by using different architectural patterns or styles for different parts of the system.

*Evaluating an architectural design is difficult because the true test of architecture is how well the system meets its functional and non-functional requirements when it is in use. However, you can do some evaluation by comparing your design against reference architectures or generic architectural patterns.*

### 3.2. System Organization

The organization of a system reflects the basic strategy that is used to structure a system. You have to make decisions on the overall organizational model of a system early in the architectural design process. The system organization may be directly reflected in the sub-system structure. However, it is often the case that the sub-system model includes more detail than the organizational model, and there is not always a simple mapping from sub-systems to organizational structure.

The three organizational styles are widely used:

- A shared data repository style
- A shared services and server style
- An abstraction machine or layered style

These organizational styles can be used separately or together, for example, a system may be organized around a shared data repository but may construct layers around this to represent a more abstract view of the data.

#### THE REPOSITORY MODEL

Sub-systems making up a system must exchange information so that they can work together efficiently. There are two fundamental ways in which this can be done:

1. All shared data is held in a central database that can be accessed by all sub-systems. A system model based on a shared database is sometimes called a **repository model**.
2. Each sub-system maintains its own database. Data is interchanged with other sub-systems by passing message to them.

When large amounts of data are to be shared, the repository model of sharing is most commonly used.

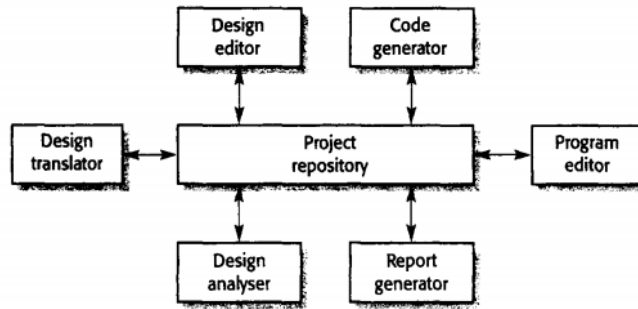


Figure 3.2: Architecture of an integrated CASE toolset [2]

The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications where data is generated by one sub-system and used by another. Examples of this type of system include command and control systems, management information systems, CAD systems, and CASE toolsets (see Figure 3.2). In this model, *the repository is passive and control is the responsibility of the sub-systems using the repository.*

#### Advantages:

- It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.
- Sub-systems that produce data need not be concerned with how data is used by other sub-systems.
- Activities such as backup, security, access control and recovery from error are centralized. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issues.
- The model of having is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.

#### Disadvantages

- Sub-systems must agree on a repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.
- Evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive; it may be difficult or even impossible.
- Different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.
- It may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.

#### THE CLIENT- SERVER MODEL

The client-server architectural model is a system model where the system is organized as set of services and associated servers and clients that access and use the services. The major components of this model, as shown in Figure 3.3, are:

- A set of servers that offer services to other sub-systems. Examples of servers are print servers that offer printing services, file servers that offer file management services and a compile server, which offers programming language compilation services.
- A set of clients that call on the services offered by servers. These are normally sub-systems in their own right. There may be several instances of a client program executing concurrently.
- A network that allows the clients to access these services. This is not strictly necessary as both the clients and the servers could run on a single machine. In practice, however, most client-server systems are implemented as distributed systems.

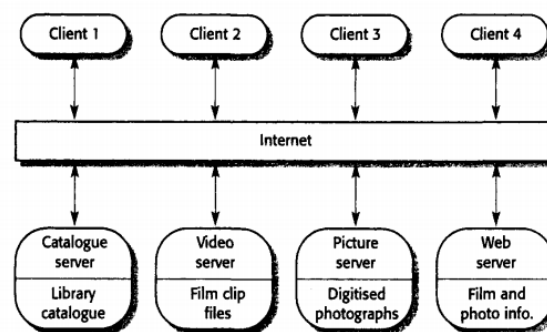


Figure 3.3: Architecture of a film and picture library [2]

Clients may have to know the names of the available servers and the services that they provide. However, servers need not know either the identity of clients or how many clients there are. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as the http protocol used in the WWW. Essentially, a client makes a request to a server and waits until it receives a reply.

### Advantages

- The client-server model is a distributed architecture
- It makes effective use of networked system with many distributed processors
- It is easy to add new servers and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system

### Disadvantages

- Changes to existing clients and servers may be required to gain the full benefits of integrating a new server
- There may be no shared data model across servers and sub-systems may organize their data in different ways. This means that specific data models may be established on each server to allow its performance to be optimized.

## ABSTRACT MACHINE (LAYERED) MODEL

The layered model of an architecture (sometimes called an *abstract machine model*) organizes a system into layers, each of which provides a set of services, shown in Figure 3.4. Each layer can be thought of as an abstract machine whose machine language is defined by the services provided by the layer. This 'language' is used to implement the next level of abstract machine. For example, a common way to implement a language is to define an ideal '*language machine*' and compile the language into code for this machine. A further translation step then converts this abstract machine code to real machine code.

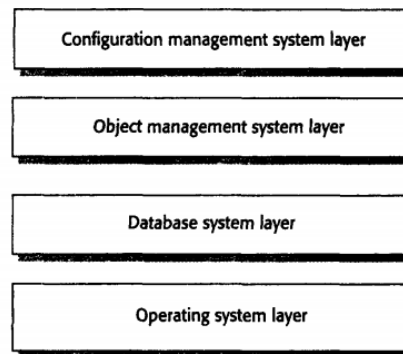


Figure 3.4: Layered model of a version management system [2]

### Advantages

- The layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users.
- This architecture is also changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer. Furthermore, when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected.
- As layered systems localize machine dependencies in inner layers, this makes it easier to provide multi-platform implementations of an application system. Only the inner, machine-dependent layers need be re-implemented to take account of the facilities of a different operating system or database.

### Disadvantages

- Structuring systems in this way can be difficult. Inner layers may provide basic facilities, such as file management, that are required at all levels. Services required by a user of the top level may therefore have to '*punch through*' adjacent layers to get access to services that are provided several levels beneath it. This subverts the model, as the outer layer in the system does not just depend on its immediate predecessor.
- Performance can also be a problem because of the multiple levels of command interpretation that are sometimes required. If there are many layers, a service request from a top layer may have to be interpreted several times in different layers before it is processed. To avoid these problems, applications may have to communicate directly with inner layers rather than use the services provided by the adjacent layer.

### 3.3. Modular Decomposition Styles

After an overall system organization has been chosen, you need to make a decision on the approach to be used in decomposing sub-systems into modules. There is not a rigid distinction between system organization and modular decomposition. The styles like repository model, client-server model, and layered model, could be applied at this level. However, the components in modules are usually smaller than sub-systems, which allow alternative decomposition styles to be used.

There is no clear distinction between sub-systems and modules, but to think them as follows can be useful:

1. *A sub-system is a system in its own right whose operation does not depend on the services provided by other sub-systems.* Sub-systems are composed of modules and have defined interfaces, which are used for communication with other sub-systems.
2. *A module is normally a system component that provides one or more services to other modules.* It makes use of services provided by other modules. It is not normally considered to be an independent system. Modules are usually composed from a number of other simpler system components.

There are two main strategies that you can use when decomposing a sub-system into modules:

#### OBJECT-ORIENTED DECOMPOSITION

In Object-Oriented Decomposition, a system is decomposed into a set of communicating objects, as shown in Figure 3.5. An object-oriented, architectural model structures the system into a set of loosely coupled objects with well-defined interfaces. In the object oriented approach, modules are objects with private state and defined operations on that state. An object-oriented decomposition is concerned with object classes, their attributes and their operations. When implemented, objects are created from these classes and some control model is used to coordinate object operations.

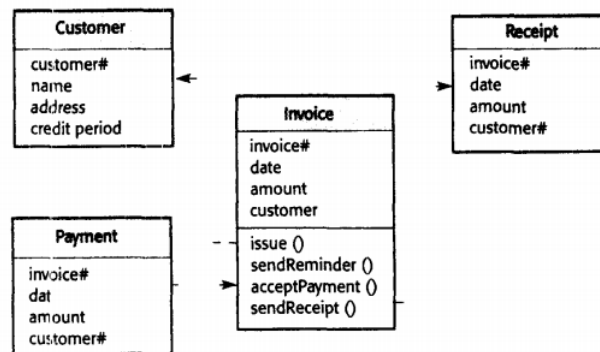


Figure 3.5: An object model of an invoice processing system [2]

#### Advantages

- In object-oriented, the objects are loosely coupled, the implementation of objects can be modified without affecting other objects.
- Objects are often representations of real-world entities so the structure of the system is readily understandable

- Objects are often representations of real-world entities, and the real-world entities are used in different systems, objects can be reused.
- Object-oriented programming languages have been developed that provide direct implementations of architectural components.

### Disadvantages

- To use services, objects must explicitly reference the name and the interface of other objects. If an interface change is required to satisfy proposed system changes, the effect of that change on all users of the changed object must be evaluated.
- Objects may map cleanly to small-scale real-world entities, more complex entities are sometimes difficult to represent as objects.

## FUNCTION-ORIENTED PIPELINING

In Function-Oriented Pipeline or data flow model, a system is decomposed into functional modules that accept input data and transform it into output data, as shown in Figure 3.6. In this model, modules are functional transformations, i.e., they process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch. When the transformations are represented as separate processes, this model is sometimes called the *pipe and filter style*.

Variants of this pipelining model have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this architectural model is a *batch sequential model*. Data-processing systems usually generate many output reports that are derived from simple computations on a large number of input records.

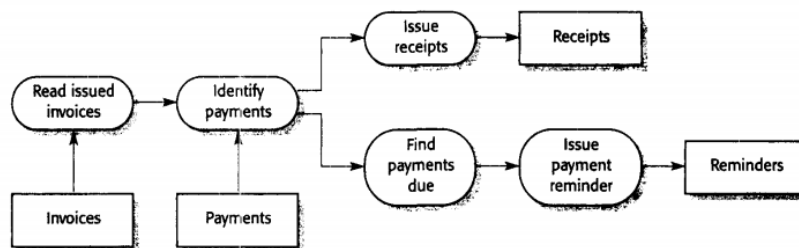


Figure 3.6: A pipeline model of an invoice processing system [2]

**Note:** The difference between this and its object-oriented equivalent discussed in the previous section. The object model is more abstract as it does not include information about the sequence of operations

### Advantages

- It supports the reuse of transformations.
- It is intuitive in that many people think of their work in terms of input and output processing.
- Evolving the system by adding new transformations is usually straight forward.

- It is simple to implement either as a concurrent or a sequential system.

### Disadvantages

- There has to be a common format for data transfer that can be recognized by all transformations. Each transformation must either agree with its communicating transformations on the format of the data that will be processed or with a standard format for all data communicated must be imposed. The latter is the only feasible approach when transformations are standalone and reusable.
- Interactive systems are difficult to write using the pipelining model because of the need for a stream of data to be processed. While simple textual input and output can be modeled in this way, graphical user interfaces have more complex I/O formats and control, which is based on events such as mouse clicks or menu selections. It is difficult to translate this into a form compatible with the pipelining model.

## 3.4. Control Styles

The models for structuring a system are concerned with how a system is decomposed into sub-systems. To work as a system, sub-systems must be controlled so that their services are delivered to the right place at the right time. Structural models do not (and should not) include control information. Rather, the architect should organize the sub-systems according to some control model that supplements the structure model that is used. Control models at the architectural level are concerned with the control flow between sub-systems.

There are two generic control styles that are used in software systems:

### CENTRALIZED CONTROL

One sub-system has overall responsibility for control and starts and stops other sub-systems. It may also devolve control to another sub-system but will expect to have this control responsibility returned to it.

In a centralized control model, one sub-system is designated as the system controller and has responsibility for managing the execution of other sub-systems. Centralized Control models fall into two classes, depending on whether the controlled sub-systems execute sequentially or in parallel.

1. ***The call-return model.*** This is the familiar top-down subroutine model where control starts at the top of a subroutine hierarchy and, through subroutine calls, passes to lower levels in the tree, shown in Figure 3.7. The subroutine model is only applicable to sequential systems.

This familiar model is embedded in programming languages such as C, Ada and Pascal. Control passes from a higher-level routine in the hierarchy to a lower-level routine, it then returns to the point where the routine was called. The currently executing subroutine has responsibility for control and can either call other routines or return control to its parent. It is poor programming style to return to some other point in the program.



This, call-return model may be used at the module level to control functions or objects" Subroutines in a programming language that are called by other subroutines are naturally functional. However, in many object-oriented systems, operations on objects (methods) are implemented as procedures or functions. For example, when a Java object requests a service from another object, it does so by calling an associated method.

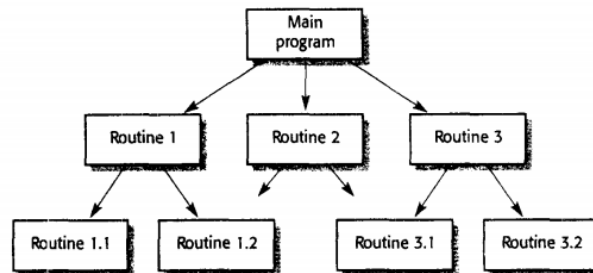


Figure 3.7: The call-return model of control [2]

The rigid and restricted nature of this model is both a strength and a weakness. It is a **strength** because it is relatively simple to analyze control flows and work out how the system will respond to particular inputs. It is a **weakness** because exceptions to normal operation are awkward to handle.

2. **The manager model.** This is applicable to concurrent systems. One system component is designated as a system manager and controls the starting, stopping and coordination of other system processes. A process is a sub-system or module that can execute in parallel with other processes. A form of this model may also be applied in sequential systems where a management routine calls particular sub-systems depending on the values of some state variables. This is usually implemented as a case statement.

This model is often used in 'soft' real-time systems which do not have very tight time constraints, as shown in Figure 3.8. The central controller manages the execution of a set of processes associated with sensors and actuators.

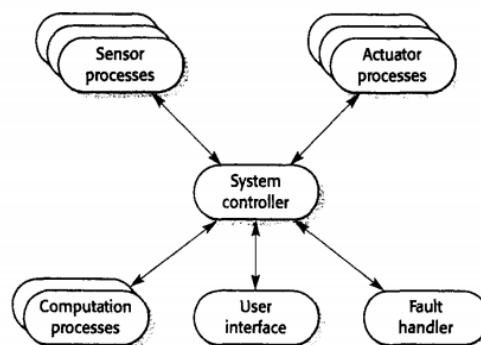


Figure 3.8: A centralized control model for a real-time system [2]

The system controller process decides when processes should be started or stopped depending on system state variables. It checks whether other processes have produced information to be processed or to pass information to them for processing. The controller usually loops continuously, polling sensors and other processes for events or state changes. For this reason, this model is sometimes called an *event loop model*.

## EVENT-BASED CONTROL

Rather than control information being embedded in a subsystem, each sub-system can respond to externally generated events. These events might come from other sub-systems or from the environment of the system. Control styles are used in conjunction with structural styles.

In centralized control models, control decisions are usually determined by the values of some system state variables. By contrast, event-driven control models are driven by externally generated events. The term event in this context does not just mean a binary signal. It may be a signal that can take a range of values or a command input from a menu. The distinction between an event and a simple input is that the timing of the event is outside the control of the process that handles that event.

There are many types of event-driven systems. These include editors where user interface events signify editing commands, rule-based production systems as used in AI where a condition becoming true causes an action to be triggered, and active objects where changing a value of an object's attribute triggers some actions.

Two event-driven control models are:

1. **Broadcast models.** In these models, an event is broadcast to all sub-systems, shown in Figure 3.9. Any sub-system that has been programmed to handle that event can respond to it. Broadcast models are effective in integrating sub-systems distributed across different computers on a network.

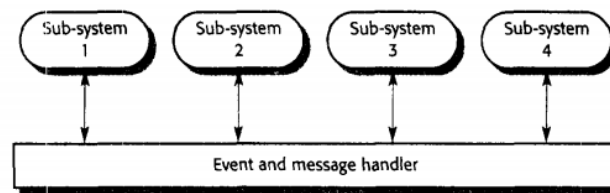


Figure 3.9: A control model based on selective broadcasting [2]

In a broadcast model, sub-systems register an interest in specific events. When these events occur, control is transferred to the sub-system that can handle the event. The distinction between this model and the centralized model is that the control policy is not embedded in the event and message handler. Sub-systems decide which events they require, and the event and message handler ensures that these events are sent to them.

All events could be broadcast to all sub-systems, but this imposes a great deal of processing overhead. More often, the event and message handler maintains a register of sub-systems and the events of interest to them. Sub-systems generate events indicating, perhaps, that some data is

available for processing. The event handler detects the events, consults the event register and passes the event to those subsystems who have declared an interest. In simpler systems, such as PC-based systems driven by user interface events, there are explicit event-listener sub-systems that listen for events from the mouse, the keyboard, and so on, and translate these into more specific commands.

The event handler also usually supports point-to-point communication. A subsystem can explicitly send a message to another sub-system. There have been a number of variations of this model, such as the Fieldenvironment and Hewlett-Packard's Softbench .Both of these have been used to control tool interactions in software engineering environments.

### Advantages

*Evolution is relatively simple in broadcast approach.* A new sub-system to handle particular classes of events can be integrated by registering its events with the event handler. Any sub-system can activate any other sub-system without knowing its name or location. The sub-systems can be implemented on distributed machines. This distribution is transparent to other subsystems.

### Disadvantages

*Sub-systems don't know if or when events will be handled in broadcast approach.* When a sub-system generates an event it does not know which other sub-systems have registered an interest in that event. It is quite possible for different sub-systems to register for the same events. This may cause conflicts when the results of handling the event are made available.

2. **Interrupt-driven models.** These are exclusively used in real-time systems where external interrupts are detected by an interrupt handler. They are then passed to some other component for processing. Interrupt-driven models are used in real-time systems with stringent timing requirements, shown in Figure 3.10.

Real-time systems that require externally generated events to be handled very quickly must be event-driven. For example, if a real-time system is used to control the safety systems in a car, it must detect a possible crash and, perhaps, inflate an air bag before the driver's head hits the steering wheel. To provide this rapid response, to events you have to use interrupt-driven control.

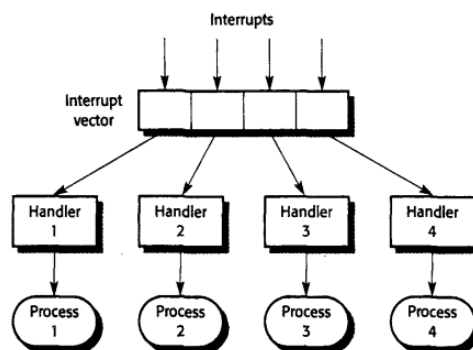


Figure 3.10: An interrupt-driven control mode [2]

There are a known number of interrupt types with a handler defined for each type. Each type of interrupt is associated with the memory location where its handler's address is stored. When an interrupt of a particular type is received, a hardware switch causes control to be transferred immediately to its handler. This interrupt handler may then start or stop other processes in response to the event signaled by the interrupt. This model is mostly used in real-time systems where an immediate response to some event is necessary. It may be combined with the centralized management model. The central manager handles the normal running of the system with interrupt-based control for emergencies.

### **Advantages**

*It allows very fast responses to events to be implemented.*

### **Disadvantages**

*It is complex to program and difficult to validate.* It may be impossible to replicate patterns of interrupt timing during system testing. It can be difficult to change systems developed using this model if the number of interrupts is limited by the hardware. Once this limit is reached, no other types of events can be handled. You can sometimes get around this limitation by mapping several types of events onto a single interrupt. The handler then works out which event has occurred. However, interrupt mapping may be impractical if a very fast response to individual interrupts is required.

## **3.5. Reference Architectures**

General architectural models can be applied to many classes of application. These general models, architectural models that are specific to a particular application domain may also be used. Although instance of these systems differ in detail, the common architectural structure can be reused when developing new systems. These architectural models are called domain specific architectures.

There are two types of *domain-specific architectural* model:

1. **Generic models** are abstractions from a number of real systems. They encapsulate the principal characteristics of these systems. For example, in real-time systems, there might be generic architectural models of different system types such as data collection systems or monitoring systems.
2. **Reference models** are more abstract and describe a larger class of systems. They are away of informing designers about the general structure of that class of system. Reference models are usually derived from a study of the application domain. They represent an idealized architecture that includes all the features that systems might incorporate.

There is not, of course, a rigid distinction between these types of model. *Generic models can also serve as reference models.*

*Reference models* are normally used to communicate domain concepts and compare or evaluate possible architectures. They are not normally considered a route to implementation. Rather, their principal function is a means of discussing domain-specific architectures and comparing different systems in a domain. A

reference model provides a vocabulary for comparison. It acts as a base, against which systems can be evaluated.

A proposed reference model is a reference model for CASE environments that identifies five sets of services that a CASE environment should provide. It should also provide *'plug-in'* facilities for individual CASE tools that use these services. The five levels services in the CASE reference model are illustrated in Figure 3.11.

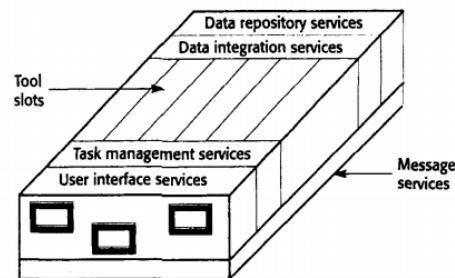


Figure 3.11: The ECMA reference architecture for CASE environments [2]

This reference model tells us what might be included in any particular CASE environment, although it is important to emphasize that not every feature of reference architecture will be included in actual architectural designs. It means that we can ask questions of a system design such as “*how are the data repository services provided?*” and “*does the system provide task management?*”

### 3.6. Multiprocessor Architectures

The simplest model of a distributed system is a multiprocessor system where the software system consists of a number of processes that may (but need not) execute on separate processors. This model is common in large real-time systems. These systems collect information, make decisions using this information on and send signals to actuators that modify the system's environment.

Logically, the processes concerned with information collection, decision making and actuator control could all run on a single processor under the control of a scheduler. Using multiple processors improves the performance and resilience of the system. The distribution of processes to processors may be pre-determined (this is common in critical systems) or may be under the control of a dispatcher that decides which process to allocate to each processor.

An example of this type of system is shown in Figure 3.12. This is a simplified model of a traffic control system. A set of distributed sensors collects information on the traffic flow and processes this locally before sending it to a control room. Operators make decisions using this information and give instructions to a separate traffic light control process. In this example, there are separate logical processes for managing the sensors, the control room and the traffic lights. These logical processes may be single processes or a group of processes. In this example, they run on separate processors.

Software systems composed of multiple processes are not necessarily distributed systems. If more than one processor is available, then distribution can be implemented, but the system designers need not always consider distribution issues during the design process. The design approach for this type of system is essentially that for real-time systems.

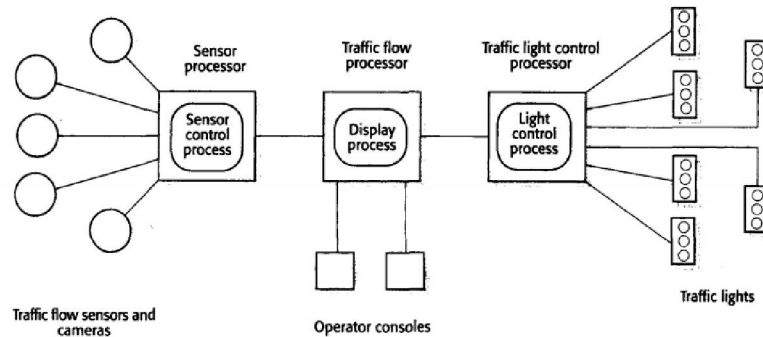


Figure 3.12: A multiprocessor traffic control system [2]

### 3.7. Client-Server Architectures

In client-server architecture, an application is modeled as a set of services that are provided by servers and a set of clients that use these services, as shown in Figure 3.13 and 3.14. Clients need be aware of the servers that are available but usually do not know of the existence of other clients. Clients and servers are separate processes, which is a logical model of a distributed client-server architecture. Several server processes can run on a single server processor so there is *not necessarily a 1:1 mapping* between processes and processors in the system.

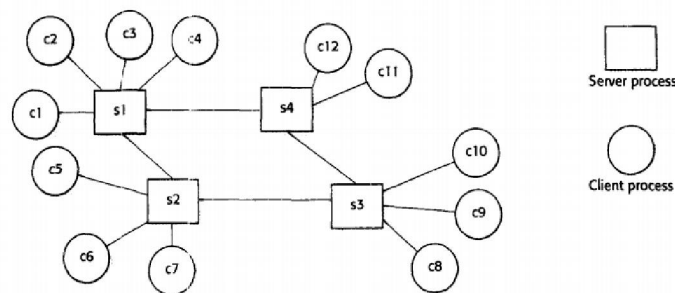


Figure 3.13: A client-server system [2]

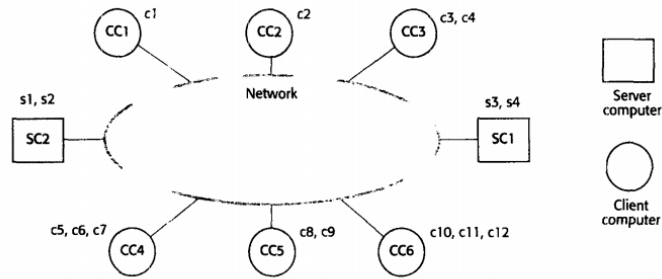


Figure 3.14: Computers in client-server network [2]

The simplest client-server architecture is called a *two-tier client-server architecture*, where an application is organized as a server (or multiple identical servers) and a set of clients. Two-tier client-server architectures can take two forms:

1. **Thin-client model.** In a thin-client model (shown in Figure 3.15), all of the application processing and data management is carried out on the server. The client is simply responsible for running the presentation software.

A major disadvantage of the thin-client model is that it places a heavy processing load on both the server and the network. The server is responsible for all computation, and this may involve the generation of significant network traffic between the client and the server. There is a lot of processing power available in modern computing devices, which is largely unused in the thin-client approach.

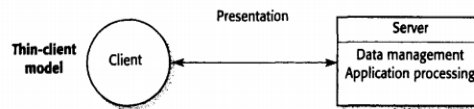


Figure 3.15: Thin-client model [2]

2. **Fat-client model.** In this model, the server is only responsible for data management (shown in Figure 3.16). The software on the client implements the application logic and the interactions with the system user. The fat-client model makes use of processing power available in the modern computing devices and distributes both the application logic processing and the presentation to the client. The server is essentially a transaction server that manages all database transactions. While the fat-client model distributes processing more effectively than a thin client model, system management is more complex. Application functionality is spread across many computers. When the application software has to be changed, this involves reinstallation on every client computer. This can be a major cost if there are hundreds of clients in the system. An example of this type of architecture is banking ATM systems, where the ATM is the client and the server is a mainframe running the customer account database. The hardware in the teller machine carries out a lot of the customer-related processing associated with a transaction.

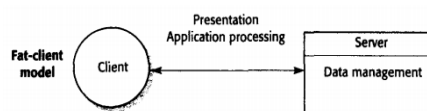


Figure 3.16: Fat-client model [2]

The advent of mobile code (such as Java applets and ActiveX controls), which can be downloaded from a server to a client, has allowed the development of client-server systems that are somewhere between the thin- and the fat-client model. Some of the application processing software may be downloaded to the client as mobile code, thus relieving the load on the server. The user interface is created using a web browser that has built-in facilities to run the downloaded code.

The problem with a two-tier client-server approach is that the three logical layers-presentations, application processes and data management-must be mapped on to two computer systems- the client and the server. There may either be problems with scalability and performance if the thin-client model is chosen, or problems of system management if the fat-client model is used. To avoid these issues, an alternative approach is to use a *three-tier client-server architecture*.

In a *three-tier client-server architecture*, the presentation, the application processing and the data management are logically separate processes that execute on different processors. An Internet banking system is an example of the three-tier client-server architecture.

In some cases, it is appropriate to extend the three-tier client-server model to a *multi-tier variant* where additional servers are added to the system. Multi-tier systems may be used where applications need to access and use data from different databases. In this case, an integration server is positioned between the application server and the database servers. The integration server collects the distributed data and presents it to the application as if it were from a single database.

Three-tier client-server architectures and multi-tier variants that distribute the application processing across several servers are inherently more scalable than two-tier architectures. Network traffic is reduced in contrast with thin-client two-tier architectures. The application processing is the most volatile part of the system, and it can be easily updated because it is centrally located. Processing, in some cases, may be distributed between the application logic and the data management servers, thus leading to more rapid response to client requests.

| Architecture                                | Applications  |
|---|---|
| Two-tier C/S Architecture with Thin Clients | <ul style="list-style-type: none"> <li>• Legacy system applications where separating application processing and data management is impractical.</li> <li>• Computationally-intensive applications such as compilers with little or no data management.</li> <li>• Data-intensive applications(browsing and querying)with little or</li> <li>• No application processing.</li> </ul>   |
| Two-tier C/S Architecture with Fat Clients  | <ul style="list-style-type: none"> <li>• Applications where application processing is provided by off-the shelf software (e.g. Microsoft Excel) on the client.</li> <li>• Applications where computationally-intensive processing of data (e.g. data visualization) is required.</li> <li>• Applications with relatively stable end-user functionality used in an environment with well-established system management.</li> </ul> |
| Three-tier or Multi-tier C/S Architecture   | <ul style="list-style-type: none"> <li>• Large-scale applications with hundreds or thousands of clients.</li> <li>• Applications where both the data and the application are volatile.</li> <li>• Applications where data from multiple sources are integrated.</li> </ul>  |

Table 3.1: Use of different client-server architectures [2]



### 3.8. Distributed Object Architectures

In the client-server model of a distributed system, clients and servers are different, shown in Figure 3.17. Clients receive services from servers and not from other clients; servers may act as clients by receiving services from other servers but they do not request services from clients; clients must know the services that are offered by specific servers and must know how to contact these servers. This model works well for many types of applications. However, it limits the flexibility of system designers in that they must decide where services are to be provided. They must also plan for scalability and so provide some means for the load on servers to be distributed as more clients are added to the system.

A more general approach to distributed system design is to remove the distinction between client and server and to design the system architecture as a distributed object architecture. In a distributed object architecture, the fundamental system components are objects that provide an interface to a set of services that they provide. Other objects call on these services with no logical distinction between a client (a receiver of a service) and a server (a provider of a service).

Object may be distributed across a number of computers on a network and communicate through middleware. This middleware is called an object request broker. Its role is to provide a seamless interface between objects. It provides a set of services that allow objects to communicate and to be added to and removed from the system. Its role is to provide a seamless interface between objects. It provides a set of services that allow objects to communicate and to be added to and removed from the system.

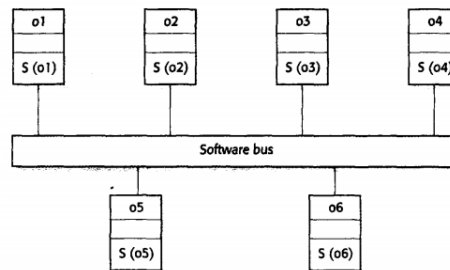


Figure 3.17: Distributed object architecture [2]

The **advantages** of the distributed object model are:

- It allows the system designer to delay decisions on where and how services should be provided. Service-providing objects may execute on any node of the network. Therefore, the distinction between fat- and thin-client models becomes irrelevant, as there is no need to decide in advance where application logic objects are located.
- It is a very open system architecture that allows new resources to be added to it as required.
- The system is flexible and scalable. Different instances of the system with the same service provided by different objects or by replicated objects can be created to cope with different system loads. New objects can be added as the load on the system increases without disrupting other system objects.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required. This may be important where there are fluctuating patterns of demand on services. A

service-providing object can migrate to the same process or as service-requesting objects, thus improving the performance of the system.

Distributed object architecture can be used as a logical model that allows you to structure and organize the system. In this case, you think about how to provide application functionality solely in terms of services and combination of services. You then workout how to provide these services using a number of distributed objects. At this level, the objects that you design are usually large-grain objects (sometimes called business objects) that provide domain-specific services. For example, in a retail application, there may be business objects concerned with stock control, customer communications, goods-ordering, and so on. This logical model can, of course, then be realized as an implementation model.

Alternatively, you can use a distributed object approach to implement client-server systems. In this case, the logical model of the system is a client-server model, but both clients and servers are realized as distributed objects communicating through a software bus. This makes it possible to change the system easily, for example, from a two-tier to a multi-tier system. In this case, the server or the client may not be implemented as a single distributed object but may be composed from smaller objects that provide specialized services.

An example of a type of system where a distributed object architecture: might be appropriate is a data mining system that looks for relationships between the data that is stored in a number of databases

The major **disadvantage** of distributed object architectures is that they are more complex to design than client-server systems. Client-server systems appear to be a fairly natural way to think about systems. They reflect many human transactions where people request and receive services from other people who specialize in providing these services. It is more difficult to think about general service provision, and we do not yet have a great deal of experience with the design and development of large-grain business objects.

### 3.9. Inter-Organizational Distributed Computing

For reasons of security and inter-operability, distributed computing has been primarily implemented at the organizational level. An organization has a number of servers and spreads its computational load across these. Because these are all located within the same organization, local standards and operational processes can be applied. Although, for web-based systems, client computers are often outside the organizational boundary, their functionality is limited to running user interface software.

Newer models of distributed computing, however, are now available that allow inter-organizational rather than intra-organizational distributed computing. The two approaches of inter-organizational distributed computing are:

#### **PEER-TO-PEER (P2P) ARCHITECTURES**

Peer-to-Peer (P2P) systems are decentralized systems where computations may be carried out by any node on the network and, in principle at least, no distinctions are made between clients and servers. In peer-to-peer applications, the overall system is designed to take advantage of the computational power and storage available across a potentially huge network of computers. The standards and protocols that

enable communications across the nodes are embedded in the application itself, and each node must run a copy of that application. For example, file-sharing systems based on the Gnutella and Kazaa protocols use P2P architectures.

You can look at the architecture of P2P applications from two perspectives. The logical network architecture is the distribution architecture of the system, whereas the application architecture is the generic organization of components in each application type.

In principle, in peer-to-peer systems every node in the network could be aware of every other node, could make connections to it, and could exchange data with it. In practice, of course, this is impossible, so nodes are organized into 'localities' with some nodes acting as bridges to other node localities,

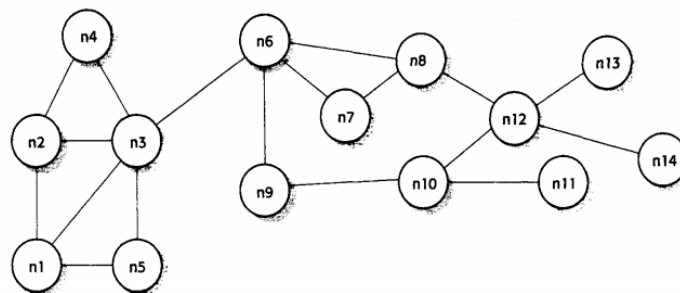


Figure 3.18: Decentralized P2P architecture [2]

In a decentralized architecture, shown in Figure 3.18, the nodes in the network are not simply functional elements but are also communications switches that can route data and control signals from one node to another.

This decentralized architecture has obvious **advantages** in that it is highly redundant, and so is fault-tolerant and tolerant of nodes disconnecting from the network. However, there are obvious **overheads** in the system in that the same search may be processed by many different nodes and there is significant overhead in replicated peer communications.

An alternative P2P architectural model that departs from pure P2P architecture is a **semi-centralized architecture (or hybrid P2P)** where, within the network, one or more nodes act as servers to facilitate node communications. In a semi-centralized architecture, the role of a server is to help establish contact between peers in the network or to coordinate the results of a computation.

In a computational P2P system where a processor-intensive computation is distributed across a large number of nodes, it is normal for some nodes to be distinguished nodes whose role is to distribute work to other nodes and to collate and check the results of the computation.

Although there are obvious overheads in peer-to-peer systems, it is a much more efficient approach to inter-organizational computing than the service-based approach. There are still problems with using p2p approaches for inter-organizational computing, as issues such as security and trust are still unresolved. This means that P2P systems are most likely to be used either for non-critical information systems or where there are already working relationships between organizations.

## SERVICE-ORIENTED SYSTEM ARCHITECTURE (SOA)

The development of the WWW meant that client computers had access to remote servers outside their own organizations. If these organizations converted their information to HTML, then this could be accessed by these computers. However, access was solely through a web browser and direct access to the information stores by other programs was not practical. This meant that opportunistic connections between servers where, for example, a program queried a number of catalogues was not possible.

To get around this problem, the notion of a web service was proposed. Using a web service, organizations that want to make their information accessible to other programs can do so by defining and publishing a web service interface. This interface defines the data available and how it can be accessed. More generally, a web service is a standard representation for some computational or information resource that can be used by other programs.

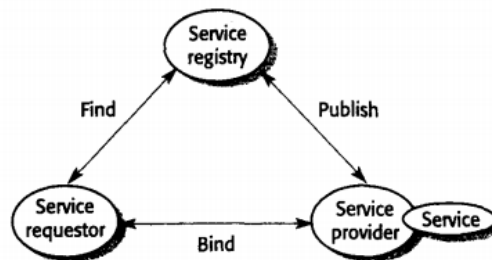


Figure 3.19: The conceptual architecture of a service-oriented system [2]

The differences between service model and the distributed object approach to distributed systems architectures are:

- Services can be offered by any service provider inside or outside of an organization. Assuming these conform to certain standards, organizations can create applications by integrating services from a range of providers.
- The service provider makes information about the service public so that any authorized user can use it. The service provider and the service user do not need to negotiate about what the service does before it can be incorporated in an application program.
- Applications can delay the binding of services until they are deployed or until execution.
- Opportunist construction of new services is possible. A service provider may recognize new services that can be created by linking existing services in innovative ways.
- Service users can pay for services according to their use rather than their provision. Therefore, instead of buying an expensive component that is rarely used, the application writer can use an external service that will be paid for only when required.
- Applications can be made smaller (which is important if they are to be embedded in other devices) because they can implement exception handling as external services.
- Applications can be reactive and adapt their operation according to their environment by binding to different services as their environment changes.

The three fundamental standards that enable communications between web services are:

1. **SOAP (Simple Object Access Protocol)**. This protocol defines an organization for structured data exchange between web services.
2. **WSDL (Web Services Description Language)**. This protocol defines how the interfaces of web services can be represented.
3. **UDDI (Universal Description, Discovery and Integration)**. This is a discovery standard that defines how service description information, used by service requestors to discover services, can be organized.

All of these standards are based on XML—a human-and machine-readable markup language.

Web service application architectures are loosely coupled architectures where service bindings can change during execution. Some systems will be solely built using web services and others will mix web services with locally developed components.

## REFERENCES

- [1] I. Sommerville , 2011. Software Engineering. Ninth Edition, Addison-Wesley.
- [2] I. Sommerville , 2009. Software Engineering. Eighth Edition, Addison-Wesley.

## 4. REAL-TIME SOFTWARE DESIGN

Computers are used to control a wide range of systems from simple domestic machines to entire manufacturing plants. These computers interact directly with hardware devices. The software in these systems is embedded real-time software that must react to events generated by the hardware and issue control signals in response to these events.

It is embedded in some larger hardware system and must respond, *in real time*, to events from the system's environment. Real-time embedded systems are different from other types of software systems. Their correct functioning is dependent on the system responding to events within a short time interval.

*A **real-time system** is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced. A soft real-time system is a system whose operation is **degraded** if results are not produced according to the specified timing*

requirements. A **hard real-time system** is a system whose operation is **incorrect** if results are not produced according to the timing specification.

Timely response is an important factor in all embedded systems but, in some cases, very fast response is not necessary. For example, the insulin pump system (embedded system), while it needs to check the glucose level at periodic intervals, it does not need to respond very quickly to external events. .

One way of looking at a real-time system is as a **stimulus/response system**. *Given a particular input stimulus, the system must produce a corresponding response.* You can therefore define the behavior of a real-time system by listing the stimuli received by the system, the *associated responses and the time at which the response must be produced.*

Stimuli fall into two classes:

- I. **Periodic stimuli.** These occur at predictable time intervals. For example, the system may examine a sensor every 50 milliseconds and take action (respond) depending on that sensor value (the stimulus).
- II. **Aperiodic stimuli.** These occur irregularly. They are usually signaled using the computer's interrupt mechanism. An example of such a stimulus would be an interrupt indicating that an I/O transfer was complete and that data was available in a buffer.

Periodic stimuli in a real-time system are usually generated by sensors associated with the system. These provide information about the state of the system's environment. The responses are directed to a set of actuators that control some equipment, such as a pump, that then influences the system's environment.

Aperiodic stimuli may be generated either by the actuators or by sensors. They often indicate some exceptional condition, such as a hardware failure, that must be handled by the system.

A real-time system has to respond to stimuli that occur at different times. You therefore have to organize its architecture so that, as soon as a stimulus is received, control is transferred to the correct handler. This is impractical in sequential programs. Consequently, real-time systems are normally designed as a set of concurrent, cooperating processes. To support the management of these processes, the execution platform for most real-time systems includes **a real-time operating system**. The facilities in this operating system are accessed through the run-time support system for the real-time programming language that is used.

The generality of this stimulus-response model of a real-time system leads to a generic, abstract architectural model where there are three types of processes. For each type of sensor, there is a sensor management process; computational processes compute the required response for the stimuli received by the system; actuator control processes manage actuator operation. This model allows data to be collected quickly from the sensor (before the next input becomes available) and allows processing and the associated actuator response to be carried out later.

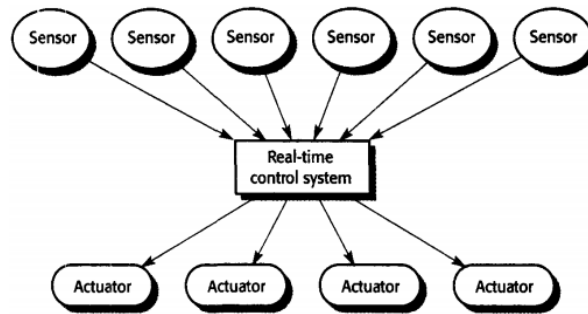


Figure 4.1: General model of a real-time system [1]

Programming languages for real-time systems development have to include facilities to access the system hardware, and it should be possible to predict the timing of particular operations in these languages. Hard real-time systems are still sometimes programmed in assembly language so that tight deadlines can be met. Systems-level languages, such as C, that allow efficient code to be generated are also widely used.

The advantage of using a low-level systems programming language such as C is that it allows the development of very efficient programs. However, *these languages do not include constructs to support concurrency* [**Meaning: Concurrency is the tendency for things to happen at the same time in a system. Concurrency is a natural phenomenon, of course. In the real world, at any given time, many things are happening simultaneously**] or the management of shared resources. These are implemented through calls to the real-time operating system that cannot be checked by the compiler, programming errors are more likely. Programs are also often more difficult to understand because real-time features are not explicit in the program.

1. Over the past few years, there has been extensive work to extend Java for real time systems development. This work involves modifying the language to address fundamental real-time problems:
2. It is not possible to specify the time at which threads [**Meaning: A Thread is the path followed when executing a program. Java is a multi-threaded application that allows multiple thread execution at any particular time**] should execute.
3. Garbage collection is uncontrollable-it may be started at any time. Therefore, the timing behavior of threads is unpredictable.
4. It is not possible to discover the sizes of queue associated with shared resources.
5. The implementation of the Java Virtual Machine varies from one computer to another, so the same program can have different timing behaviors.
6. The language does not allow for detailed run-time space or processor analysis.
7. There are no standard ways to access the hardware of the system.

#### 4.1. System Design

The system design process involves deciding which system capabilities are to be implemented in software and which in hardware. For many real-time systems embedded in consumer products, such as the systems in cell phones, the **costs** and **power consumption** of the hardware are critical. Specific processors designed to support embedded systems may be used and, for some systems, special-purpose hardware may have to be designed and built.

This means that a top-down design process-where the design starts with an abstract model that is decomposed and developed in a series of stages-is impractical for most real-time systems. Low-level decisions on hardware, support software and system timing must be considered early in the process. These limit the flexibility of system designers and may mean that additional software functionality, such as battery and power management, is required.

Events (the stimuli) rather than objects or functions should be central to the real time software design process. There are several interleaved stages in this design process:

1. Identify the stimuli that the system must process and the associated responses.
2. For each stimulus and associated response, identify the timing constraints that apply to both stimulus and response processing.
3. Choose an execution platform for the system: the hardware and the real-time operating system to be used. Factors that influence these choices include the timing constraints on the system, limitations on power available, the experience of the development team and the price target for the delivered system.
4. Aggregate the stimulus and response processing into a number of concurrent processes. A good rule of thumb in real-time systems design is to associate a process with each class of stimulus and response as shown in Figure 4.2.

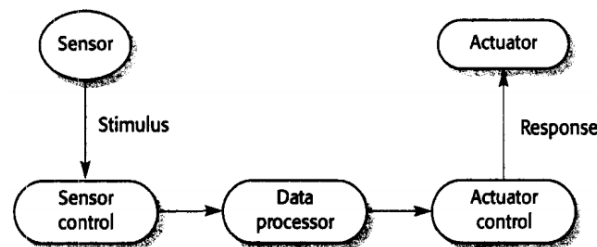


Figure 4.2: Sensor/ Actuator Control Processes [1]

5. For each stimulus and response, design algorithms to carry out the required computations. Algorithm designs often have to be developed relatively early in the design process to give an indication of the amount of processing required and the time required to complete that processing.
6. Design a scheduling system that will ensure that processes are started in time to meet their deadlines

The order of these activities in the process depends on the type of system being developed and its process and platform requirements. In some cases, you may be able to follow a fairly abstract approach where you start with the stimuli and associated processing and decide on the hardware and execution platforms late in the process. In other cases, the choice of hardware and operating system is made before the software design starts and you have to orient your design around the hardware capabilities.

Processes in a real-time system have to be coordinated. Process coordination mechanisms ensure mutual exclusion to shared resources. When one process is modifying a shared resource, other processes should not be able to change that resource.

Mechanisms for ensuring mutual exclusion includes semaphores [**Meaning:** *A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent*



*system such as a multiprogramming operating system*], monitors [**Meaning:** *In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait or block for a certain condition to become true. Monitors also have a mechanism for signaling other threads that their condition has been met*] and critical regions. These mechanisms are described in most operating system texts.

Once you have chosen the execution platform for the system, designed a process-architecture, and decided on a scheduling policy, you may need to check that the system will meet its timing requirements. You can do this through static analysis of the system using knowledge of the timing behavior of components or through simulation. This analysis may reveal that the system will not perform adequately. The process architecture, the scheduling policy, the execution platform or all of these may then have to be redesigned to improve the performance of the system.

Timing analysis for real-time systems is difficult. Because aperiodic stimuli are unpredictable, designers have to make assumptions about the probability of these stimuli occurring (and therefore requiring service) at any particular time. These assumptions may be incorrect, and system performance after delivery may not be adequate.

Because real-time systems must meet their timing constraints, you may not be able to use object-oriented development for hard real-time systems. Object-oriented development involves hiding data representations and accessing attribute values through operations defined with the object. This means that there is a significant performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations. The consequent loss of performance may make it impossible to meet real-time deadlines.

Timing constraints or other requirements may sometimes mean that it is best to implement some system functions, such as signal processing, in hardware rather than in software. Hardware components deliver much better performance than the equivalent software. System-processing bottlenecks can be identified and replaced by hardware, thus avoiding expensive software optimization.

## 4.2. Real-Time Operating Systems

All but the very simplest embedded systems now work in conjunction with a real-time operating system (RTOS). A real-time operating system manages processes and resource allocation in a real-time system. It starts and stops processes so that stimuli can be handled and allocates memory and processor resources. There are many RTOS products available, from very small, simple systems for consumer devices to complex systems for cell phones and mobile devices and operating systems specially designed for process control and telecommunications.

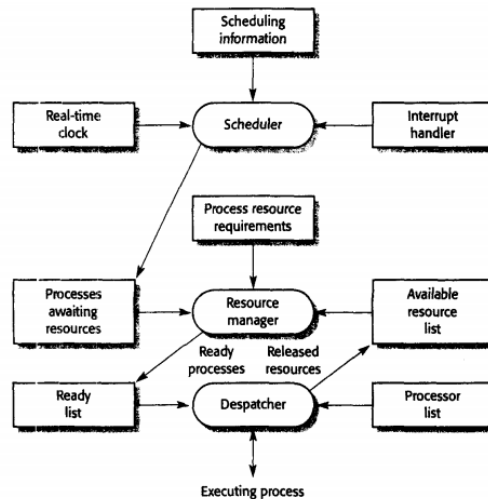


Figure 4.3: Components of a real-time operating system [1]

The components of an RTOS (Figure 4.3) depend on the size and complexity of the real-time system being developed. For all except the simplest systems, they usually include:

1. **A real-time clock.** This provides information to schedule processes periodically.
2. **An interrupt handler.** This manages aperiodic requests for service.
3. **A scheduler.** This component is responsible for examining the processes that can be executed and choosing one of these for execution.
4. **A resource manager.** Given a process that is scheduled for execution, the resource manager allocates appropriate memory and processor resources.
5. **A dispatcher.** This component is responsible for starting the execution of a process.

Real-time operating systems for large systems, such as process control or telecommunication systems may have additional facilities, such as disk storage management and fault management facilities that detect and report system faults and a configuration manager that supports the dynamic reconfiguration of real-time applications.

### 4.3. Monitoring and Control Systems

Monitoring and control systems are an important class of real-time system. They check sensors providing information about the systems environment and take actions depending on the sensor reading. Monitoring systems take action when some exceptional sensor value is detected. Control systems continuously control hardware actuators depending on the value of associated sensors.

The characteristic architecture of monitoring and control systems is shown in Figure 4.4. Each type of sensor being monitored has its own monitoring process, as does each type of actuator that is being controlled. A monitoring process collects and integrate; the data before passing it to a control process, which makes decisions based on this data and sends appropriate control commands to the equipment control processes. In simple systems, the monitoring and control responsibilities may be integrated into a single process. Two other processes that can be included in monitoring and control systems are: **a testing**

*process* that can run hardware test programs and *a control panel process* that manages the system control panels or operator console.

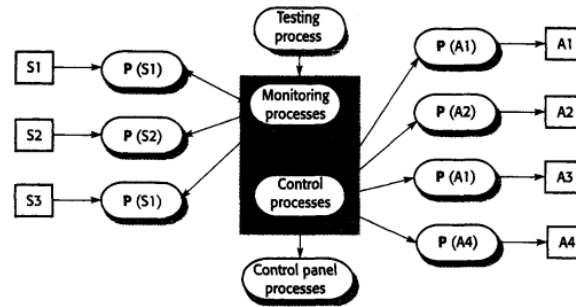


Figure 4.4: Generic architecture for a monitoring and control system [1]

#### 4.4. Data Acquisition Systems

Data acquisition systems collect data from sensors for subsequent processing and analysis. These systems are used in circumstances where the sensors are collecting lots of data from the system's environment and it isn't possible or necessary to process the data collected in real-time. Data acquisition systems are commonly used in scientific experiments and process control systems where physical processes, such as a chemical reaction, happen very quickly.

In data acquisition systems, the sensors may be generating data very quickly, and the key problem is to ensure that a sensor reading is collected before the sensor value changes. This leads to a generic architecture, as shown in Figure 4.5. The essential feature of the architecture of data acquisition systems is that each group of sensors has three processes associated with it, and they are:

- *a sensor process* that interfaces with the sensor and converts analogue data to digital values if necessary,
- *a buffer process*, and
- a process that consumes the data and carries out further processing

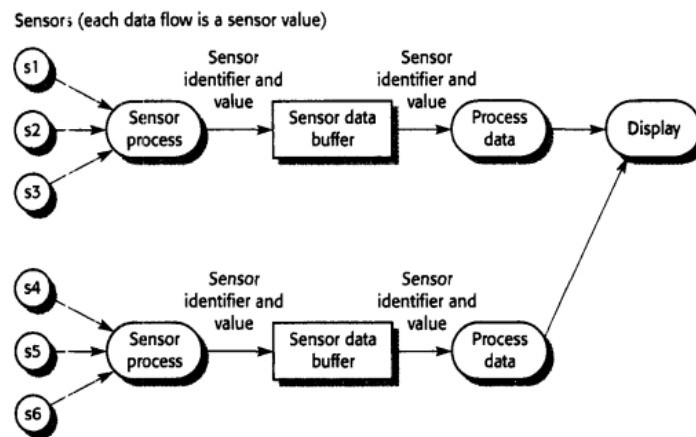


Figure 4.5: The generic architecture of data acquisition systems [1]

Sensors, of course, can be of different types, and the number of sensors in a group depends on the rate at which data arrives from the environment. In Figure 4.5, two groups of sensors, s1-s3 and s4-s6 are shown, and on the right, a further process that displays the sensor data is also shown. Most data acquisition systems include display and reporting processes that aggregate the collected data and carry out further processing.

In real-time systems that involve data acquisition and processing, the execution speeds and periods of the acquisition process (the producer) and the processing process (the consumer) may be out of step. When significant processing is required, the data acquisition may go faster than the data processing. If only simple computations need be carried out, the processing may be faster than the data acquisition.

To smooth out these speed differences, data acquisition systems buffer input data using a circular buffer. The process producing the data (the producer) adds information to this buffer, and the process using the data (the consumer) takes information from the buffer. (Figure 4.6)

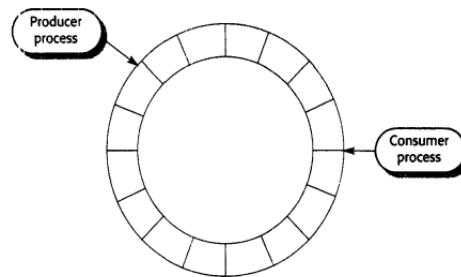


Figure 4.6: A ring buffer for data acquisition [1]

Obviously, mutual exclusion must be implemented to prevent the producer and consumer processes from accessing the same element in the buffer at the same time. The system must also ensure that the producer does not try to add information to a full buffer and the consumer does not take information from an empty buffer

## REFERENCESS

[1] I. Sommerville , 2008. Software Engineering. Eighth Edition, Addison-Wesley

## 5. SOFTWARE REUSE

The design process in most engineering disciplines is based on reuse of existing systems or components. Mechanical or electrical engineers do not normally specify a design where every component has to be manufactured specially. They base their design on components that have been tried and tested in other systems. These are not just small components such as flanges and valves but include major subsystems such as engines, condensers or turbines.

Reuse-based software engineering is a comparable software engineering strategy where the development process is geared to reusing existing software. The move to reuse-based development has been in response to demands for *lower software production and maintenance costs, faster delivery of systems and increased software quality*. More and more companies see their software as a valuable asset and are promoting reuse to increase their return on software investments.

Reuse-based software engineering is an approach to development that tries to maximize the reuse of existing software. The software units that are reused may be of radically different sizes. For example:

1. **Application system reuse.** The whole of an application system may be reused by incorporating it without change into other systems, by configuring the application for different customers or by developing application families that have a common architecture but are tailored for specific customers.
2. **Component reuse.** Components of an application ranging in size from sub-systems to single objects may be reused. For example, a pattern-matching system developed as part of a text-processing system may be reused in a database management system.
3. **Object and function reuse.** Software components that implement a single function, such as a mathematical function or an object class, may be reused. This form of reuse is based around standard libraries. Many libraries of functions and classes for different types of application and development platform are available. These can be easily used by linking them with other application code. In areas such as mathematical algorithms and graphics, where specific expertise is needed to develop objects and functions, this is a particularly effective approach.

Software systems and components are specific reusable entities, but their specific nature sometimes means that it is expensive to modify them for a new situation. A complementary form of reuse is concept reuse where, rather than reuse a component, the reused entity is more abstract and is designed to be configured and adapted for a range of situations. Concept reuse can be embodied in approaches such as design patterns, configurable system products and program generators. There use process, when concepts are reused, includes an instantiation activity where the abstract concepts are configured for a specific situation.

An obvious advantage of software reuse is that overall development costs should be reduced. Fewer software components need be specified, designed, implemented and validated. However, cost reduction is only one advantage of reuse. Other advantages of software reuse are explained in Table 5.1.

| Benefit                      | Explanation  |
|------------------------------|--|
| Increased dependability      | Reused software, which has been tried and tested in working systems, should be more dependable than new software because its design and implementation faults have already been found and fixed.   |
| Reduced process risk         | The cost of existing software is already known, while the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.   |
| Effective use of specialists | Instead doing the same work over and over, these application specialists can develop reusable software that encapsulates their knowledge.  |
| Standards compliance         | Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users are less likely to make mistakes when presented with a familiar interface. |
| Accelerated development      | Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced  |

Table 5.1: Benefits of software reuse [1]

However, there are also costs and problems associated with reuse, mentioned in Table 5.2. In particular, there is a significant cost associated with understanding whether a component is suitable for reuse in a particular situation and in testing that component to ensure its dependability. These additional costs may inhibit the introduction of reuse and may mean that the reductions in overall development cost through reuse may be less than anticipated.

| Problem   | Explanation   |
|---|---|
| Increased Maintenance Costs                             | If the source code of a reused software system or component is not available then maintenance costs may be increased because the reused elements of the system may become increasingly incompatible with system changes.  |
| Lack of Tool Support                                    | CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account   |
| Non-Invented-Here Syndrome                              | Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.                                 |
| Creating and Maintaining a Component Library            | Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.  |
| Finding, Understanding and Adapting Reusable Components | Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they will make include a component search as part of their normal development process. |

Table 5.2: Problems with reuse [1]

Systematic reuse does not just happen-it must be planned and introduced through an organization-wide reuse program. This has been recognized for many years in Japan, where reuse is an integral part of the Japanese 'factory' approach to software development. Companies, such as, Hewlett-Packard have also been very successful in their reuse programs.

### 5.1. The Reuse Landscape

Over the past 20 years, many techniques have been developed to support software reuse. These exploit the facts that systems in the same application domain are similar and have potential for reuse, that reuse is

possible at different levels (from simple functions to complete applications), and that standards for reusable components facilitate reuse. (Shown in Figure 5.1)

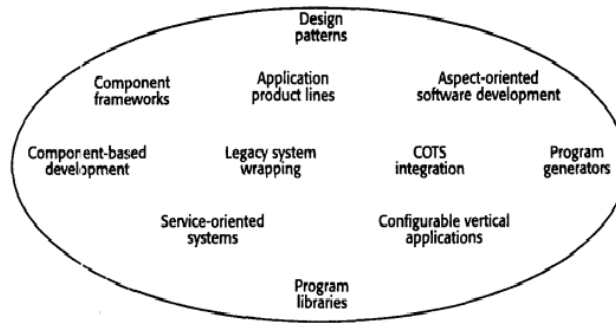


Figure 5.1: The reuse landscape [1]

| Approach                             | Description  |
|--------------------------------------|--|
| Design patterns                      | Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions. |
| Component-based development          | Systems are developed by integrating components (collections of objects) that conform to component model standards.                            |
| Application frameworks               | Collections of abstract and concrete classes can be adapted and extended to create application systems.  |
| Legacy system wrapping               | Legacy systems that can be wrapped by defining a set of interfaces and providing access to these legacy systems through these interfaces.      |
| Service-oriented systems             | Systems are developed by linking shared services, which may be externally provided.  |
| Application product lines            | An application type is generalized around a common architecture so that it can be adapted for different customers.                             |
| COTS integration                     | Systems are developed by integrating existing application systems.   |
| Configurable vertical applications   | A generic system is designed so that it can be configured to the needs of specific system customers.   |
| Program libraries                    | Class and function libraries implementing commonly used abstractions are available for reuse.  |
| Program generators                   | A generator system embeds knowledge of a particular type of application and can generate systems or system fragments in that domain.           |
| Aspect-oriented software development | Shared components are woven into an application at different places when the program is compiled.  |

Table 5.3: Approaches that support software reuse [1]

Given this array of techniques for reuse, the key question is which is the most appropriate technique to use? Obviously, this depends on the requirements for the system being developed, the technology and reusable assets available, and the expertise of the development team. Key factors that you should consider when planning reuse are:

- **The development schedule for the software.** If the software has to be developed quickly, you should try to reuse off-the-shelf systems rather than individual components. These are large-grain reusable assets. Although the fit to requirements may be imperfect, this approach minimizes the amount of development required.
- **The expected software lifetime.** If you are developing a long-lifetime system, you should focus on the maintainability of the system. In those circumstances, you should not just think about the immediate possibilities of reuse but also the long term implications. You will have to adapt the system to new requirements, which will probably mean making changes to components and how

they are used. If you do not have access to the source code, you should probably avoid using components and systems from external suppliers; you cannot be sure that these suppliers will be able to continue supporting the reused software.

- ***The background, skills and experience of the development team.*** All reuse technologies are fairly complex and you need quite a lot of time to understand and use them effectively. Therefore, if the development team has skills in a particular area, this is probably where you should focus.
- ***The criticality of the software and its non-functional requirements.*** For a critical system that has to be certified by an external regulator, you may have to create a dependability case for the system. This is difficult if you don't have access to the source code of the software. If your software has stringent performance requirements, it may be impossible to use strategies such as reuse through program generators. These systems tend to generate relatively inefficient code.
- ***The application domain.*** In some application domains, such as manufacturing and medical information systems, there are several generic products that may be reused by configuring them to a local situation. If you are working in such a domain, you should always consider these an option.
- ***The platform on which the system will run.*** Some components models, such as COM/ActiveX, are specific to Microsoft platforms. If you are developing on such a platform, this may be the most appropriate approach. Similarly, generic application systems may be platform-specific and you may only be able to reuse these if your system is designed for the same platform.

The range of available reuse techniques is such that, in most situations, there is the possibility of some software reuse. ***Whether or not reuse is achieved is often a managerial rather than a technical issue.*** Managers may be unwilling to compromise their requirements to allow reusable components to be used, or they may decide that original component development would help create software asset base. They may not understand the risks associated with reuse as well as they understand the risks of original development. Therefore, although the risks of new software development may be higher, some managers may prefer known to unknown risks.

## 5.2. Design Patterns

When you try to reuse executable components, you are inevitably constrained by detailed design decisions that have been made by the implementers of these components. These range from the particular algorithms that have been used to implement the components to the objects and types in the component interfaces. When these design decisions conflict with your particular requirements, reusing the component is either impossible or introduces inefficiencies into your system.

One way around this is to reuse abstract designs that do not include implementation detail. You can implement these to fit your specific application requirements. The first instances of this approach to reuse came in the documentation and publication of fundamental algorithms) and, later, in the documentation of abstract data types such as stacks, trees and lists. More recently, this approach to reuse has been embodied in design patterns.

Design patterns were derived from ideas put forward by Christopher Alexander, who suggested that there were certain patterns of building design that were common and that were inherently pleasing and effective. The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification. Rather, you can think of it



as a description of accumulated wisdom and experience, a well-tried solution to a common problem. A quote from the hillside.net website, which is dedicated to maintaining information about patterns, encapsulates their role in reuse:

*Patterns and Pattern Languages are ways to describe best practices, good designs. And capture experience in a way that it is possible for others to reuse this experience.*

Most designers think of design patterns as a way of supporting object-oriented design. Patterns often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle of encapsulating experience in a pattern is one that is equally applicable to all software design approaches.

The four essential elements of design patterns are:

1. A name that is a meaningful reference to the pattern
2. A description of the problem area that explains when the pattern may be applied
3. A solution description of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A statement of the consequences-the results and trade-offs-of applying the pattern. This can help designers understand whether a pattern can be effectively applied in a particular situation.

These essential elements of a pattern description may be decomposed, shown in the example in Table 5.4.

| Pattern name                | Observer   |
|-----------------------------|--|
| <b>Description</b>          | Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.   |
| <b>Problem description</b>  | In many situations, it is necessary to provide multiple displays of some state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations may support interaction and, when the state is changed, all displays must be updated.<br>This pattern may be used in all situations where more than one display format for state information may be required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.  |
| <b>Solution description</b> | This defines two abstract objects, Subject and Observer, and two concrete objects, Concrete Subject and Concrete object, which inherit the attributes of the related abstract objects. The state to be displayed is maintained in Concrete Subject, which also inherits operations from Subject allowing it to add and remove Observers and to issue a notification when the state has changed.<br>The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update () interface of Observer hat allows these copies to be kept in step. The ConcreteObserver automatically displays its state-this is not normally an interface operation. |
| <b>Consequences</b>         | The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects; Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated some of which may not be necessary.  |

Table 5.4: A description of the observer pattern [1]

A huge number of published patterns are now available covering a range of application domains and languages. The notion of a pattern as a reusable concept has been developed in a number of areas apart

from software design, including configuration management, user interface design and interaction scenarios.

The use of patterns is an effective form of reuse. However, only experienced software engineers who have a deep knowledge of patterns can use them effectively. These developers can recognize generic situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution.

### 5.3. Generator-Based Reuse

Concept reuse through patterns relies on describing the concept in an abstract way and leaving it up to the software developer to create an implementation. An alternative approach to this is generator-based reuse. In this approach reusable knowledge is captured in a program generator system that can be programmed by domain experts using either a domain-oriented language or an interactive CASE tool that supports system generation. The application description specifies, in an abstract way, which reusable components are to be used, how they are to be combined and their parameterization. Using this information, an operational software system can be generated.

Generator-based reuse takes advantage of the fact that applications in the same domain, such as business systems, have common architectures and carry out comparable functions. For example, data-processing systems normally follow an input-process-output model and usually include operations such as data verification and report generation. Therefore, generic components for selecting items from a database, checking that these are within range and creating reports can be created and incorporated in an application generator. To reuse these components, the programmer simply has to select the data items to be used, the checks to be applied and the format of reports.

Generator-based reuse has been particularly successful for business application systems, and there are many different business application generator products available. These may generate complete applications or may partially automate application creation and leave the programmer to fill in specific details. The generator-based approach to reuse is also used in other areas, including:

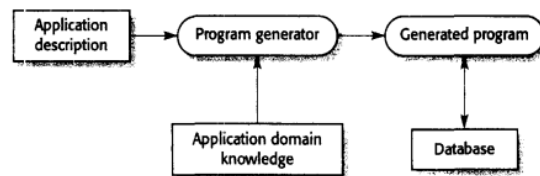


Figure 5.2: Generator-based reuse [1]

- **Parser generators for language processing.** The generator input is a grammar describing the language to be parsed, and the output is a language parser. This approach is embodied in systems such as lex and yacc for C and JavaCC, a compiler for Java.
- **Code generators in CASE tools.** The input to these generators is a software design and the output is a program implementing the designed system. These may be based on UML models and,

depending on the information in the UML models, generate either a complete program or component, or a code skeleton. The software developer then adds detail to complete the code.

These approaches to generator-based reuse take advantage of the common structure of applications in these areas. The technique has also been used in more specific application domains such as command and control systems and scientific instrumentation where libraries of components have been developed. Domain experts then use a domain-specific language to compose these components and create applications. However, there is a high initial cost in defining and implementing the domain concepts and composition language. This has meant that many companies are reluctant to take the risks of adopting this approach.

Generator-based reuse is cost-effective for applications such as business data processing. It is much easier for end-users to develop programs using generators compared to other component-based approaches to reuse. Inevitably, however, there are inefficiencies in generated programs. This means that it may not be possible to use this approach in systems with high-performance or throughput requirements.

Generative programming is a key component of emerging techniques of software development that combine program generation with component-based development.

The most developed of these approaches is *aspect-oriented software development (AOSD)*. Aspect-oriented software development addresses one of the major problems in software design—the problem of separation of concerns. Separation of concerns is a basic design principle; you should design your software so that each unit or component does one thing and one thing only. For example, in the LIBSYS system, there should be a component concerned with searching for documents, a component concerned with printing documents, a component concerned with managing downloads, and so on.

However, in many situations, concerns are not associated with clearly defined application functions but are cross-cutting—that is, they affect all of the components in the system. For example, say you want to keep track of the usage of each of the system modules by each system user. You therefore have a monitoring concern that has to be associated with all components. This can't be simply implemented as an object that is referenced by these components. The specific monitoring that is carried out needs context information from the system function that is being monitored.

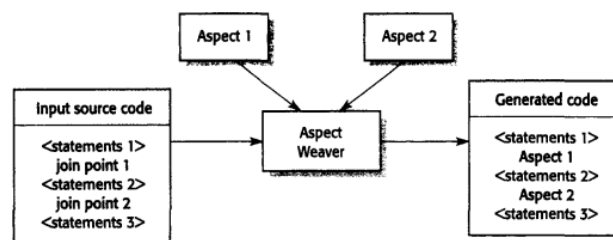


Figure 5.3: Aspects oriented software development [1]

In aspect-oriented programming, these cross-cutting concerns are implemented as aspects and, within the program; you define where an aspect should be associated. These are called the *join points*. Aspects are developed separately; then, in a pre-compilation step called *aspect weaving*, they are linked to the join

points. Aspect weaving is a form of program generation-the output from the weaver is a program where the aspect code has been integrated. A development of Java called AspectJ is the best-known language for aspect oriented development.

#### 5.4. Application Frameworks

The early proponents of object-oriented development suggested that objects were the most appropriate abstraction for reuse. However, experience has shown that objects are often too fine-grain and too specialized to a particular application. Instead, it has become clear that object-oriented reuse is best supported in an object-oriented development process through larger-grain abstractions called frameworks.

A framework (or application framework) is a sub-system design made up of a collection of abstract and concrete classes and the interface between them. Particular details of the application sub-system are implemented by adding components and by providing concrete implementations of abstract classes in the framework. Frameworks are rarely applications in their own right. Applications are normally constructed by integrating a number of frameworks.

The three classes of framework are:

1. ***System infrastructure frameworks***. These frameworks support the development of system infrastructures such as communications, user interfaces and compilers.
2. ***Middleware integration frameworks***. These consist of a set of standards and associated object classes that support component communication and information exchange. Examples of this type of framework include CORBA, Microsoft's COM+, and Enterprise JavaBeans. These frameworks provide support for standardized component models.
3. ***Enterprise application frameworks***. These are concerned with specific application domains such as telecommunications or financial systems. These embed application domain knowledge and support the development of end-user applications.

As the name suggests, a framework is a generic structure that can be extended to create a more specific sub-system or application. It is implemented as a collection of concrete and abstract object classes. To extend the framework, you may have to add concrete classes that inherit operations from abstract classes in the framework. In addition, you may have to define callbacks. Callbacks are methods that are called in response to events recognized by the framework.

One of the best-known and most widely used frameworks for GUI design is the Model-View-Controller (MVC) framework. The MVC framework was originally proposed as an approach to GUI design that allowed for multiple presentations of an object and separate styles of interaction with each of these presentations. The MVC framework supports the presentation of data in different ways and separate interaction with each of these presentations. When the data is modified through one of the presentations, all of the other presentations are updated.

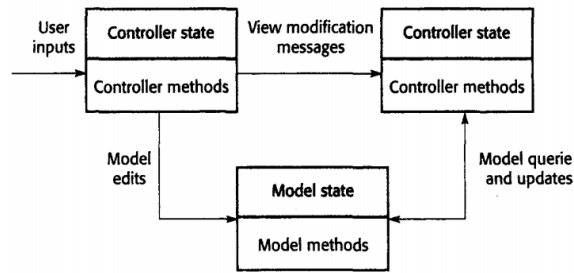


Figure 5.4: Mode-View-Controller (MVC) framework [2]

- A. **Model:** The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.
- B. **View:** The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.
- C. **Controller:** Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

Frameworks are often instantiations of a number of patterns. For example, the MVC framework includes the Observer pattern, the Strategy pattern that is concerned with updating the model, the Composite pattern and a number of others.

Applications that are constructed using frameworks can be the basis for further reuse through the concept of software product lines or application families. Because these applications are constructed using a framework, modifying family members to create new family members is simplified. However, frameworks are usually more abstract than generic products and thus allow a wider range of applications to be created.

The fundamental problem with frameworks is their inherent complexity and the time it takes to learn to use them. Several months may be required to completely understand a framework, so it is likely that, in large organizations, some software engineers will become framework specialists. There is no doubt that this is an effective approach to reuse, but it is very expensive to introduce into software development processes.

## 5.5. Application System Reuse

Application system reuse involves reusing entire application systems either by configuring a system for a specific environment or by integrating two or more systems to create a new application. Application system reuse is often the most effective reuse technique. It involves the reuse of large-grain assets that can be quickly configured to create a new system.

The two types of application reuse are: the creation of new systems by integrating two or more off-the-shelf applications and the development of product lines. A product line is a set of systems based around a common core architecture and shared components. The core system is specifically designed to be configured and adapted to suit the specific needs of different system customers.

## COMMERCIAL-OFF-THE-SHELF (COTS) PRODUCT REUSE

A commercial-off-the-shelf (COTS) product is a software system that can be used without change by its buyer. Virtually all desktop software and a wide variety of server products are COTS software. Because this software is designed for general use, it usually includes many features and functions so has the potential to be reused in different applications and environments. Although there can be problems with this approach to system construction, there is an increasing number of success stories that demonstrate its viability.

Some types of COTS product have been reused for many years. Database systems are perhaps the best example of this. Very few developers would consider implementing their own database management system. However, until the mid-1990s, there were only a few large systems such as database management systems and teleprocessing monitors that were routinely reused. Most large systems were designed as standalone systems, and there were often many problems in making these systems work together.

It is now common for large systems to have defined Application Programming Interfaces (APIs) that allow program access to system functions. This means that creating large systems such as e-commerce systems by integrating arrange of COTS systems should always be considered as a serious design option. Because of the functionality that these COTS products offer, it is possible to reduce costs and delivery times by orders of magnitude compared to the development of new software.

Furthermore, risks may be reduced as the product is already available and managers can see whether it meets their requirements. To develop systems using COTS products, you have to make a number of design choices:

1. ***Which COTS products offer the most appropriate functionality?*** If you don't already have experience with a COTS product, it can be difficult to decide which product is the most suitable.
2. ***How will data be exchanged?*** In general, individual products use unique data structures and formats, and you have to write adaptors that convert from one representation to another.
3. ***What features of a product will actually be used?*** Most COTS products have more functionality than you need, and functionality is often duplicated across different products. You have to decide which features in what product are most appropriate for your requirements. If possible, you should also deny access to unused functionality because this can interfere with normal system operation. The failure of the first flight of the Ariane5 rocket was a consequence of failure in unused functionality in a reused sub-system.

The four problems with COTS system integration are:

- ***Lack of control over functionality and performance.*** Although the published interface of a product may appear to offer the required facilities, these may not be properly implemented or may perform poorly. The product may have hidden operations that interfere with its use in a specific situation. Fixing these problems may be a priority for the COTS product integrator but

may not be of real concern to the product vendor. Users may simply have to find workarounds to problems if they wish to reuse the COTS product.

- **Problems with COTS system interoperability.** It is sometimes difficult to get COTS products to work together because each product embeds its own assumptions about how it will be used.
- **No control over system evolution.** Vendors of COTS products make their own decisions on system changes in response to market pressures. For PC products in particular, new versions are often produced frequently and may not be compatible with all previous versions. New versions may have additional unwanted functionality, and previous versions may become unavailable and unsupported.
- **Support from COTS vendors.** The level of support available from COTS vendors varies widely. Because these are off-the-shelf systems, vendor support is particularly important when problems arise because developers do not have access to the source code and detailed documentation of the system. While vendors may commit to providing support, changing market and economic circumstances may make it difficult for them to deliver this commitment. For example, a COTS system vendor may decide to discontinue a product because of limited demand or may be taken over by another company that does not wish to support all of its current products.

Of course, it is unlikely that all of these problems will arise in every case, but my guess is that at least one of them should be expected in most COTS integration projects. Consequently, the cost and schedule benefits from COTS reuse are likely to be less than they might first appear.

Furthermore, in many cases, the cost of system maintenance and evolution may be greater when COTS products are used. All of the above difficulties are lifecycle problems; they don't just affect the initial development of the system. The further removed from the original system developers the people involved in the system maintenance become, the more likely it is that real difficulties, will arise with the integrated COTS products.

In spite of these problems, the benefits of COTS product reuse are potentially large because these systems offer so much functionality to the reuser. Months and sometimes, years of implementation effort can be saved if an existing system is reused and system development times drastically reduced. If rapid system delivery is essential and you have some requirements flexibility, then COTS product integration is often the most effective reuse strategy to adopt.

## SOFTWARE PRODUCT LINES

One of the most effective approaches to reuse is creating software product lines or application families. A product line is a set of applications with a common application-specific architecture. Each specific application is specialized in some way. The common core of the application family is reused each time a new application is required. The new development may involve specific component configuration, implementing additional components and adapting some of the components to meet new demands. Various types of specialization of a software product line may be developed:

1. **Platform specialization.** Versions of the application are developed for different platforms. For example, versions of the application may exist for Windows, Solaris and Linux platforms. In this case, the functionality of the application is normally unchanged; only those components that interface with the hardware and operating system are modified.

2. **Environment specialization.** Versions of the application are created to handle particular operating environments and peripheral devices. For example, a system for the emergency services may exist in different versions depending on the type of radio system used. In this case, the system components are changed to reflect the functionality of the communications equipment used.
3. **Functional specialization.** Versions of the application are created for specific customers who have different requirements. For example, a library automation system may be modified depending on whether it is used in a public library, a reference library or a university library. In this case, components that implement functionality may be modified and new components added to the system.
4. **Process specialization.** The system is adapted to cope with specific business processes. For example, an ordering system may be adapted to cope with a centralized ordering process in one company and a distributed process in another.

Software product lines are designed to be reconfigured. This reconfiguration may involve adding or removing components from the system, defining parameters and constraints for system components, and including knowledge of business processes. Software product lines can be configured at two points in the development process:

1. **Deployment-time configuration** where a generic system is designed for configuration by a customer or consultants working with the customer. Knowledge of the customer specific requirements and the system's operating environment is embedded in a set of configuration files that are used by the generic system.
2. **Design-time configuration** where the organization that is developing the software modifies a common product line core by developing, selecting or adapting components to create a new system for a customer.

Deployment-time configuration is the approach used in vertical software packages that are designed for a specific application such as a hospital information management system. It is also used in Enterprise Resource Planning (ERP) systems, such as those produced by SAP and BEA. These are large-scale, integrated system designed to support business processes such as ordering and invoicing, inventory management and manufacturing scheduling. The configuration process for these systems involves gathering detailed information about the customers business and business processes and then embedding this information in a configuration database. This often requires detailed knowledge of configuration notations and tools and is usually carried out by consultants working alongside system customers.

The generic ERP system includes a large number of modules that may be composed in different ways to create a specific system. The configuration process involves choosing which modules are to be included, configuring these individual modules, defining business processes and business rules, and defining the structure and organization of the system database.

ERP systems are perhaps the most wide spread example of software reuse. The majority of large companies use these systems to support some or all of their functions. However, there is the obvious limitation that the functionality of the system is restricted to the functionality of the generic core. Furthermore, a company's processes and operations have to be expressed in the system configuration language, and there may be a mismatch between the concepts in the business and the concepts supported in the configuration language. For example, in an ERP system that was sold to a university, the concept of



a customer had to be defined. This caused real problems because universities have multiple types of customer (students, research-funding agencies, educational charities, etc.) and none of these are comparable to a commercial customer. A serious mismatch between the business model used by the system and that of the customer makes it highly probable that the ERP system will not meet the customers real needs.

The alternative approach to application family reuse is configuration by the system supplier before delivery to the customer. The supplier starts with a generic system and then, by modifying and extending modules in this system, creates a specific system that delivers the required customer functionality. This approach usually involves changing and extending the source code of the core system so greater flexibility is possible than with deployment-time configuration.

Software product lines usually emerge from existing applications. That is, an organization develops an application and, when a new application is required, uses this as a basis for the new application. Further demands for new applications cause the process to continue. However, because change tends to corrupt application structure, at some stage a specific decision to design a generic product line is made. This design is based on reusing the knowledge gained from developing the initial set of applications.

You can think of software product lines as instantiations and specializations of more general application architectures. Application architecture is very general; software product lines specialize the architecture for a specific type of application.

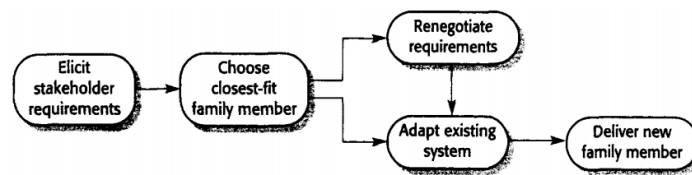


Figure 5.5: Product instance development [2]

The steps involved in adapting an application family to create a new application are shown in Figure 5.5.

1. **Elicit stakeholder requirements.** You may start with a normal requirements engineering process. However, because a system already exists, you will need to demonstrate and have stakeholders experiment with that system, expressing their requirements as modifications to the functions provided.
2. **Choose closest-fit family member.** The requirements are analyzed and the family member that is the closest fit is chosen for modification. This need not be the system that was demonstrated.
3. **Renegotiate requirements.** As more details of required changes emerge and the project is planned, there may be some requirements renegotiation to minimize the changes that are needed.
4. **Adapt existing system.** New modules are developed for the existing system, and existing system modules are adapted to meet the new requirements.
5. **Deliver new family member.** The new instance of the product line is delivered to the customer. At this stage, you should document its key features so that it may be used as a basis for other system developments in the future.

When you create a new member of an application family, you may have to find a compromise between reusing as much of the generic application as possible and satisfying detailed stakeholder requirements. The more detailed the system requirements, the less likely it is that the existing components will meet these requirements. However, if stakeholders are willing to be flexible and to limit the system modifications that are required, you can usually deliver the system more quickly and data lower cost.

In general, developing applications by adapting a generic version of the application means that a very high proportion of the application code is reused. Furthermore, application experience is often transferable from one system to another, so that when software engineers join a development team, their learning process is shortened. Testing is simplified because tests for large parts of the application may also be reused, reducing the overall application development time.

## REFERENCES

[1] I. Sommerville , 2008. Software Engineering. Eighth Edition, Addison-Wesley

## 6. COMPONENT-BASED SOFTWARE ENGINEERING

Component-based software engineering (CBSE) emerged in the late 1990s as a reuse-based approach to software systems development. Its creation was motivated by designers' frustration that object-oriented development had not led to extensive reuse, as originally suggested. Single object classes were too detailed and specific, and often had to be bound with an application at compile-time. You had to have detailed knowledge of the classes to use them, which usually meant that you had to have the component source code. This made marketing objects as reusable components difficult. In spite of early optimistic predictions, no significant market for individual objects has ever developed.

CBSE is the process of defining, implementing and integrating or composing loosely coupled independent components into systems. It has become as an important software development approach because software systems are becoming larger and more complex and customers are demanding more dependable software that is developed more quickly. The only way that we can cope with complexity and deliver better software more quickly is to reuse rather than re-implement software components.

The essentials of component-based software engineering are:

1. ***Independent components that are completely specified by their interfaces.*** There should be a clear separation between the component interface and its implementation so that one implementation of a component can be replaced by another without changing the system.
2. ***Component standards that facilitate the integration of components.*** These standards are embodied in a component model and define, at the very minimum, how component interfaces should be specified and how components communicate. Some models define interfaces that should be implemented by all conformant components. If components conform to standards, then their operation is independent of their programming language. Components written in different languages can be integrated into the same system.
3. ***Middleware that provides software support for component integration.*** To make independent, distributed components work together, you need middleware support that handles component communications. Middleware such as CORBA handles low-level level issues efficiently and allows you to focus on application-related problems. In addition, middleware to implement a component model may provide support for resource allocation, transaction management, security and concurrency.
4. ***A development process that is geared to component-based software engineering.*** If you try to add a component-based approach to a development process that is geared to original software production, you will find that the assumptions inherent in the process limit the potential of CBSE.

Component-based development is being increasingly adopted as a mainstream approach to software engineering even if reusable components are not available.

Underlying CBSE are sound design principles that support the construction of understandable and maintainable software. Components are independent so they do not interfere with each other's operation. Implementation details are hidden, so the component's implementation can be changed without affecting the rest of the system. The components communicate through well-defined interfaces, so if these interfaces are maintained, one component can be replaced by another that provides additional or enhanced

functionality. In addition, component infrastructures provide high-level platform that reduce the costs of application development.

Although CBSE is developing rapidly into a mainstream approach to software development, a number of problems remain:

- **Component trustworthiness.** Components are black-box program units and the source code of the component may not be available to component users. In such case, how does a user know that a component is to be trusted? The component may have undocumented failure modes that compromise the system where the component is used. Its non-functional behavior may not be as expected and, most seriously, the black-box component could be a Trojan horse that conceals malicious code that breaches system security.
- **Component certification.** Closely related to trustworthiness is the issue of certification. It has been proposed that independent assessors should certify components to assure users that the components could be trusted. However, it is not clear how this can be made to work. Who would pay for certification, who would be responsible if the component did not operate as certified, and how could the certifiers limit their liability? In my view, the only viable solution is to certify that components conform to a formal specification. However, the industry does not appear to be willing to pay for this.
- **Emergent property prediction.** All systems have emergent properties, and trying to predict and control these emergent properties is important in the system development process. Because components are opaque, predicting their emergent properties is particularly difficult. Consequently, you may find that when components are integrated, the resulting system has undesirable properties that limit its use.
- **Requirements trade-offs.** You usually have to make trade-offs between ideal requirements and available components in the system specification and design process. At the moment, making these trade-offs is an intuitive process. We need a more structured, systematic trade-off analysis method to help designers select and configure components.

The main use of CBSE so far has been to build enterprise information systems, such as e-commerce systems. The components that are reused are internally developed or are procured from known, trusted suppliers. Although some vendors sell components online, most companies are still reluctant to trust externally procured, binary components. It is unlikely that the complete vision of CBSE with specialized component suppliers will be realized until these major problems have been solved.

## 6.1. Components and Component Model

There is general agreement in the community that a component is an independent software unit that can be composed with other components to create software system. Beyond that, however, different people have proposed definitions of a software component. A component can be defined as:

*A software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*

Another definition can be:

*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

| Component characteristics | Description   |
|---------------------------|---|
| Standardized              | Component standardization means that a component used in a CBSE process has to conform to some standardized component model. This model may define component interfaces, component metadata, documentation, composition and deployment.   |
| Independent               | A component should be independent-it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification. |
| Composable                | For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.  |
| Deployable                | To be deployable, a component has to be self-contained and must be able to operate as a standalone entity on a component platform that implements the component model. This usually means that the component is binary and does not have to be compiled before it is deployed.      |
| Documented                | Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified.  |

Table 6.1: Component characteristics [1]

What these definitions have in common is that they agree that components are independent and that they are the fundamental unit of composition in a system. These formal component definitions are rather abstract and do not really give you a clear picture of what a component does. One of the most useful ways to consider *a component is as a standalone service provider. When a system needs some service, it calls on a component to provide that service without caring about where that component is executing or the programming language used to develop the component.* For example, a component in a library system might provide a search service that allows users to search different library catalogues; a component that converts from one graphical format to another (e.g., Tiff to JPEG) provides a data-conversion service.

Viewing a component as a service provider emphasizes two critical characteristics of a reusable component:

1. The component is an independent executable entity. Source code is not available, so the component does not have to be compiled before it is used with other system components.
2. The services offered by a component are made available through an interface, and all interactions are through that interface. The component interface is expressed in terms of parameterized operations and its internal state is never exposed.

Components are defined by their interfaces and, in the most general cases, can be thought of as having two related interfaces:

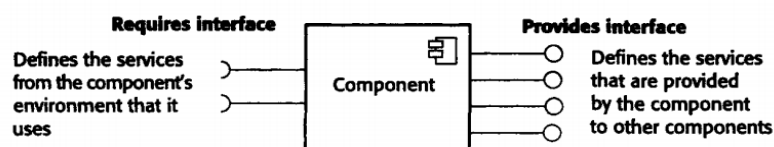


Figure 6.1: Component interfaces [1]

1. ***A provides interface*** defines the services provided by the component. Provides interface, essentially, is the component API. It defines the methods that can be called by a user of the component. Provides interfaces are indicated by a circle at the end of a line from the component icon.
2. ***A requires interface*** specifies what services must be provided by other components in the system. If these are not available, then the component will not work. This does not compromise the independence or deployability of the component because it is not required that a specific component should be used to provide the services. Requires interfaces are indicated by a semi-circle at the end of a line from the component icon. Notice that provides and required interface icons can fit together like a ball and socket.

Object classes have associated methods that are clearly similar to the methods defined in component interfaces. What, then, is the distinction between components and objects? Components are usually developed using an object-oriented approach, but they differ from objects in a number of important ways:

1. ***Components are deployable entities.*** That is, they are not compiled into an application program but are installed directly on an execution platform. The methods and attributes defined in their interfaces can then be accessed by other components.
2. ***Components do not define types.*** A class definition defines an abstract data type and objects are instances of that type. A component is an instance, not a template that is used to define an instance.
3. ***Component implementations are opaque.*** Components are, in principle at least; completely defined by their interface specification. The implementation is hidden from component users. Components are often delivered as binary units so the buyer of the component does not have access to the implementation.
4. ***Components are language-independent.*** Object classes have to follow the rules of a particular object-oriented programming language and, generally, can only interoperate with other classes in that language. Although components are usually implemented using object-oriented languages such as Java, you can implement them in non-object-oriented programming languages.
5. ***Components are standardized.*** Unlike object classes that you can implement in anyway, components must conform to some component model that constrains their implementation.

## COMPONENT MODELS

A component model is a definition of standards for component implementation, documentation and deployment. These standards are for component developers to ensure that components can interoperate. They are also for providers of component execution infrastructures who provide middleware to support component operation. Many component models have been proposed, but the most important models are the CORBA component model.

The specific infrastructure technologies such as COM+ and EJB that are used in CBSE are very complex. Consequently, it is difficult to describe these technologies without going into a lot of implementation detail about the assumptions that underlie each approach and the interfaces that are used.

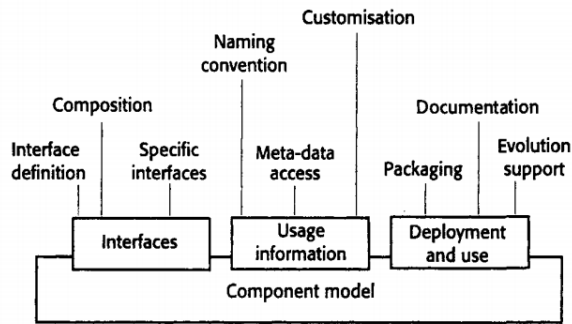


Figure 6.2: Basic elements of a component model [1]

The Figure 6.2 shows that the elements in a component model can be classified as elements relating to the component interfaces, elements relating to information that you need to use the component in a program and elements concerned with component deployment.

The defining elements of a component are its interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions that should be included in an interface definition. The model should also specify the language used to define the interfaces (the IDL). In CORBA and COM+, this is a specific interface definition language; EJB is Java-specific so Java is used as the IDL. Some component models require specific interfaces that must be defined by a component. These are used to compose the component with the component model infrastructure that provides standardized services such as security and transaction management.

In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. In COM+, this is a unique 128-bit identifier. In the CORBA component model and in EJB, it is a hierarchical name with the root based on an Internet domain name. Component metadata is data about the component itself, such as information about its interfaces and attributes. The metadata is important so that users of the component can find out what services are provided and required. Component model implementations normally include specific ways (such as the use of a reflection interface in Java) to access this component metadata.

Components are generic entities and, when deployed, they have to be customized to their particular application environment. The component model should therefore specify how the binary components can be configured for a particular deployment environment.

An important part of a component model is a definition of how components should be packaged for deployment as independent, executable entities. Because components are independent entities, they have to be packaged with everything that is not provided by the component infrastructure or not defined in a *requires-interface*. Deployment information includes information about the contents of a package and its binary organization.

Inevitably, as new requirements emerge, components will have to be changed or replaced. The component model should therefore include rules governing when and how component replacement is allowed. Finally, the component model should define the component documentation that should be produced. This is used to find the component and to decide whether it is appropriate.

Component models are not just standards; they are also the basis for system middleware that provides support for executing components. The services provided by a component model implementation fall into two categories (see Figure 6.3):

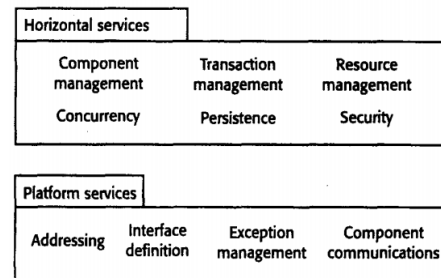


Figure 6.3: Services provided by a component model [1]

1. **Platform services.** These fundamental services enable components to communicate with each other. CORBA is an example of a component model platform.
2. **Horizontal services.** These application-independent services are likely to be used by many different components. The availability of these services reduces the costs of component development and means that potential component incompatibilities can be avoided.

To make use of the services provided by a component model infrastructure, components are deployed in a predefined, standardized container. A container is a set of interfaces used to access the implementations of the support services. Including the component in the container automatically provides service access. The component interfaces themselves are not accessed directly by other components; they are accessed through the container.

## COMPONENT DEVELOPMENT FOR REUSE

The long-term vision of CBSE is that there will be component suppliers whose business is based on the development and sale of reusable components. The problems of trust mean that an open market for components has not yet developed, and most components that are reused are developed within a company. The reusable components are not developed specially but are based on existing components that have already been implemented and used in application systems.

Generally, internally developed components are not immediately reusable. They include application-specific features and interfaces that are unlikely to be required in other applications. Therefore, you have to adapt and extend these components to create a more generic and hence more reusable version. Obviously, this has an associated cost. You have to decide, first, whether a component is likely to be reused and second, whether the cost savings of reuse justify the costs of making the component reusable.

To answer the first of these questions, you have to decide whether the component implements one or more stable domain abstractions. Stable domain abstractions are fundamental concepts in the application domain that change slowly. For example, in a banking system, domain abstractions might include accounts, account holders and statements. In a hospital management system, domain abstractions might include patients, treatments and nurses. These domain abstractions are sometimes called business objects.



If the component is an implementation of a commonly used business object or group of related objects, it can probably be reused.

To answer the question about the cost-effectiveness, you have to assess the costs of changes that are required to make the component reusable. These costs are the costs of component documentation, of component validation and of making the component more generic. Changes that you may make to a component to make it more reusable include:

- Removing application-specific methods
- Changing names to make them more general
- Adding methods to provide more complete functional coverage
- Making exception handling consistent for all methods
- Adding a configuration interface to allow the component to be adapted to different situations of use
- Integrating required components to increase independence

The problem of exception handling is a particularly difficult one. In principle, all exceptions should be part of the component interface. Components should not handle exceptions themselves, because each application will have its own requirements for exception handling. Rather, the component should define what exceptions can arise and should publish these as part of the interface. For example, a simple component implementing a stack data structure should detect and publish stack overflow and stack underflow exceptions. In practice, however, a component may provide some local exception handling, and changing this may have serious implications for the functionality of the component.

It is necessary to find ways of estimating the costs of making a component reusable and estimating the returns from that investment. The benefits of reusing rather than redeveloping a component are not simply productivity gains. They also include quality gains, because a reused component should be more dependable and time-to-market gains. These are the increased returns that accrue from deploying the software more quickly. However, it is difficult to estimate accurately the costs of making a component reusable and the returns from that investment.

Obviously, whether a component is reusable depends on its application domain and functionality. As you add generality to a component, you increase its reusability. However, this normally means that the component has more operations and is more complex, which makes the component harder to understand and use.

There is an inevitable trade-off between the reusability and the usability of a component. Making the component reusable involves providing a set of generic interfaces with operations that cater to all ways in which the component could be used. Making the component usable means providing a simple and minimal interface, that is easy to understand. Reusability adds complexity and hence reduces component understandability. It is therefore more difficult to decide when and how to reuse that component. When designing a reusable component, you must find a compromise between generality and understandability.

Another important source of components is existing legacy systems. These are systems that fulfill an important business function but are written using obsolete software technologies. Because of this, it may

be difficult to use them with new systems. However, if you convert these old systems to components, their functionality can be reused in new applications.

Of course, these legacy systems do not normally have clearly defined requires and provides interfaces. To make these components reusable, you have to conduct a wrapper that defines the component interfaces. The wrapper hides the complexity of the underlying code and provides an interface for external components to access services that are provided. Naturally, this wrapper is a fairly complex piece of software as it has to access the legacy system functionality. However, the cost of wrapper development is often much less than the cost of re-implementing the legacy system

## 6.2. The CBSE Process

The successful reuse of components requires a development process tailored to CBSE. Figure 6.4 shows the principal sub-activities within a CBSE process.

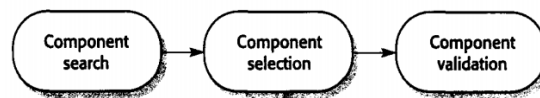


Figure 6.4: The CBSE process [1]

Some of the activities within this process, such as the initial discovery of user requirements, are carried out in the same way as in other software processes. However, the essential differences between this process and software processes based on original software development are:

1. The user requirements are initially developed in outline rather than in detail, and stakeholders are encouraged to be as flexible as possible in defining their requirements. The reason for this is that very specific requirements limit the number of components that might meet these requirements. Unlike incremental development, however, you need a complete set of requirements so that you can identify as many components as possible for reuse.
2. Requirements are refined and modified early in the process depending on the components available. If the user requirements cannot be satisfied from available components, you should discuss the related requirements that can be supported. Users may be willing to change their minds if this means cheaper or quicker system delivery.
3. There is a further component search and design refinement activity after the system architecture has been designed. Some apparently usable components may turn out to be unsuitable or do not work properly with other chosen components. This implies that further requirements changes may be necessary.
4. Development is a composition process where the discovered components are integrated. This involves integrating the components with the component model infrastructure and, often, developing 'glue code' to reconcile the interfaces of incompatible components. Of course, additional functionality may be required over and above that provided by usable components. Naturally, you should develop this as components that can be reused in future systems.

One activity that is unique to the CBSE process is component identification. This involves a number of sub-activities, as shown in Figure 6.4. There are two stages in the CBSE process where you have to

identify components for possible use in the system. In the early stage, your focus should be on search and selection. You need to convince yourself that there are components available to meet your requirements.

Obviously, you should do some initial checking that the component is suitable but detailed testing may not be required. In the later stage, after the system architecture has been designed, you should spend more time on component validation. You need to be confident that the identified components are really suited to your application; if not, then you have to repeat the search and selection processes.

The first stage in identifying components is to look for components that are available locally or from trusted suppliers. The vision of advocates of CBSE is that there should be a viable component market place where external vendors compete to provide components. At the time of this writing, this has not emerged to any significant extent. The main reason for this is that users of external components face risks that these components will not work as advertised. If this is the case, the costs of reuse exceed the benefits, and few project managers believe that the risks are worth taking. Another important reason why component markets have not developed is that many components are in specialized application domains. There is not a sufficiently large market in these domains for external component suppliers to establish a viable, long-term business.

As a consequence, component search is often confined to a software development organization. Software development companies can build their own database of reusable components without the risks inherent in using components from external suppliers.

Once the component search process has identified candidate components, specific components from this list have to be selected. In some cases, this will be a straightforward task. Components on the list will map directly onto the user requirements, and there will not be competing components that match these requirements. In other cases, however, the selection process is much more complex. There will not be a clean mapping of requirements to components, and you will find that several components have to be used to meet a specific requirement or group of requirements.

Unfortunately, it is likely that different requirements will require different groups of components, so you have to decide which component compositions provide the best coverage of the requirements.

Once you have selected components for possible inclusion in a system, you should validate them to check that they behave as advertised. The extent of the validation required depends on the source of the components. If you are using a component that has been developed by a known and trusted source, you may decide that separate component testing is unnecessary and you test the component when it is integrated with other components. On the other hand, if you are using a component from an unknown source, you should always check and test that component before including it in your system.

Component validation involves developing a set of test cases for the component (or, possibly, extending test cases supplied with the component) and developing a test harness to run the component tests. The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests. Components are usually specified informally, with the only formal documentation being their interface specification. This may not include enough information for you to develop a complete set of tests that would convince you that the components advertised interface is what you require.

A further validation problem, which may arise at this stage, is that the component may have features that could interfere with your use of the component. Reusable components will often have more functionality than you need. You can simply ignore the unwanted functionality, but it can sometimes interfere with other components or with the system as a whole. In some cases, the unwanted functionality can even cause serious system failures.

### 6.3. Component Composition

Component composition is the process of assembling components to create a system. If we assume a situation where reusable components are available, then most systems will be constructed by composing these reusable components with each other, with specially written components and with the component support infrastructure provided by the model framework. This infrastructure provides facilities to support component communication and horizontal services such as user interface services, transaction management, concurrency and security.

Composition is not a simple operation; there are a number of types (see Figure 6.5):

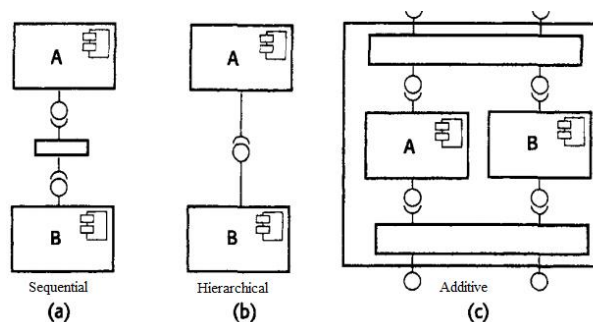


Figure 6.5: Types of composition [1]

- **Sequential composition.** This occurs when, in the composite component, the constituent components are executed in sequence. It corresponds to situation (a) in Figure 6.5, where the *provides-interface* of each component are composed. Some extra code is required to make the link between the components.
- **Hierarchical composition.** This occurs when one component calls directly on the services provided by another component. It corresponds to a situation where the *provides-interface* of one component is composed with the *requires-interface* of another component. This is situation (b) in Figure 6.5.
- **Additive composition.** This occurs when the interfaces of two or more components are put together (added) to create a new component. The interfaces of the composite component are created by putting together all of the interfaces of the constituent components, with duplicate operations removed if necessary. This corresponds to situation (c) in Figure 6.5.

You might use all the forms of component composition when creating a system. In all cases, you may have to write 'glue code' that links the components. For example, for sequential composition, the output of component A typically becomes the input to component B. You need intermediate statements that call component A, collect the result and then call component B with that result as a parameter.

When you write components especially for composition, you design the interfaces of these components so that they are compatible. You can therefore easily compose these components into a single unit. However, when components are developed independently for reuse, you will often be faced with interface incompatibilities where the interfaces of the components that you wish to compose are not the same. Three types of incompatibility can occur:

- ***Parameter incompatibility.*** The operations on each side of the interface have the same name but their parameter types or the numbers of parameters are different.
- ***Operation incompatibility.*** The names of the operations in the provides-interface and requires-interface are different.
- ***Operation incompleteness.*** The provides-interface of a component is a subset of the requires-interface of another component or vice versa.

In all cases, you tackle the problem of incompatibility by writing an adaptor component that reconciles the interfaces of the two components being reused. When you know the interfaces of the components that you want to use, you write an adaptor component that converts one interface to another. The precise form of the adaptor depends on the type of composition. Sometimes, as in the next example, the adaptor simply takes a result from one component and converts it into a form where it can be used as an input to another. In other cases, the adaptor may be called by component A, and itself calls component B. This latter situation would arise if A and B were compatible but the number of parameters in their interfaces was different.

Another case in which an adaptor component may be used is where one component wishes to make use of another, but there is an incompatibility between the provides-interface and requires-interface of these components.

When you create a system by composing components, you may find that there are potential conflicts between functional and non-functional requirements, the need to deliver a system as quickly as possible, and the need to create a system that can evolve as requirements change. The decisions where you may have to make trade-offs are:

- What composition of components is most effective in delivering the functional requirements for the system?
- What composition of the components will allow adaptations for future changes to the requirements?
- What will be the emergent properties of the composed system? These emergent properties are properties such as performance and dependability. You can only assess these once the complete system is implemented.

Unfortunately, there are many situations where the solutions to the composition problems are mutually conflicting. For example, consider a situation such as that illustrated in Figure 6.6, where a system can be created through two alternative compositions. The system is a data collection and reporting system where data is collected from different sources, stored in a database, and a different report summarizing that data is produced.

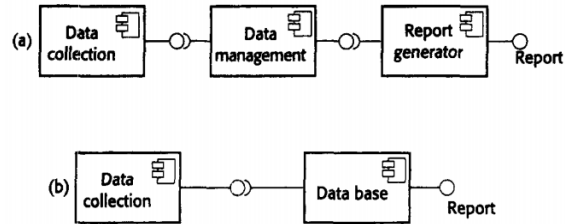


Figure 6.6: Data collection and report generation components [1]

The advantages of composition (a) are that reporting and data management are separate, so there is more flexibility for future change. The data management system could be replaced and, if reports are required that the current reporting component cannot produce, that component can also be replaced. In composition (b), a database component with built-in reporting facilities (e.g., Microsoft Access) is used.

The advantages of composition (b) are that there are fewer components, so this will probably be faster because there are no component communication overheads. Furthermore, data integrity rules that apply to the database will also apply to reports. These reports will not be able to combine data in incorrect ways. In composition (a), there are no such constraints, so errors in reports are more likely.

*In general, a good composition principle to follow is the principle of separation of concerns. That is, you should try to design your system in such a way that each component has a clearly defined role and that, ideally, these roles should not overlap. However, it may be cheaper to buy one multifunctional component rather than two or three separate components. Furthermore, there may be dependability or performance penalties when multiple components are used*

## REFERENCE

[1] I. Sommerville , 2008. Software Engineering. Eighth Edition, Addison-Wesley

## 7. VERIFICATION AND VALIDATION

During and after the implementation process, the program being developed must be checked to ensure that it meets its specification and delivers the functionality expected by the people paying for the software. Verification and validation (V&V) is the name given to these checking and analysis processes. Verification and activities take place at each stage of the software process. V&V starts with requirements reviews and continues through design reviews and code inspections to product testing.

Verification and validation are not the same thing, although they are often confused.

- **Validation:** Are we building the right product?
- **Verification:** Are we building the product right?

These definitions tell us that the role of verification involves checking that the software conforms to its specification. You should check that it meets its specified functional and non-functional requirements. Validation, however, is a more general process. The aim of validation is to ensure that the software system meets the customer's expectations. It goes beyond checking that the system conforms to its specification to showing that the software does what the customer expects it to do. Software system specifications do not always reflect the real wishes or needs of users and system owners.

The ultimate goal of the verification and validation process is to establish confidence that the software system is 'fit for purpose'. This means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users and the current marketing environment for the system:

- **Software function.** The level of confidence required depends on how critical the software is to an organization. For example, the level of confidence required for software that is used to control a safety-critical system is very much higher than that required for a prototype software system that has been developed to demonstrate some new ideas.
- **User expectations.** It is a sad reflection on the software industry that many users have low expectations of their software and are not surprised when it fails during use. They are willing to accept these system failures when the benefits of use outweigh the disadvantages. However, user tolerance of system failures has been decreasing since the 1990s. It is now less acceptable to deliver unreliable systems, so software companies must devote more effort to verification and validation.
- **Marketing environment.** When a system is marketed, the sellers of the system must take into account competing programs, the price those customers are willing to pay for a system and the required schedule for delivering that system. Where a company has few competitors, it may decide to release a program before it has been fully tested and debugged because they want to be the first into the market. Where customers are not willing to pay high prices for software, they may be willing to tolerate more software faults. All of these factors must be considered when deciding how much effort should be spent on the V&V process.

Within the V&V process, there are two complementary approaches to system checking and analysis:

1. **Software inspections or peer reviews** analyze and check system representations such as the requirements document, design diagrams and the program source code. You can use inspections

at all stages of the process. Inspections may be supplemented by some automatic analysis of the source text of a system or associated documents. Software inspections and automated analyses are static V&V techniques, as you don't need to run the software on a computer.

2. **Software testing** involves running an implementation of the software with test data. You examine the output of the software and its operational behavior to check that it is performing as required. Testing is a dynamic technique of verification and validation.

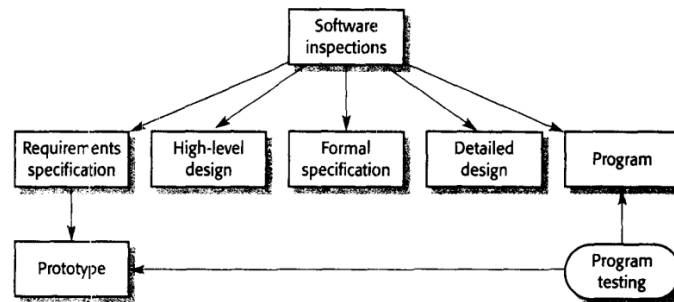


Figure 7.1: Static and dynamic verification and validation [1]

Figure 7.1 shows that software inspections and testing play complementary roles in the software process. The arrows indicate the stages in the process where the techniques may be used. Therefore, you can use software inspections at all stages of the software process. Starting with the requirements, any readable representations of the software can be inspected. Requirements and design reviews are the main techniques used for error detection in the specification and design.

You can only test a system when a prototype or an executable version of the program is available. An advantage of incremental development is that a testable version of the system is available at a fairly early stage in the development process. Functionality can be tested as it is added to the system so you don't have to have a complete implementation before testing begins.

Inspection techniques include program inspections, automated source code analysis and formal verification. However, static techniques can only check the correspondence between a program and its specification (verification); they cannot demonstrate that the software is operationally useful. You also can't use static techniques to check emergent properties of the software such as its performance and reliability.

Although software inspections are now widely used, program testing will always be the main software verification and validation technique. Testing involves exercising the program using data like the real data processed by the program. You discover program defects or inadequacies by examining the outputs of the program and looking for anomalies. There are two distinct types of testing that may be used at different stages in the software process:

1. **Validation testing** is intended to show that the software is what the customer wants—that it meets its requirements. As part of validation testing, you may use statistical testing to test the programs performance and reliability, and to check how it works under operational conditions.
2. **Defect testing** is intended to reveal defects in the system rather than to simulate its operational use. The goal of defect testing is to find inconsistencies between a program and its specification



Of course, there is no hard-and-fast boundary between these approaches to testing. During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

The processes of V&V and debugging are normally interleaved. As you discover faults in the program that you are testing, you have to change the program to correct these faults. However, testing (or, more generally verification and validation) and debugging have different goals:

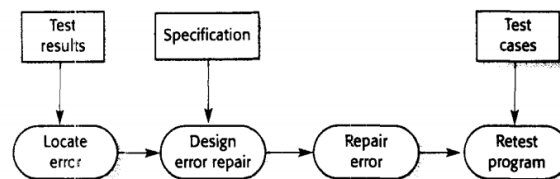


Figure 7.2: The debugging process [1]

1. **Verification and validation** processes are intended to establish the existence of defects in a software system.
2. **Debugging** is a process (see Figure 7.2) that locates and corrects these defects

There is no simple method for program debugging. Skilled debuggers look for patterns in the test output where the defect is exhibited and use their knowledge of the type of defect, the output pattern, the programming language and the programming process to locate the defect. When you are debugging, you can use your knowledge of common programmer errors (such as failing to increment a counter) and match these against the observed patterns. You should also look for characteristic programming language errors, such as pointer misdirection in C.

Locating the faults in a program is not always a simple process, since the fault may not be close to the point where the program failed. To locate a program fault, you may have to design additional tests that reproduce the original fault and that pinpoint its location in the program. You may have to trace the program manually, line by line. Debugging tools that collect information about the programs execution may also help you locate the source of a problem.

Interactive debugging tools are generally part of a set of language support tools that are integrated with a compilation system. They provide a specialized run-time environment for the program that allows access to the compiler symbol table and, from there, to the values of program variables. You can control execution by 'stepping' through the program statement by statement. After each statement has been executed, you can examine the values of variables and so discover the location of the fault.

After a defect in the program has been discovered, you have to correct it and revalidate the system. This may involve re-inspecting the program or regression testing where existing tests are executed again. Regression testing is used to check that the changes made to a program have not introduced new faults. Experience has shown that a high proportion of fault 'repairs are either incomplete or introduce new faults into the program.

In principle, you should repeat all tests after every defect repair; in practice, this is usually too expensive. As part of the test plan, you should identify dependencies between components and the tests associated

with each component. That is, there should be traceability from the test cases to the components that are tested. If this traceability is documented, you may then run a subset of the system test cases to check the modified component and its dependents.

## 7.1. Planning Verification and Validation

Verification and validation is an expensive process. For some systems, such as real-time systems with complex non-functional constraints, more than half the system development budget may be spent on V&V. Careful planning is needed to get the most out of inspections and testing and to control the costs of the verification and validation process.

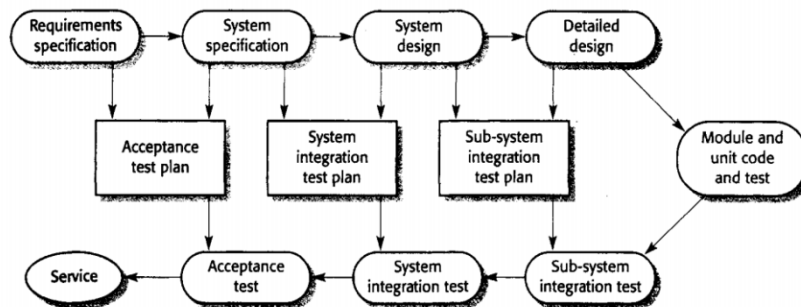


Figure 7.3: Test plans as a link between development and testing [1]

You should start planning system validation and verification early in the development process. The software development process model shown in Figure 7.3 is sometimes called the V-model. It is an instantiation of the generic waterfall model and shows that test plans should be derived from the system specification and design. This model also breaks down system V&V into a number of stages. Each stage is driven by tests that have been defined to check the conformance of the program with its design and specification.

As part of the V&V planning process, you should decide on the balance between static and dynamic approaches to verification and validation, draw up standards and procedures for software inspections and testing, establish checklists to drive program inspections and define the software test plan.

The relative effort devoted to inspections and testing depends on the type of system being developed and the organizational expertise with program inspection. As a general rule, the more critical a system, the more effort should be devoted to static verification techniques.

Test planning is concerned with establishing standards for the testing process, not just with describing product tests. As well as helping managers allocate resources and estimate testing schedules, test plans are intended for software engineers involved in designing and carrying out system tests. They help technical staff get an overall picture of the system tests and place their own work in this context.

The major components of a test plan for a large and complex system are shown in Figure 7.4. As well as setting out the testing schedule and procedures, the test plan defines the hardware and software resources that are required. This is useful for system managers who are responsible for ensuring that these resources are available to the testing team. Test plans should normally include significant amounts of contingency

so that slip pages in design and implementation can be accommodated and staff redeployed to other activities (See Table 7.1).

| Test steps                         | Description   |
|------------------------------------|---|
| The testing process                | A description of the major phases of the testing process. These might be as described earlier in this chapter.  |
| Requirement traceability           | Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.  |
| Tested items                       | The products of the software process that are to be tested should be specified.   |
| Testing schedule                   | An overall testing schedule and resource allocation for this schedule is, obviously, linked to the more general project development schedule.   |
| Test recording procedures          | It is not enough simply to run tests; the results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it has been carried out correctly. |
| Hardware and software requirements | This section should set out the software tools required and estimated hardware utilization.   |
| Constraints                        | Constraints affecting the testing process such as staff shortages should be anticipated in this section.  |

Table 7.1: The structure of a software test plan [1]

For smaller systems, a less formal test plan may be used, but there is still a need for a formal document to support the planning of the testing process. For some agile processes such as extreme programming, testing is inseparable from development. Like other planning activities, test planning is also incremental. In XP, the customer is ultimately responsible for deciding how much effort should be devoted to system testing.

Test plans are not a static document but evolve during the development process. Test plans change because of delays at other stages in the development process. If part of a system is incomplete, the system as a whole cannot be tested. You then have to revise the test plan to redeploy the testers to some other activity and bring them back when the software is once again available.

## 7.2. Software Inspections

*Software inspection is a static V&V process in which a software system is reviewed to find errors, omissions, and anomalies.* Generally, inspections focus on source code. But any readable representation of the software such as its requirements or a design model can be inspected. When you inspect a system, you use knowledge of the system, its application domain and the programming language or design model to discover errors.

There are three major advantages of inspection over testing:

1. During testing, errors can mask (hide) other errors. Once one error is discovered, you can never be sure if other output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.
2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.
3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program such as compliance with standards, portability and maintainability. You can look for inefficiencies, inappropriate algorithms and poor programming style that could make the system difficult to maintain and update.

Inspections are an old idea. There have been several studies and experiments that have demonstrated that inspections are more effective for defect discovery than program testing. It has been reported that more than 60% of the errors in a program can be detected using informal program inspections. It has been suggested that a more formal approach to inspection based on correctness arguments can detect more than 90% of the errors in a program. When empirically compared the effectiveness of inspections and testing, the result showed that static code reviewing was more effective and less expensive than defect testing in discovering program faults.

*Reviews and testing each have advantages and disadvantages* and should be used together in the verification and validation process. Indeed, one of the most effective uses of reviews is to review the test cases for a system. Reviews can discover problems with these tests and can help design more effective ways to test the system. You can start system V&V with inspections early in the development process, but once a system is integrated, you need testing to check its emergent properties and that the system's functionality is what the owner of the system really wants.

In spite of the success of inspections, it has proven to be difficult to introduce formal inspections into many software development organizations. Software engineers with experience of program testing are sometimes reluctant to accept that inspections can be more effective for defect detection than testing. Managers may be suspicious because inspections require additional costs during design and development. They may not wish to take the risk that there will be no corresponding savings during program testing.

There is no doubt that inspections 'front-load' software V&V costs and result in cost savings only after the development teams become experienced in their use. Furthermore, there are the practical problems of arranging inspections: Inspections take time to arrange and appear to slow down the development process. It is difficult to convince a hard-pressed manager that this time can be made up later because less time will be spent on program debugging.

## **THE PROGRAM INSPECTION PROCESS**

Program inspections are reviews whose objective is program defect detection. The notion of a formalized inspection process was first developed at IBM in the 1970s. It is now a fairly widely used method of program verification, especially in critical systems engineering. These are all based on a team with members from different backgrounds making a careful, line-by-line review of the program source code.

The key difference between program inspections and other types of quality review is that the specific goal of inspections is to find program defects rather than to consider broader design issues. Defects may be logical errors, anomalies in the code that might indicate an erroneous condition or non compliance with organizational or project standards. By contrast, other types of review may be more concerned with schedule costs, progress against defined milestones or assessing whether the software is likely to meet organizational goals.

The program inspection is a formal process that is carried out by a team of at least four people, such as author, reader, tester and moderator. Team members systematically analyze the code and point out possible defects. The reader reads the code aloud to the inspection team; the tester inspects the code from a testing perspective and the moderator organizes the process.

As organizations have gained experience with inspection, other proposals for team roles have emerged. In a discussion of how inspection was successfully introduced in Hewlett-Packard's development process, it had been suggested six roles, as shown in Table 7.2. They do not think that reading the program aloud is necessary. The same person can take more than one role so the team size may vary from one inspection to another. Further it has been suggested that inspectors should be selected to reflect different viewpoints such as testing, end-user and quality management.

| Role                  | Description   |
|-----------------------|---|
| Author or owner       | The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.    |
| Inspector             | Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team |
| Reader                | Presents the code or document at an inspection meeting.   |
| Scribe                | Records the results of the inspection meeting.  |
| Chairman or moderator | Manages the process and facilitates the inspection. Reports process results to the chief moderator..  |
| Chief moderator       | Responsible for inspection process improvements, checklist updating, standards development, etc.  |

Table 7.2: Roles in the inspection process [1]

The activities in the inspection process are shown in Table 7.2. .Before a program inspection process begins, it is essential that:

- You have a precise specification of the code to be inspected. It is impossible to inspect a component at the level of detail required to detect defects without a complete specification.
- The inspection team members are familiar with the organizational standards.
- An up-to-date, compliable version of the code has been distributed to all team members. There is no point in inspecting code that is 'almost complete' even if a delay causes schedule disruption.

The inspection team moderator is responsible for inspection planning. This involves selecting an inspection team, organizing a meeting room and ensuring that the material to be inspected and its specifications are complete (See Figure 7.4). The program to be inspected is presented to the inspection team during the overview stage when the author of the code describes what the program is intended to do. This is followed by a period of individual preparation. Each inspection team member studies the specification and the program and looks for defects in the code.

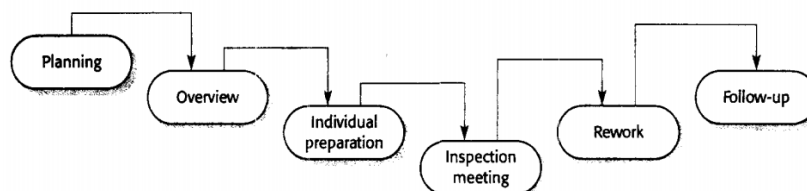


Figure 7.4: The inspection process [1]

The inspection itself should be fairly short (no more than two hours) and should focus on defect detection, standards conformance and poor-quality programming.

The inspection team should not suggest how these defects should be corrected nor should it recommend changes to other components.

Following the inspection, the programs author should make changes to it to correct the identified problems. In the follow-up stage, the moderator should decide whether a re-inspection of the code is required. He or she may decide that a complete re-inspection is not required and that the defects have been successfully fixed. The program is then approved by the moderator for release.

During an inspection, a checklist of common programmer errors is often used to focus the discussion (See Table 7.3). This checklist can be based on checklist examples from books or from knowledge of defects that are common in a particular application domain. You need different checklists for different programming languages because each language has its own characteristic errors.

This checklist varies according to programming language because of the different levels of checking provided by the language compiler. For example, a Java compiler checks that functions have the correct number of parameters, a C compiler does not. Possible checks that might be made during the inspection process are shown in Table 7.3. *It has been emphasized that each organization should develop its own inspection checklist based on local standards and practices. Checklists should be regularly updated as new types of defects are found.*

| Fault class                 | Inspection check   |
|-----------------------------|--|
| Data faults                 | <ul style="list-style-type: none"> <li>• Are all program variables initialized before their values are used?</li> <li>• Have all constants been named?</li> <li>• Should the upper bound of arrays be equal to the size of the array or Size-1?</li> <li>• If character strings are used, is a delimiter explicitly assigned?</li> <li>• Is there any possibility of buffer overflow?</li> </ul> |
| Control faults              | <ul style="list-style-type: none"> <li>• For each conditional statement, is the condition correct?</li> <li>• Is each loop certain to terminate?</li> <li>• Are compound statements correctly bracketed?</li> <li>• In case statements, are all possible cases accounted for?</li> <li>• If a break is required after each case in case statements, has it been included?</li> </ul>             |
| Input/ Output faults        | <ul style="list-style-type: none"> <li>• Are all input variables used?</li> <li>• Are all output variables assigned a value before they are output?</li> <li>• Can unexpected inputs cause corruption?.</li> </ul>   |
| Interface faults            | <ul style="list-style-type: none"> <li>• Do all function and method calls have the correct number of parameters?</li> <li>• Do formal and actual parameter types match?</li> <li>• Are the parameters in the right order?</li> <li>• If components access shared memory, do they have the same model of the shared memory structure?</li> </ul>  |
| Storage management faults   | <ul style="list-style-type: none"> <li>• If a linked structure is modified, have all links been correctly reassigned?</li> <li>• If dynamic storage is used, has space been allocated correctly?</li> <li>• Is space explicitly de-allocated after it is no longer required?</li> </ul>  |
| Exception management faults | <ul style="list-style-type: none"> <li>• Have all possible error conditions been taken into account?</li> </ul>  |

Table 7.3: Possible checks during inspection process [1]

The time needed for an inspection and the amount of code that can be covered depends on the experience of the inspection team, the programming language and the application domain. It has been found that:

- About 500 source code statements per hour can be presented during the overview stage.
- During individual preparation, about 125 source code statements per hour can be examined.
- From 90 to 125 statements per hour can be inspected during the inspection meeting.

With four people involved in an inspection team, the cost of inspecting 100 lines of codes is roughly equivalent to one person-day of effort. This assumes that the inspection itself takes about an hour and that

each team member spends one to two hours preparing for the inspection. Testing costs vary widely and depend on the number of faults in the program. However, the effort required for the program inspection is probably less than half the effort that would be required for equivalent defect testing

Some organizations have now abandoned component testing in favor of inspections. They have found that program inspections are so effective at finding errors that the costs of component testing are not justifiable. These organizations found that inspections of components, combined with system testing, were the most cost-effective V&V strategy.

The introduction of inspections has implications for project management. Sensitive management is important if inspections are to be accepted by software development teams. Program inspection is a public process of error detection compared with the more private component testing process. Inevitably, mistakes that are made by individuals are revealed to the whole programming team. Inspection team leaders must be trained to manage the process carefully and to develop a culture that provides support without blame when errors are discovered.

As an organization gains experience of the inspection process, it can use the results of inspections to help with process improvement. Inspections are an ideal way to collect data on the type of defects that occur. The inspection team and the authors of the code that was inspected can suggest reasons why these defects were introduced. Wherever possible, the process should then be modified to eliminate the reasons for defects so they can be avoided in future systems.

### 7.3. Verification and Formal Methods

*Formal methods of software development are based on mathematical representations of the software, usually as a formal specification.* These formal methods are mainly concerned with a mathematical analysis of the specification; with transforming the specification to a more detailed, semantically equivalent representation; or with formally verifying that one representation of the system is semantically equivalent to another representation.

You can think of the use of formal methods as the ultimate static verification technique. They require very detailed analyses of the system specification and the program, and their use is often time consuming and expensive. Consequently, the use of formal methods is mostly confined to safety- and security-critical software development processes.

Formal methods may be used at different stages in the V&V process:

- A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors and omissions.
- You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification and is effective in discovering programming and some design errors. A transformational development process where a formal specification is transformed through a series of more detailed representations or a *Cleanroom* process may be used to support the formal verification process.

The argument for the use of formal specification and associated program verification is that formal specification forces a detailed analysis of the specification. It may reveal potential inconsistencies or

omissions that might not otherwise be discovered until the system is operational. Formal verification demonstrates that the developed program meets its specification so implementation errors do not compromise dependability

The argument against the use of formal specification is that it requires specialized notations. These can only be used by specially trained staff and cannot be understood by domain experts. Hence, problems with the system requirements can be concealed by formality. Software engineers cannot recognize potential difficulties with the requirements because they don't understand the domain; domain experts cannot find these problems because they don't understand the specification. Although the specification may be mathematically consistent, it may not specify the system properties that are really required.

Verifying a non trivial software system takes a great deal of time and requires specialized tools such as theorem provers and mathematical expertise. It is therefore an extremely expensive process and, as the system size increases, the costs of formal verification increase disproportionately. Many people therefore think that formal verification is not cost-effective. The same level of confidence in the system can be achieved more cheaply by using other validation techniques such as inspections and system testing.

It is sometimes claimed that the use of formal methods for system development leads to more reliable and safer systems. There is no doubt that a formal system specification is less likely to contain anomalies that must be resolved by the system designer. However, formal specification and proof do not guarantee that the software will be reliable in practical use. The reasons for this are:

- The specification may not reflect the real requirements of system users. It has been discovered that many failures experienced by users were a consequence of specification errors and omissions that could not be detected by formal system specification. Furthermore, system users rarely understand formal notations so they cannot read the formal specification directly to find errors and omissions.
- The proof may contain errors. Program proofs are large and complex, so, like large and complex programs, they usually contain errors.
- The proof may assume a usage pattern which is incorrect. If the system is not used as anticipated, the proof may be invalid.

*In spite of their disadvantages, formal methods have an important role to play in the development of critical software systems. Formal specifications are very effective in discovering specification problems that are the most common causes of system failure. Formal verification increases confidence in the most critical components of these systems. The use of formal approaches is increasing as procurers demand it and as more and more engineers become familiar with these techniques.*

## **CLEANROOM SOFTWARE DEVELOPMENT**

A model of the Cleanroom process is shown in Figure 7.5. The objective of this approach to software development is zero-defect software. The name 'Cleanroom' was derived by analogy with semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.



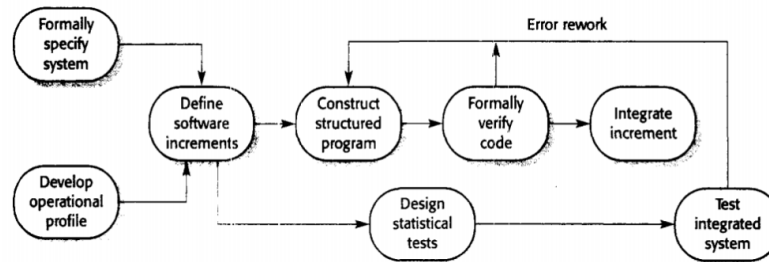


Figure 7.5: The cleanroom development process [1]

The Cleanroom approach to software development is based on five key strategies:

1. **Formal specification.** The software to be developed is formally specified. A state transition model that shows system responses to stimuli is used to express the specification.
2. **Incremental development.** The software is partitioned into increments that are developed and validated separately using the Cleanroom process. These increments are specified, with customer input, at an early stage in the process.
3. **Structured programming.** Only a limited number of control and data abstraction constructs are used. The program development process is a process of stepwise refinement of the specification. A limited number of constructs are used and the aim is to systematically transform the specification to create the program code.
4. **Static verification.** The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
5. **Statistical testing of the system.** The integrated software increment is tested statistically to determine its reliability. These statistical tests are based on an operational profile, which is developed in parallel with the system specification.

There are three teams involved when the Cleanroom process is used for large system development:

1. **The specification team.** This group is responsible for developing and maintaining the system specification. This team produces customer-oriented specifications (the user requirements definition) and mathematical specifications for verification. In some cases, when the specification is complete, the specification team also takes responsibility for development.
2. **The development team.** This team has the responsibility of developing and verifying the software. The software is not executed during the development process. A structured, formal approach to verification based on inspection of code supplemented with correctness arguments is used.
3. **The certification team.** This team is responsible for developing a set of statistical tests to exercise the software after it has been developed. These tests are based on the formal specification. Test case development is carried out in parallel with software development. The test cases are used to certify the software reliability. Reliability growth models may be used to decide when to stop testing.

*Use of the Cleanroom approach has generally led to software with very few errors. The costs of Cleanroom projects are comparable with other projects that use conventional development techniques.*

The approach to incremental development in the Cleanroom process is to deliver critical customer functionality in early increments. Less important system functions are included in later increments. The customer therefore has the opportunity to try these critical increments before the whole system has been delivered. If requirements problems are discovered, the customer feeds back this information to the development team and requests a new release of the increment.

As with extreme programming, this means that the most important customer functions receive the most validation. As new increments are developed, they are combined with the existing increments and the integrated system is tested. Therefore, existing increments are re-tested with new test cases as new system increments are added.

Rigorous program inspection is a fundamental part of the Cleanroom process. A state model of the system is produced as a system specification. This is refined through a series of more detailed system models to an executable program. The approach used for development is based on well-defined transformations that attempt to preserve the correctness at each transformation to a more detailed representation. At each stage, the new representation is inspected, and mathematically rigorous arguments are developed that demonstrate that the output of the transformation is consistent with its input.

The mathematical arguments used in the Cleanroom process are not, however, formal proofs of correctness. Formal mathematical proofs that a program is correct with respect to its specification are too expensive to develop. They depend on using knowledge of the formal semantics of the programming language to construct theories that relate the program and its formal specification. These theories must then be proven mathematically, often with the assistance of large and complex theorem-prover programs. Because of their high cost and the specialist skills that are needed, proofs are usually developed only for the most safety-or security-critical applications.

Inspection and formal analysis has been found to be very effective in the Cleanroom process. The vast majority of defects are discovered before execution and are not introduced into the developed software. On average, only 2.3 defects per thousand lines of source code are discovered during testing for Cleanroom projects. Overall development costs are not increased because less effort is required to test and repair the developed software.

*Most teams could successfully use the Cleanroom method.* The programs produced can be of higher quality than those developed using traditional techniques, i.e., the source code with more comments and a simpler structure. Moreover, the Cleanroom teams meet the development schedule.

Cleanroom development works when practiced by skilled and committed engineers. Reports of the success of the Cleanroom approach in industry have mostly, though not exclusively, come from people already committed to it. We don't know whether this process can be transferred effectively to other types of software development organizations. These organizations may have less committed and less skilled engineers. Transferring the Cleanroom approach or, indeed, any other approach where formal methods are used, to less technically advanced organizations still remains a challenge.

## 7.1. Critical System Verification and Validation

The verification and validation of a critical system has, obviously, much in common with the validation of any other system. The V&V processes should demonstrate that the system meets its specification, and that the system services and behavior support the customers' requirements. However, for critical systems, where a high level of dependability is required, additional testing and analysis are required to produce evidence that the system is trustworthy. There are two reasons why you should do this:

1. **Costs of failure:** The costs and consequences of critical systems failure are potentially much greater than for non critical systems. You lower the risks of system failure by spending more on system verification and validation. It is usually cheaper to find and remove faults before the system is delivered than to pay for the consequent costs of accident so disruptions to system service.
2. **Validation of dependability attributes:** You may have to make a formal case to customers that the system meets its specified dependability requirements (availability, reliability, safety and security). Assessing these dependability characteristics requires specific V&V activities discussed. In some cases, external regulators such as national aviation authorities may have to certify that the system is safe before it can be deployed. To obtain this certification, you may have to design and carry out special V&V procedures that collect evidence about the system's dependability.

*For these reasons, the costs of V&V for critical systems are usually much higher than for other classes of system. It is normal for V&V to take up more than 50% of the total development costs for critical software systems. This cost is, of course, justified, if an expensive system failure is avoided.*

Although the critical systems validation process mostly focuses on validating the system, related activities should verify that defined system development processes have been followed. System quality is affected by the quality of processes used to develop the system. In short, good processes lead to good systems. Therefore, to produce dependable systems, you need to be confident that a sound development process has been followed.

This process assurance is an inherent part of the ISO 9000 standards for quality management. These standards require documentation of the processes that are used and associated activities that ensure that these processes have been followed. This normally requires the generation of process records, such as Signed forms, that certify the completion of process activities and product quality checks. ISO 9000 standards specify what tangible process outputs should be produced and who is responsible for producing them.

### RELIABILITY VALIDATION

A number of metrics are available to specify a system's reliability requirements. To validate that the system meets these requirements, you have to measure the reliability of the system as seen by a typical system user. The process of measuring the reliability of a system is illustrated in Figure 7.6. This process involves four stages:

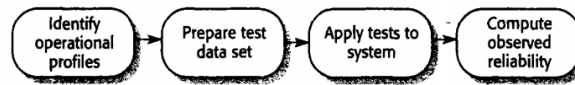


Figure 7.6: The reliability measurement process [1]

1. You start by studying existing systems of the same type to establish an operational profile. An operational profile identifies classes of system inputs and the probability that these inputs will occur in normal use.
2. You then construct a set of test data that reflect the operational profile. This means that you create test data with the same probability distribution as the test data for the systems that you have studied. Normally, you use a test data generator to support this process.
3. You test the system using these data and the count the number and type of failures that occur. The times of these failures are also logged.
4. After you have observed a statistically significant number of failures, you can compute the software reliability and work out the appropriate reliability metric value.

This approach is sometimes called **statistical testing**. The aim of statistical testing is to assess system reliability. This contrasts with defect testing, where the aim is to discover system faults..

This conceptually attractive approach to reliability measurement is not easy to apply in practice. The principal difficulties that arise are due to:

- **Operational profile uncertainty.** The operational profiles based on experience with other systems may not be an accurate reflection of the real use of the system.
- **High costs of test data generation.** It can be very expensive to generate the large volume of data required in an operational profile unless the process can be totally automated.
- **Statistical uncertainty when high reliability is specified.** You have to generate a statistically significant number of failures to allow accurate reliability measurements. When the software is already reliable, relatively few failures occur and it is difficult to generate new failures.

Developing an accurate operational profile is certainly possible for some types of systems, such as telecommunication systems, that have a standardized pattern of use. For other system types, however, there are many different users who each have their own ways of using the system. Moreover, different users can get quite different impressions of reliability because they use the system in different ways.

By far the best way to generate the large data set required for reliability measurement is to use a test data generator that can be setup to automatically generate inputs matching the operational profile. However, it is not usually possible to automate the production of all test data for interactive systems because the inputs are often a response to system outputs. Data sets for these systems have to be generated manually, with correspondingly higher costs. Even where complete automation is possible, writing commands for the test data generator may take a significant amount of time.

Statistical uncertainty is a general problem in measuring the reliability of a system. To make an accurate prediction of reliability, you need to do more than simply cause a single system failure. You have to generate a reasonably large, statistically significant number of failures to be confident that your reliability measurement is accurate. The better you get at minimizing the number of faults in a system, the harder it

becomes to measure the effectiveness of fault minimization techniques. If very high levels of reliability are specified, it is often impractical to generate enough system failures to check these specifications.

## **SAFETY ASSURANCE**

The processes of safety assurance and reliability validation have different objectives. You can specify reliability quantitatively using some metric and then measure the reliability of the completed system. Within the limits of the measurement process, you know whether the required level of reliability has been achieved. Safety, however, cannot be meaningfully specified in a quantitative way and so cannot be measured when a system is tested.

Safety assurance is therefore concerned with establishing a confidence level in the system that might vary from 'very low' to 'very high'. This is a matter for professional judgment based on a body of evidence about the system, its environment and its development processes. In many cases, this confidence is partly based on the experience of the organization developing the system. If a company has previously developed a number of control systems that have operated safely, then it is reasonable to assume that they will continue to develop safe systems of this type.

However, such an assessment must be backed-up by tangible evidence from the system design, the results of system V&V, and the system development processes that have been used. For some systems, this tangible evidence is assembled in a safety case that allows an external regulator to come to a conclusion of the developers confidence in the system's safety is justified.

The V&V processes for safety-critical systems have much in common with the comparable processes of any other systems with high reliability requirements. There must be extensive testing to discover as many defects as possible, and where appropriate, statistical testing may be used to assess the system reliability.

However, because of the ultra-low failure rates required in many safety-critical systems, statistical testing cannot always provide a quantitative estimate of the system reliability. The tests simply provide some evidence, which is used with other evidence such as the results of reviews and static checking, to make a judgment about the system safety.

Extensive reviews are essential during a safety-oriented development process to expose the software to people who will look at it from different perspectives. The following five types of review should be mandatory for safety-critical systems:

1. review for correct intended function
2. review for maintainable, understandable structure
3. review to verify that the algorithm and data structure design are consistent with the specified behavior
4. review the consistency of the code and the algorithm and data structure design
5. review the adequacy of the system test cases

An assumption that underlies work in system safety is that the number of system faults that can lead to safety-critical hazards is significantly less than the total number of faults that may exist in the system. Safety assurance can concentrate on these faults with hazard potential. If it can be demonstrated that these

faults cannot occur or, if they do, the associated hazard will not result in an accident, then the system is safe. This is the basis of safety arguments.

## SECURITY ASSESSMENT

The assessment of system security is becoming increasingly important as more and more critical systems are Internet-enabled and can be accessed by anyone with a network connection. There are daily stories of attacks on web-based systems, and viruses and worms are regularly distributed using Internet protocols.

All of this means that the verification and validation processes for web-based systems must focus on security assessment, where the ability of the system to resist different types of attack is tested. However, this type of security assessment is very difficult to carry out. Consequently, systems are often deployed with security loopholes that attackers use to gain access to or damage these systems.

Fundamentally, the reason why security is so difficult to assess is that security requirements, like some safety requirements, are 'shall not' requirements. That is, they specify what should not happen rather than system functionality or required behavior. It is not usually possible to define this unwanted behavior as simple constraints that may be checked by the system.

### Security checklist

1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorized users.
2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may allow unauthorized access through an unattended computer.
3. If the system is written in a programming language without array bound checking, are there situations where buffer overflow may be exploited?
4. Buffer overflow may allow attackers to send code strings to the system and then execute them
5. If passwords are set, does the system check that passwords are 'strong'? Strong passwords consist of mixed letters, numbers and punctuation, and are not normal dictionary entries. They are more difficult to break than simple passwords.

If resources are available, you can always demonstrate that a system meets its functional requirements. However, it is impossible to prove that a system does not do something, so, irrespective of the amount of testing, security vulnerabilities may remain in a system after it has been deployed. Even in systems that have been in use for many years, an ingenious attacker can discover a new form of attack-and penetrate what was thought to be a secure system. There are four complementary approaches to security checking:

1. ***Experience-based validation.*** In this case, the system is analyzed against types of attack that are known to the validation team. This type of validation is usually carried out in conjunction with tool-based validation. Checklists of known security problems as shown above may be created to assist with the process. This approach may use all system documentation and could be part of other system reviews that check for errors and omissions.
2. ***Tool-based validation.*** In this case, various security tools such as password checkers are used to analyze the system. Password checkers detect insecure passwords such as common names or strings of consecutive letters. This is really an extension of experience-based validation, where the experience is embodied in the tools used.

3. **Tiger teams.** In this case, a team is setup and given the objective of breaching the system security. They simulate attacks on the system and use their ingenuity to discover new ways to compromise the system security. This approach can be very effective, especially if team members have previous experience with breaking into systems.
4. **Formal verification.** A system can be verified against a formal security specification. However, as in other areas, formal verification for security is not widely used

It is very difficult for end-users of a system to verify its security. Consequently, bodies in North America and in Europe have established sets of security evaluation criteria that can be checked by specialized evaluators. Software product suppliers can submit their products for evaluation and certification against these criteria.

Therefore, if you have a requirement for a particular level of security, you can choose a product that has been validated to that level. However, many products are not security-certified or their certification applies only to individual products. *When the certified system is used in conjunction with other uncertified systems, such as locally developed software, the security level of the overall system cannot be assessed.*

## SAFETY AND DEPENDABILITY CASES

Safety cases and, more generically, dependability cases are structured documents setting out detailed arguments and evidence that a system is safe or that a required level of dependability has been achieved. For many types of critical systems, the production of a safety case is a legal requirement, and the case must satisfy some certification body before the system can be deployed.

Regulators are created by government to ensure that private industry does not profit by failing to following national standards for safety, security, and so on. There are regulators in many different industries. For example, airlines are regulated by national aviation authorities such as the FAA (in the US), the CAA (in the UK), and the CAAN (in Nepal). Railway regulators exist to ensure the safety of railways, and nuclear regulators must certify the safety of a nuclear plant before it can go online. In the banking sector, national banks serve as regulators, establishing procedures and practices to reduce the probability of fraud and to protect banking customers from risky banking practices, for example, the Nepal Rastra Bank (in Nepal). As software systems have become increasingly important in the critical infrastructure of countries, these regulators have become more and more concerned with safety and dependability cases for software systems.

The role of a regulator is to check that a completed system is as safe as practicable, so they are mainly involved when a development project is complete. However, regulators and developers rarely work in isolation; they communicate with the development team to establish what has to be included in the safety case. The regulator and developers jointly examine processes and procedures to make sure that these are being enacted and documented to the regulator's satisfaction.

Of course, software itself is not dangerous. It is only when it is embedded in a large, computer-based or socio-technical system that software failures can result in failures of other equipment or processes that can cause injury or death. Therefore, a software safety case is always part of a wider system safety case that demonstrates the safety of the overall system. When constructing a software safety case, you have to

relate software failures to wider system failures and demonstrate that either these software failures will not occur or that they will not be propagated in such a way that dangerous system failures may occur.

| Component                   | Description   |
|-----------------------------|---|
| System description          | An overview of the system and a description of its critical components  |
| Safety requirements         | The safety requirements abstracted from the system requirements specification                                       |
| Hazard and risk analysis    | Documents describing the hazards and risks that have been identified and the measures taken to reduce risk          |
| Design analysis             | A set of structured arguments that justify why the design is safe   |
| Verification and validation | A description of the V&V procedures used and, where appropriate, the test plans for the system                      |
| Review reports              | Records of all design and safety reviews  |
| Team competences            | Evidence of the competence of all of the team involved in safety-related systems development and validation         |
| Process QA                  | Records of the quality assurance processes carried out during system development                                    |
| Change management processes | Records of all changes proposed, actions taken and, where appropriate, justification of the safety of these changes |
| Associated safety cases     | References to other safety cases that may impact on this safety case  |

Table 7.4: Components of a software safety case [1]

A safety case is a set of documents that include a description of the system, that has to be certified, information about the processes used to develop the system and, critically, logical arguments that demonstrate that the system is likely to be safe. More succinctly, **safety case** can be defined as:

*A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment*

The organization and contents of a safety case depends on the type of system that is to be certified and its context of operation. Table 7.4 shows one possible organization for a software safety case.

A key component of a safety case is a set of logical arguments for system safety. These may be absolute arguments (event X will or will not happen) or probabilistic arguments (the probability of event X is G.Y); when combined, these should demonstrate safety. As shown in Figure 7.7, an argument is a relationship between what is thought to be the case (a claim) and a body of evidence that has been collected. The argument essentially explains why the claim (which is generally that something is safe) can be inferred from the available evidence. Naturally, given the multilevel nature of systems claims are organized in a hierarchy.

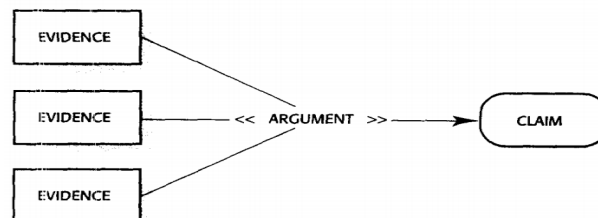


Figure 7.7: The structure of an argument [1]

## REFERENCES

[1] I. Sommerville , 2008. Software Engineering. Eighth Edition, Addison-Wesley



## 8. SOFTWARE TESTING AND COST ESTIMATION

A general testing process starts with the testing of individual program units such as functions or objects (i.e., *unit testing*). These are then integrated into sub-systems and systems, and the interactions of these units are tested (i.e., *integration testing*). Finally, after delivery of the system, the customer may carry out a series of acceptance tests to check that the system performs as specified (i.e., *acceptance testing*). This model of the testing process is appropriate for large system development but for smaller systems, or for systems that are developed through scripting or reuse, there are often fewer distinct stages in the process. A more abstract view of software testing is shown in Figure 8.1. The two fundamental testing activities are *component testing*-testing the parts of the system-and *system testing*-testing the system as a whole.

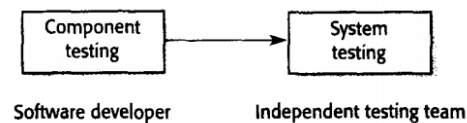


Figure 8.1: Testing phases [1]

The aim of the component testing stage is to discover defects by testing individual program components. These components may be functions, objects or reusable components. During system testing, these components are integrated to form sub-systems or the complete system. At this stage, system testing should focus on establishing that the system meets its functional and non-functional requirements, and does not behave in unexpected ways. Inevitably, defects in components that have been missed during earlier testing are discovered during system testing.

The software testing process has two distinct goals:

1. ***To demonstrate to the developer and the customer that the software meets its requirements.*** For custom software, this means that there should be at least one test for every requirement in the user and system requirements documents. For generic software products, it means that there should be tests for all of the system features that will be incorporated in the product release. Some systems may have an explicit acceptance testing phase where the customer formally checks that the delivered system conforms to its specification.
2. ***To discover faults or defects in the software where the behavior of the software is incorrect, undesirable or does not conform to its specification.*** Defect testing is concerned with rooting out all kinds of undesirable system behavior, such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

The first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect the systems expected use. The second goal leads to defect testing, where the test cases are designed to expose defects. The test cases can be deliberately obscure and need not reflect how the system is normally used. For validation testing, a successful test is one where the system performs correctly. For defect testing, a successful test is one that exposes a defect that causes the system to perform incorrectly.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test that you have overlooked could discover further problems with the system. Testing can only show the presence of errors, not their absence.

Overall, therefore, the goal of software testing is to convince system developers and customers that the software is good enough for operational use. Testing is a process intended to build confidence in the software.

A general model of the testing process is shown in Figure 8.2. Test cases are specifications of the inputs to the test and the expected output from the system plus a statement of what is being tested. Test data are the inputs that have been devised to test the system. Test data can sometimes be generated automatically. Automatic test case generation is impossible. The output of the tests can only be predicted by people who understand what the system should do.

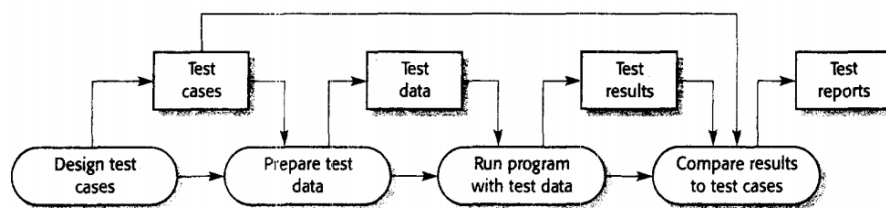


Figure 8.2: A model of the software testing process [1]

Exhaustive testing, where every possible program execution sequence is tested, is impossible. Testing, therefore, has to be based on a subset of possible test cases. Ideally, software companies should have policies for choosing this subset rather than leave this to the development team. These policies might be based on general testing policies, such as a policy that all program statements should be executed at least once. Alternatively, the testing policies may be based on experience of system usage and may focus on testing the features of the operational system. For example:

1. All system functions that are accessed through menus should be tested.
2. Combinations of functions (e.g., text formatting) that are accessed through the same menu must be tested.
3. Where user input is provided, all functions must be tested with both correct and incorrect input.

It is clear from experience with major software products such as word processors or spread sheets that comparable guidelines are normally used during product testing. When features of the software are used in isolation, they normally work. Problems arise, when combinations of features have not been tested together.

As part of the V&V planning process, managers have to make decisions on who should be responsible for the different stages of testing. For most systems, programmers take responsibility for testing the components that they have developed. Once this is completed, the work is handed over to an integration team, which integrates the modules from different developers, builds the software and tests the system as a whole. For critical systems, a more formal process may be used where independent testers are responsible for all stages of the testing process. In critical system testing, the tests are developed separately and detailed records are maintained of the test results.

Component testing by developers is usually based on an intuitive understanding of how the components should operate. System testing, however, has to be based on a written system specification. This can be a detailed system requirements specification, or it can be a higher-level user-oriented specification of the features that should be implemented in the system. A separate team is normally responsible for system testing. The system testing team works from the user and system requirements documents to develop system-testing plans.

## 8.1. System Testing

System testing involves integrating two or more components that implement system functions or features and then testing this integrated system. In an iterative development process, system testing is concerned with testing an increment to be delivered to the customer; in a waterfall process, system testing is concerned with testing the entire system. For most complex systems, there are two distinct phases to system testing:

1. **Integration testing**, where the test team has access to the source code of the system. When a problem is discovered, the integration team tries to find the source of the problem and identify the components that have to be debugged. Integration testing is mostly concerned with finding defects in the system.
2. **Release testing**, where a version of the system that could be released to users is tested. Here, the test team is concerned with validating that the system meets its requirements and with ensuring that the system is dependable. Release testing is usually '**black-box**' testing where the test team is simply concerned with demonstrating that the system does or does not work properly. Problems are reported to the development team whose job is to debug the program. Where customers are involved in release testing, this is sometimes called **acceptance testing**. If the release is good enough, the customer may then accept it for use.

Fundamentally, you can think of integration testing as the testing of incomplete systems composed of clusters or groupings of system components. Release testing is concerned with testing the system release that is intended for delivery to customers. Naturally, these overlap, especially when incremental development is used and the system to be released is incomplete. Generally, the priority in integration testing is '0 discover defects in the system and the priority in system testing, is to validate that the system meets its requirements. However, in practice, there is some validation testing and some defect testing during both of these processes.

## INTEGRATION TESTING

The process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions. The components that are integrated may be off-the-shelf components, reusable components that have been adapted for a particular system or newly developed components. For many large systems, all three types of components are likely to be used. Integration testing checks that these components actually work together are called correctly and transfer the right data at the right time across their interfaces.

System integration involves identifying clusters of components that deliver some system functionality and integrating these by adding code that makes them work together. Sometimes, the overall skeleton of the system is developed first, and components are added to it. This is called **top-down integration**.

Alternatively, you may first integrate infrastructure components that provide common services, such as network and database access, and then add the functional components. This is **bottom-up integration**. In practice for many systems, the integration strategy is a mixture of these, with both infrastructure components and functional components added in increments. In both top-down and bottom-up integration, you usually have to develop additional code to simulate other components and allow the system to execute.

*A major problem that arises during integration testing is localizing errors.* There are complex interactions between the system components and, when an anomalous output is discovered, you may find it hard to identify where the error occurred. To make it easier to locate errors, you should always use an incremental approach to system integration and testing. Initially, you should integrate a **minimal system configuration** and test this system. You then add components to this minimal configuration and test after each **added increment**.

In the example shown in Figure 8.3, A, B, C and D are components and T1 to T5 are related sets of tests of the features incorporated in the system. T1, T2 and T3 are first run on a system composed of component A and component B (the minimal system). If these reveal defects, they are corrected. Component C is integrated and T1, T2 and T3 are repeated to ensure that there have not been unexpected interactions with A and B. If problems arise in these tests, this probably means that they are due to interactions with the new component. The source of the problem is localized, thus simplifying defect location and repair. Test set T4 is also run on the system. Finally, component D is integrated and tested using existing and new tests (T5).

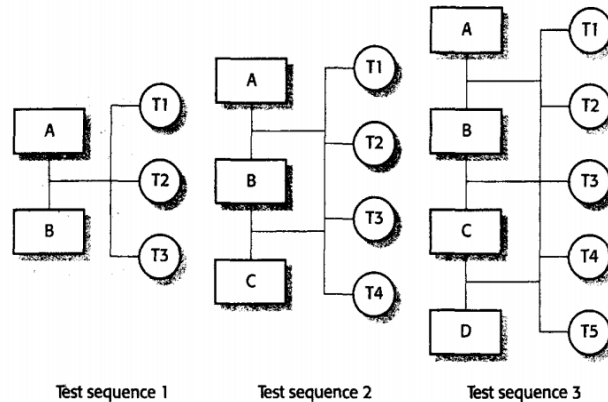


Figure 8.3: Incremental integration testing [1]

When planning integration, you have to decide the order of integration of components. In a process such as XP, the customer is involved in the development process and decides which functionality should be included in each system increment. Therefore, system integration is driven by customer priorities. In other approaches to development when off-the-shelf components and specially developed components are integrated, the customer may not be involved and the integration team decides on the integration priorities.

In such cases, a good rule of thumb is to integrate the components that implement the most frequently used functionality first. This means that the components that are most used receive the most testing. For

example, in the library system, LIB SYS, you should start by integrating the search facility so that, in a minimal system, users can search for documents that they need. You should then add the functionality to allow users to download a document, and then progressively add the components that implement other system features.

Of course, reality is rarely as simple as this model suggests. The implementation of system features may be spread across a number of components. To test a new feature, you may have to integrate several different components. The testing may reveal errors in the interactions between these individual components and other parts of the system. Repairing errors may be difficult because a group of components that implement the system feature may have to be changed. Furthermore, integrating and testing a new component can change the pattern of already tested component interactions. Errors may be revealed that were not exposed in the tests of the simpler configuration.

These problems mean that when a new increment is integrated, it is important to rerun the tests for previous increments as well as the new tests that are required to verify the new system functionality. Rerunning an existing set of tests is called **regression testing**. If regression testing exposes problems, then you have to check whether these are problems in the previous increment that the new increment has exposed or whether these are due to the added increment of functionality

Regression testing is clearly an expensive process and is impractical without some automated support. In extreme programming, all tests are written as executable code where the test input and the expected outputs are specified and automatically checked. When used with an automated testing framework such as JUnit, this means that tests can be automatically rerun. It is a basic principle of extreme programming that the complete test set is executed whenever new code is integrated and that this new code is not accepted until all tests run successfully.

## RELEASE TESTING

Release testing is the process of testing a release of the system that will be distributed to customers. The primary goal of this process is to increase the supplier's confidence that the system meets its requirements. If so, it can be released as a product or delivered to the customer. To *demonstrate that the system meets its requirements, you have to show that it delivers the specified functionality, performance and dependability, and that it does not fail during normal use.*

Release testing is usually a black-box testing process where the tests are derived from the system specification. The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs.

Another name for this is functional testing because the tester is only concerned with the functionality and not the implementation of the software. Figure 8.4 illustrates the model of a system that is assumed in black-box testing. The tester presents inputs to the component or the system and examines the corresponding outputs. If the outputs are not those predicted (i.e., if the outputs are in set  $O_e$ ) then the test has detected a problem with the software.

When testing system releases, you should try to 'break' the software by choosing test cases that are in the set  $I_e$  in Figure 8.4. That is your aim should be to select inputs that have a high probability of generating

system failures (outputs in set  $O_e$ )' You use previous experience of what are likely to be successful defect tests and testing guidelines to help you make your choice.

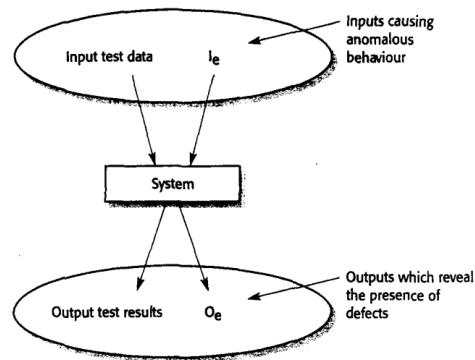


Figure 8.4: Black box testing [1]

Some examples of guidelines that increase the probability that the defect tests will be successful are:

1. Choose inputs that force the system to generate all error messages.
2. Design inputs that cause input buffers to overflow.
3. Repeat the same input or series of inputs numerous times.
4. Force invalid outputs to be generated.
5. Force computation results to be too large or too small.

To validate that the system meets its requirements, the best approach to use is scenario-based testing, where you devise a number of scenarios and develop test cases from these scenarios. For example, the following scenario might describe how the library system LIB SYS, might be used:

*A student in Scotland studying American history has been asked to write a paper on 'Frontier mentality in the American West from 1840 to 1880'. To do this, she needs to find sources from a range of libraries. She logs onto the LIB SYS system and uses the search facility to discover whether she can access original documents from that time. She discovers sources in various US university libraries and downloads copies of some of these. However, for one document, she needs to have confirmation from her university that she is a genuine student and that use is for non-commercial purposes. The student then uses the facility in LIB SYS that can request such permission and registers her request. If granted, the document will be downloaded to the registered library's server and printed for her. She receives a message from LIB SYS telling her that she will receive an e-mail message when the printed document is available for collection.*

From this scenario, it is possible to devise a number of tests that can be applied to the proposed release of LIB SYS:

1. Test the login mechanism using correct and incorrect logins to check that valid users are accepted and invalid users are rejected.
2. Test the search facility using queries against known sources to check that the search mechanism is actually finding documents.
3. Test the system presentation facility to check that information about documents is displayed properly.

4. Test the mechanism to request permission for downloading.
5. Test the e-mail response indicating that the downloaded document is available.

For each of these tests, you should design a set of tests that include valid and invalid inputs and that generate valid and invalid outputs. You should also organize scenario-based testing so that the most likely scenarios are tested first, and unusual or exceptional scenarios considered later, so your efforts are devoted to those parts of the system that receive the most use.

## PERFORMANCE TESTING

Once a system has been completely integrated, it is possible to test the system for emergent properties such as performance and reliability. Performance tests have to be designed to ensure that the system can process its intended load. This usually involves planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system. To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile is a set of tests that reflect the actual mix of work that will be handled by the system. Therefore, if 90% of the transactions in a system are of type A, 5% of type B and the remainder of types C, D, and E, then you have to design the operational profile so that the vast majority of tests are of type A. Otherwise, you will not get an accurate test of the operational performance of the system.

This approach, of course, is not necessarily the best approach for defect testing. An effective way to discover defects is to design tests around the limits of the system. In performance testing, this means stressing the system (hence the name *stress- testing*) by making demands that are outside the design limits of the software.

For example, a transaction processing system may be designed to process up to 300 transactions per second; an operating system may be designed to handle up to 1,000 separate terminals. Stress testing continues these tests beyond the maximum design load of the system until the system fails. This type of testing has two functions:

1. It tests the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, it is important that system failure should not be caused at a corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to 'fail-soft' rather than collapse under its load.
2. It stresses the system and may cause defects to come to light that would not normally be discovered. Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unusual combinations of normal circumstances that the stress testing replicates.

Stress testing is particularly relevant to distributed systems based on a network of processors. These systems often exhibit severe degradation when they are heavily loaded. The network becomes swamped with coordination data that the different processes must exchange, so the processes become slower and slower as they wait for the required data from other processes.

## 8.2. Component Testing

Component testing (sometimes called *unit testing*) is the process of testing individual components in the system. This is a defect testing process so its goal is to expose faults in these components. For most systems, the developers of components are responsible for component testing.

There are different types of component that may be tested at this stage:

1. Individual functions or methods within an object
2. Object classes that have several attributes and methods
3. Composite components made up of several different objects or functions. These composite components have a defined interface that is used to access their functionality.

Individual functions or methods are the simplest type of component and your tests are a set of calls to these routines with different input parameters. You can use the approaches to test case design, to design the function or method tests. When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. Therefore, object class testing should include:

1. The testing in isolation of all operations associated with the object
2. The setting and interrogation of all attributes associated with the object
3. The exercise of the object in all possible states. This means that all events that cause a state change in the object should be simulated

Consider, for example, the weather station whose interface is shown in Figure 8.5. It has only a single attribute, which is its identifier. This is a constant that is set when the weather station is installed. You therefore only need a test that checks whether it has been set up. You need to define test cases for *reportWeather*, *calibrate*, *test*, *startup* and *shutdown*. Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary. For example, to test *shutdown* you need to have executed the *startup* method.

| WeatherStation  |
|---|
| identifier  |
| reportWeather ()<br>calibrate (instruments)<br>test ()<br>startup (instruments)<br>shutdown (instruments) |

Figure 8.5: Interface for weather station [1]

To test the states of the weather station, you use a state model. Using this model, you can identify sequences of state transitions that have to be tested and define event sequences to force these transitions. In principle, you should test every possible state transition sequence, although in practice this may be too expensive. Examples of state sequences that should be tested in the weather station include:

Shutdown → Waiting → Shutdown  
Waiting → Calibrating → Testing → Transmitting → Waiting  
Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting



If you use inheritance, this makes it more difficult to design object class tests. Where a super class provides operations that are inherited by a number of subclasses, all of these subclasses should be tested with all inherited operations. The reason for this is that the inherited operation may make assumptions about other operations and attributes, which these may have been changed when inherited. Equally, when a super class operation is overridden then the overwriting operation must be tested.

Tests that fall into the same equivalence class might be those that use the same attributes of the objects. Therefore, equivalence classes should be identified that initialize, access and update all object class attributes.

## INTERFACE TESTING

Many components in a system are not simple functions or objects but are composite components that are made up of several interacting objects. *The functionality of these components can be accessed through their defined interface.* Testing these composite components then is primarily concerned with testing that the component interface behaves according to its specification.

Figure 8.6 illustrates this process of interface testing. Assume that components A, B and C have been integrated to create a larger component or sub-system. The test cases are not applied to the individual components but to the interface of the composite component created by combining these components.

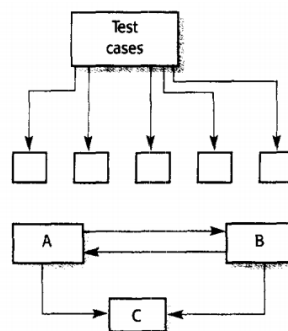


Figure 8.6: Interface testing [1]

*Interface testing is particularly important for object-oriented and component-based development.* Objects and components are defined by their interfaces and may be reused in combination with other components in different systems. Interface errors in the composite component cannot be detected by testing the individual objects or components. Errors in the composite component may arise because of interactions between its parts.

There are different types of interfaces between program components and, consequently, different types of interface errors that can occur:

1. **Parameter interfaces.** These are interfaces where data or sometimes function references are passed from one component to another.
2. **Shared memory interfaces.** These are interfaces where a block of memory is shared between components. Data is placed in the memory by one sub-system and retrieved from there by other sub-systems.

3. **Procedural interfaces.** These are interfaces where one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.
4. **Message passing interfaces.** These are interfaces where one component requests a service from another component by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client-server systems.

Interface errors are one of the most common forms of error in complex systems. These errors fall into three classes:

1. **Interface misuse.** A calling component calls some other component and makes an error in the use of its interface. This type of error is particularly common with parameter interfaces where parameters may be of the wrong type, maybe passed in the wrong order or the wrong number of parameters may be passed.
2. **Interface misunderstanding.** A calling component misunderstands the specification of the interface of the called component and makes assumptions about the behavior of the called component. The called component does not behave as expected and this causes unexpected behavior in the calling component. For example, a binary search routine may be called with an unordered array to be searched. The search would then fail.
3. **Timing errors.** These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

Testing for interface defects is difficult because some interface faults may only manifest themselves under unusual conditions. For example, say an object implements a queue as a fixed-length data structure. A calling object may assume that the queue is implemented as an infinite data structure and may not check for queue overflow when an item is entered. This condition can only be detected during testing by designing test cases that force the queue to overflow and cause that overflow to corrupt the object behavior in some detectable way.

A further problem may arise because of interactions between faults in different modules or objects. Faults in one object may only be detected when some other object behaves in an unexpected way. For example, an object may call some other object to receive some service and may assume that the response is correct. If there has been a misunderstanding about the value computed, the returned value may be valid but incorrect. This will only manifest itself when some later computation goes wrong.

Some general guidelines for interface testing are:

1. Examine the code to be tested and explicitly list each call to an external component. Design a set of tests where the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.
2. Where pointers are passed across an interface, always test the interface with null pointer parameters.

3. Where a component is called through a procedural interface, design tests that should cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.
4. Use stress testing, in message-passing system. Design tests that generate many more messages than are likely to occur in practice. Timing problems may be revealed in this way.
5. Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

Static validation techniques are often more cost-effective than testing for discovering interface errors. A strongly typed language such as Java allows many interface errors to be trapped by the compiler. Where a weaker language, such as C, is used, a static analyzer such as LINT can detect interface errors. Program inspections can concentrate on component interfaces and questions about the assumed interface behavior asked during the inspection process.

### 8.3. Test Case Design

Test case design is a part of system and component testing where you design the test cases (inputs and predicted outputs) that test the system. The goal of the test case design process is to create a set of test cases that are effective in discovering program defects and showing that the system meets its requirements.

*To design a test case, you select a feature of the system or component that you are testing.* You then select a set of inputs that execute that feature, document the expected outputs or output ranges and, where possible, design an automated check that tests that the actual and expected outputs are the same.

There are various approaches that you can take to test case design:

1. **Requirements-based testing** where test cases are designed to test the system requirements. This is mostly used at the system-testing stage as system requirements are usually implemented by several components. For each requirement, you identify test cases that can demonstrate that the system meets that requirement.
2. **Partition testing** where you identify input and output partitions and design tests so that the system executes inputs from all partitions and generates outputs in all partitions. Partitions are groups of data that have common characteristics such as all negative numbers, all names less than 30 characters, all events arising from choosing items on a menu, and so on.
3. **Structural testing** where you use knowledge of the program's structure to design tests that exercise all parts of the program. Essentially, when testing a program, you should try to execute each statement at least once. Structural testing helps identify test cases that can make this possible.

In general, when designing test cases, you should start with the highest-level tests from the requirements then progressively add more detailed tests using partition and structural testing.

## REQUIREMENTS-BASED TESTING

A general principle of requirements engineering, *is that requirements should be testable*. That is, the requirement should be written in such a way that a test can be designed so that an observer can check that the requirement has been satisfied. Requirements-based testing, therefore, is a systematic approach to test case design where you consider each requirement and derive a set of tests for it. Requirements-based testing is validation rather than defect testing-you are trying to demonstrate that the system has properly implemented its requirements.

For example, consider the requirements for the LIB SYS system introduced earlier:

1. The user shall be able to search either all of the initial set of databases or select a subset from it.
2. The system shall provide appropriate viewers for the user to read documents in the document store.
3. Every order shall be allocated a unique identifier (ORDER\_ID) that the user shall be able to copy to the account's permanent storage area.

Possible tests for the first of these requirements, assuming that a search function has been tested, are:

- Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes one database.
- Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes two databases.
- Initiate user searches for items that are known to be present and known not to be present where the set of databases includes more than two databases.
- Select one database from the set of databases and initiate user searches for items that are known to be present and known not to be present.
- Select more than one database from the set of databases and initiate searches for items that are known to be present and known not to be present.

You can see from this that testing a requirement does not mean just writing a single test. You normally have to write several tests to ensure that you have coverage of the requirement.

Tests for the other requirements in the LIB SYS system can be developed in the same way. For the second requirement, you would write tests that delivered documents of all types that could be processed by the system and check that these are properly displayed. The third requirement is simpler. To test this, you simulate placing several orders and then check that the order identifier is present in the user confirmation and is unique in each case.

## PARTITION TESTING

The input data and output results of a program usually fall into a number of different classes that have common characteristics such as positive numbers, negative numbers and menu selections. Programs normally behave in a comparable way for all members of a class. That is, if you test a program that does some computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers.

Because of this equivalent behavior, these classes are sometimes called *equivalence partitions or domains*. One systematic approach to test case design is based on identifying all partitions for a system or component. Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.

In Figure 8.7, each equivalence partition is shown as an ellipse. Input equivalence partitions are sets of data where all of the set members should be processed in an equivalent way. Output equivalence partitions are program outputs that have common characteristics, so they can be considered as a distinct class. You also identify partitions where the inputs are outside the other partitions that you have chosen. These test whether the program handles invalid input correctly. Valid and invalid inputs also form equivalence partitions.

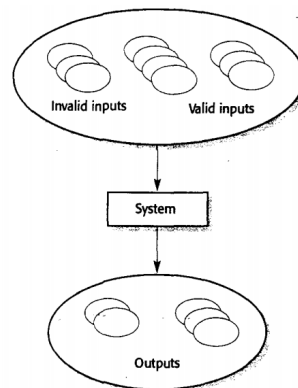


Figure 8.7: Equivalence partitioning [1]

Once you have identified a set of partitions, you can choose test cases from each of these partitions. A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions plus cases close to the mid-point of the partition. The rationale for this is that designers and programmers tend to consider typical values of inputs when developing a system. You test these by choosing the mid-point of the partition. Boundary values are often a typical (e.g., zero may behave differently from other non negative numbers) so are overlooked by developers. Program failures often occur when processing these typical values.

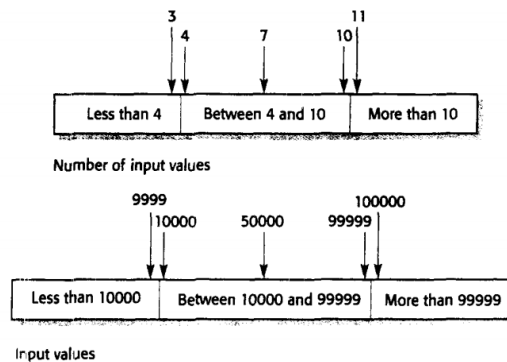


Figure 8.8: Equivalence partitions [1]

You identify partitions by using the program specification or user documentation and, from experience, where you predict the classes of input value that are likely to detect errors. *For example, say a program specification states that the program accepts 4 to 8 inputs that are five-digit integers greater than 10,000.* Figure 8.8 shows the partitions for this situation and possible test input values.

To illustrate the derivation of test cases, let us use the specification of a search component, as shown in Figure 8.9. This component searches a sequence of elements for a given element (the key). It returns the position of that element in the sequence. The specification of a search component here is specified in an abstract way by defining **pre-conditions**, which are true before the component is called, and **post-conditions**, which are true after execution.

```
procedure Search (Key : ELEM ; T: SEQ of ELEM ;
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
  – the sequence has at least one element
  T'FIRST <= T'LAST
Post-condition
  – the element is found and is referenced by L
  ( Found and T (L) = Key)
or
  – the element is not in the sequence
  ( not Found and
not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

Figure 8.9: The specification of a search routine [1]

The precondition states that the search routine will only work with sequences that are not empty. The post-condition states that the variable *Found* is set if the key element is in the sequence. The position of the key element is the index L. The index value is undefined if the element is not in the sequence.

From this specification, you can see two equivalence partitions:

1. Inputs where the key element is a member of the sequence (Found =true)
2. Inputs where the key element is not a sequence member (Found =false)

When you are testing programs with sequences, arrays or lists, there are a number of guidelines that are often useful in designing test cases:

1. Test software with sequences that have only a single value. Programmers naturally think of sequences as made up of several values, and sometimes they embed this assumption in their programs. Consequently, the program may not work properly when presented with a single-value sequence.
2. Use different sequences of different sizes in different tests. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.
3. Derive tests so that the first, middle and last elements of the sequence are accessed. This approach reveals problems at partition boundaries.

From these guidelines, two more equivalence partitions can be identified:

1. The input sequence has a single value.
2. The number of elements in the input sequence is greater than 1.

You then identify further partitions by combining these partitions—for example, the partition where the number of elements in the sequence is greater than 1 and the element is not in the sequence. Figure 8.10 shows the partitions that have been identified to test the search component.

A set of possible test cases based on these partitions is also shown in Figure 8.10. If the key element is not in the sequence, the value of L is undefined ('n').

| Sequence          | Element                    |
|-------------------|----------------------------|
| Single value      | In sequence                |
| Single value      | Not in sequence            |
| More than 1 value | First element in sequence  |
| More than 1 value | Last element in sequence   |
| More than 1 value | Middle element in sequence |
| More than 1 value | Not in sequence            |

| Input sequence (T)         | Key (Key) | Output (Found, L) |
|----------------------------|-----------|-------------------|
| 17                         | 17        | true, 1           |
| 17                         | 0         | false, ??         |
| 17, 29, 21, 23             | 17        | true, 1           |
| 41, 18, 9, 31, 30, 16, 45  | 45        | true, 7           |
| 17, 18, 21, 23, 29, 41, 38 | 23        | true, 4           |
| 21, 23, 29, 33, 38         | 25        | false, ??         |

Figure 8.10: Equivalence Partitions for Search Routine [1]

The guideline that different sequences of different sizes should be used has been applied in these test cases.

The set of input values used to test the search routine is not exhaustive. The routine may fail if the input sequence happens to include the elements 1, 2, 3 and 4.

However, it is reasonable to assume that if the test fails to detect defects when one member of a class is processed, no other members of that class will identify defects. Of course defects may still exist. Some equivalence partitions may not have been identified, errors may have been made in equivalence partition identification or the test data may have been incorrectly prepared..

## STRUCTURAL TESTING

Structural testing is an approach to test case design where the tests are derived from knowledge of the software's structure and implementation. This approach is sometimes called '*white-box*', '*glass-box*' testing, or '*clear-box*' testing to distinguish it from black-box testing.

Understanding the algorithm used in a component can help you identify further partitions and test cases. To illustrate, let us take an example of a binary search routine. The sequence is implemented as an array

that array must be ordered and the value of the lower bound of the array must be less than the value of the upper bound.

As shown in Figure 8.11, the binary searching involves splitting the search engine into three parts. Each of these parts makes up an equivalence partition. You then design test cases where the key lies at the boundaries of each these partitions (See Figure 8.12, the test case for search routine).

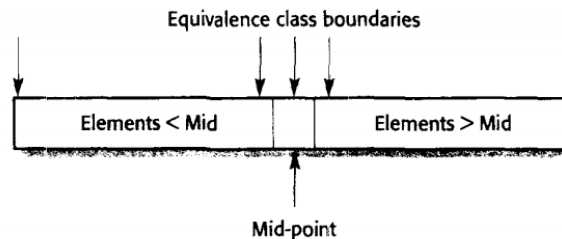


Figure 8.11: Binary search implementation [1]

| Input array (T)            | Key (Key) | Output (Found, L) |
|----------------------------|-----------|-------------------|
| 17                         | 17        | true, 1           |
| 17                         | 0         | false, ??         |
| 17, 21, 23, 29             | 17        | true, 1           |
| 9, 16, 18, 30, 31, 41, 45  | 45        | true, 7           |
| 17, 18, 21, 23, 29, 38, 41 | 23        | true, 4           |
| 17, 18, 21, 23, 29, 33, 38 | 21        | true, 3           |
| 12, 18, 21, 23, 32         | 23        | true, 4           |
| 21, 23, 29, 33, 38         | 25        | false, ??         |

Figure 8.12: Test case for search routine [1]

## PATH TESTING

Path testing is a structural testing strategy whose objective is to exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once. Furthermore, all conditional statements are tested for both true and false cases. In an object-oriented development process, path testing may be used when testing methods associated with objects.

The number of paths through a program is usually proportional to its size. As modules are integrated into systems, it becomes unfeasible to use structural testing techniques. Path testing techniques are therefore mostly used during component testing.

Path testing does not test all possible combinations of all paths through the program. For any components apart from very trivial ones without loops, this is an impossible objective. There are an infinite number of possible path combinations in programs with loops. Even when all program statements have been executed at least once, program defects may still show up when particular paths are combined.

The starting point for path testing is a program flow graph. This is a skeletal model of all paths through the program. A flow graph consists of nodes representing decisions and edges showing flow of control.



The flow graph is constructed by replacing program control statements by equivalent diagrams. If there are no *goto* statements in a program, it is a simple process to derive its flow graph. Each branch in a conditional statement (*if-then-else* or *case*) is shown as a separate path. An arrow looping back to the condition node denotes a loop. Figure 8.13 shows the flow graph for the binary search method.

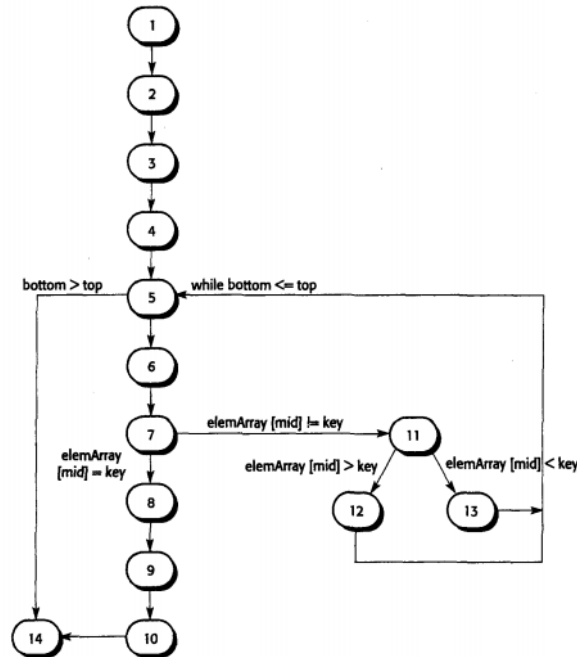


Figure 8.13: Flow graph for a binary search routine [1]

The objective of path testing is to ensure that each independent path through the program LS executed at least once. An independent program path is one that traverses at least one new edge in the flow graph. In program terms, this means exercising one or more new conditions. Both the true and false branches of all condition must be executed.

The flow graph for the binary search procedure is shown in Figure 8.13 where each node represents a line in the program with an executable statement. By tracing the flow, therefore, you can see that the paths through the binary search flow graph are:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14

1, 2, 3, 4, 5, 14

1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ..

1, 2, 3, 4, 6, 7, 2, 11, 13, 5, ..

If all of these paths are executed, we can be sure that every statement in the method has been executed at least once and that every branch has been exercised for true and false conditions.

You can find the number of independent paths in a program by computing the *cyclomatic complexity* of the program flow graph. For programs without *goto* statements, the value of the cyclomatic complexity is

one more than the number of conditions in the program. A simple condition is logical expression without 'and' or 'or' connectors. If the program includes compound conditions, which are logical expressions including 'and' or 'or' connectors, then you count the number of simple conditions in the compound conditions when calculating the cyclomatic complexity.

Therefore, if there are six *if-statements* and a *while loop* and all conditional expressions are simple, the cyclomatic complexity is 8. If one conditional expression is a compound expression such as '*if A and B or C*', then you count this as three simple conditions. The cyclomatic complexity is therefore 10.

After discovering the number of independent paths through the code by computing the cyclomatic complexity, you next design test cases to execute each of these paths. The minimum number of test cases that you need to test all program paths is equal to the cyclomatic complexity.

Test case design is straight forward in the case of the binary search routine. However, when programs have a complex branching structure, it may be difficult to predict how any particular test case will be processed. In these cases, a dynamic program analyzer can be used to discover the programs execution profile.

Dynamic program analyzers are testing tools that work in conjunction with compilers. During compilation, these analyzers add extra instructions to the generated code. These count the number of times each program statement has been executed.

After the program has been run, an execution profile can be printed. This shows which parts of the program have and have not been executed using particular test cases. This execution profile therefore reveals untested program sections.

#### 8.4. Test Automation

Testing is an expensive and laborious phase of the software process. As a result, testing tools were among the first software tools to be developed. These tools now offer a range of facilities and their use can significantly reduce the costs of testing. For example, JUnit is a set of Java classes that the user extends to create an automated testing environment. Each individual test is implemented as an object and a test runner runs all of the tests. The tests themselves should be written in such a way that they indicate whether the tested system has behaved as expected.

A software testing workbench is an integrated set of tools to support the testing process. In addition to testing frameworks that support automated test execution, a work bench may include tools to simulate other parts of the system and to generate system test data. Figure 8.14 shows some of the tools that might be included in such a testing workbench:

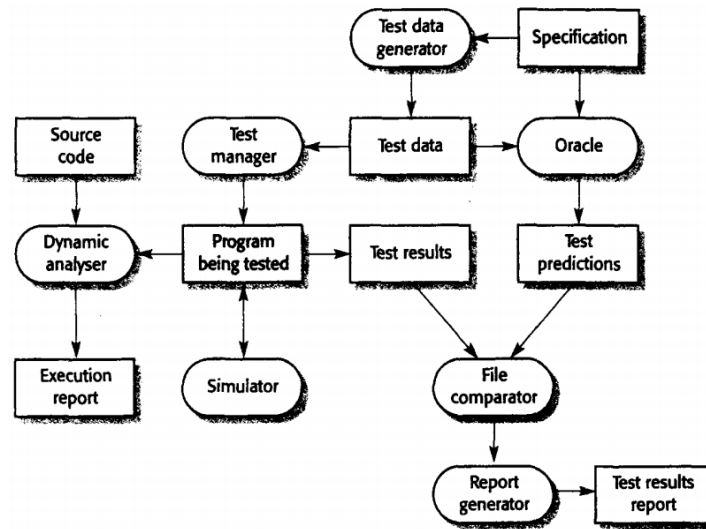


Figure 8.14: Tools for testing workbench [1]

1. **Test manager** manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested. Test automation frameworks such as JUnit are examples of test managers.
2. **Test data generator** generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.
3. **Oracle** generates predictions of expected test results. Oracles may either be previous program versions or prototype systems. Back-to-back testing involves running the oracle and the program to be tested in parallel. Differences in their outputs are highlighted.
4. **File comparator** compares the results of program tests with previous test results and reports differences between them. Comparators are used in regression testing where the results of executing different versions are compared. Where automated tests are used, this may be called from within the tests themselves.
5. **Report generator** provides report definition and generation facilities for test results.
6. **Dynamic analyzer** adds code to a program to count the number of times each statement has been executed. After testing, an execution profile is generated showing how often each program statement has been executed.
7. **Simulator** different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute. User interface simulate on are script-driven programs that simulate multiple simultaneous user interactions. Using simulators for I/O means that the timing of transaction sequences is repeatable.

When used for large system testing, tools have to be configured and adapted for the specific system that is being tested. For example:

- New tools may have to be added to test specific application characteristics, and some existing testing tools may not be required.

- Scripts may have to be written for user interface simulators and patterns defined for test data generators. Report formats may also have to be defined.
- Sets of expected test results may have to be prepared manually if no previous program versions are available to serve as an oracle.
- Special-purpose file comparators may have to be written that include knowledge of the structure of the test results on file.

A significant amount of effort and time is usually needed to create a comprehensive testing workbench. Complete test workbenches, as shown in Figure 8.14, are therefore only used when large systems are being developed. For these systems, the overall testing costs may be up to 50% of the total development costs so it is cost-effective to invest on high-quality CASE tool support for testing. However, because different types of systems require different types of testing support, off the-shelf testing tools may not be available.

### 8.5. Metrics for Testing

In software testing, **Metric** is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute. In other words, metrics help estimating the progress, quality and health of a software testing effort. Test metrics are the means by which the software quality can be measured. They provide the visibility into the readiness of the product, and give clear measurement of the quality and completeness of the product. The ideal example to understand metrics would be a weekly mileage of a car compared to its ideal mileage recommended by the manufacturer.

Software test metrics improve the efficiency and effectiveness of a software testing process. Software testing metrics or software test measurement is the quantitative indication of extent, capacity, dimension, amount or size of some attribute of a process or product. Example for software test measurement is the total number of defects. "*We cannot improve and control what we cannot measure*" and **Test Metrics** helps us to do exactly the same. Without measurement it is impossible to tell whether the process implemented is improving or not. Some of the needs why we use test metrics are mentioned below:

- Take decision for next phase of activities
- Evidence of the claim or prediction
- Understand the type of improvement required
- Take decision on process or technology change

There will be certain questioning during and after testing, such as:

- How long would it take to test?
- How bad/good is the product?
- How many bugs still remain in the product?
- Will testing be completed on time?
- Was the testing done effectively?
- How much effort went into testing the product?

To answer these questions properly, we need some type of measurements and record keeping to justify answers. This is where the test metrics come into picture.

The steps of test metric are mentioned in Figure 8.15 and explained in Table 8.1.

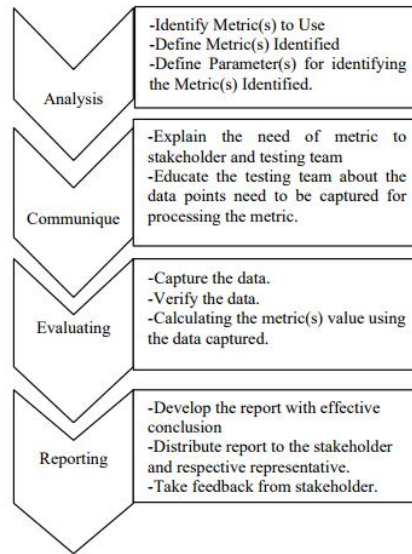


Figure 8.15: Software metrics lifecycle [3]

| Steps to test metrics   | Example   |
|---|---|
| Identify the key software testing processes to be measured  | Testing progress tracking process   |
| In this step, the tester uses the data as baseline to define the metrics                          | The number of test cases planned to be executed per day   |
| Determination of the information to be followed, frequency of tracking and the person responsible | The actual test execution per day will be captured by the test manager at the end of the day                              |
| Effective calculation, management and interpretation of the defined metrics                       | The actual test cases executed per day  |
| Identify the areas of improvement depending on the interpretation of the defined metrics          | The test cases execution falls below the goal ser, we need to investigate the reason and suggest the improvement measures |

Table 8.1: Steps of test metrics and their example [2]

The three types of test metrics are shown in Figure 8.16.

1. **Process Metrics:** It can be used to improve the process efficiency of the SDLC (Software Development Life Cycle)
2. **Product Metrics:** It deals with the quality of the software product
3. **Project Metrics:** It can be used to measure the efficiency of a project team or any testing tools being used by the team members

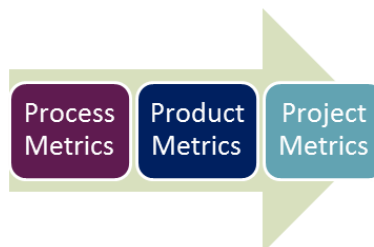


Figure 8.16: Types of test metrics [2]

Identification of correct testing metrics is very important. The test metrics should be reviewed and interpreted on regular basis throughout the test effort and particularly after the application is released into production. There are several key factors in implementing and using the metrics in the organization, beginning with determining the goal for developing the metrics, followed by the identification of metrics to be tracked and ending with sufficient analysis of the resulting data to be able to make changes to the software development lifecycle. Few things need to be considered before identifying the test metrics

- Fix the target audience for the metric preparation
- Define the goal for metrics
- Introduce all the relevant metrics based on project needs
- Analyze the cost benefits aspect of each metrics and the project lifestyle phase in which it results into the maximum output

| Types of metrics | Example   |
|------------------|---|
| Manual           | <ul style="list-style-type: none"> <li>• Test Case Productivity</li> <li>• Test Execution Summary</li> <li>• Defect Acceptance</li> <li>• Defect Rejection</li> <li>• Bad Fix Defect</li> <li>• Test Execution Productivity</li> <li>• Test Efficiency</li> <li>• Defect Severity Index</li> </ul>                  |
| Performance      | <ul style="list-style-type: none"> <li>• Performance Scripting Productivity</li> <li>• Performance Execution Summary</li> <li>• Performance Execution Data -Client Side</li> <li>• Performance Execution Data - Server Side</li> <li>• Performance Test Efficiency</li> <li>• Performance Severity Index</li> </ul> |
| Automation       | <ul style="list-style-type: none"> <li>• Automation Scripting Productivity</li> <li>• Automation Test Execution Productivity</li> <li>• Automation Coverage</li> <li>• Cost Compression</li> </ul>  |
| Common Metrics   | <ul style="list-style-type: none"> <li>• Effort variance</li> <li>• Schedule Variance</li> <li>• Scope change</li> </ul>  |

Table 8.2: Types of metrics and their example [3]

Examples of different software test metrics are mentioned in Table 8.2. For example, manual test metrics is classified into two classes, shown in figure 8.17.

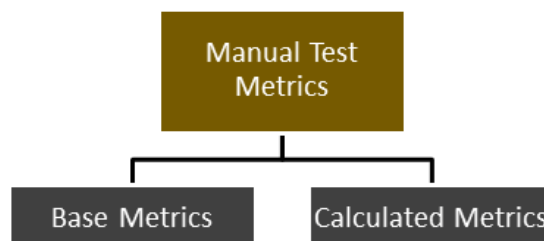


Figure 8.17; Types of manual test metrics [2]

1. **Base Metrics (Direct measure):** Base metrics is the raw data collected by Test Analyst during the test case development and execution. They are used to provide project status reports to the Test Lead and to the Project Manager. The base metrics provide the input to feed into the formulas used to derive **Calculated Metrics**, for example, (# of test cases executed, # of test cases)
2. **Calculated Metrics:** Calculated metrics is derived from the data collected in base metrics. They convert the Base Metrics data into more useful information. The Calculated metrics are generally prepared by the Test Lead and is used to track the progress of the project at different levels like Module level, the Tester level and for the project as whole. The Calculated metrics is usually followed by the Test Manager for test reporting purpose, for example, (% Complete, % Test Coverage).

To understand how to calculate the test metrics, we will see an example of percentage test case executed.

- To obtain the execution status of the test cases in percentage, we use the formula.  

$$\text{Percentage test cases executed} = \left( \frac{\text{No of test cases executed}}{\text{Total no of test cases written}} \right) \times 100$$
- Likewise, you can calculate for other parameters like test cases not executed, test cases passed, test cases failed, test cases blocked, etc.

Other metrics calculations are:

- **Rework Effort Ratio** =  $\left( \frac{\text{Actual rework efforts spent in that phase}}{\text{total actual efforts spent in that phase}} \right) \times 100$
- **Requirement Creep** =  $\left( \frac{\text{Total number of requirements added}}{\text{No of initial requirements}} \right) \times 100$
- **Schedule Variance** =  $\left( \frac{\text{Actual efforts} - \text{estimated efforts}}{\text{Estimated Efforts}} \right) \times 100$
- **Cost of finding defect in testing** =  $\left( \frac{\text{Total effort spent on testing}}{\text{defects found in testing}} \right)$
- **Schedule slippage:**  $\left( \frac{\text{Actual end date} - \text{Estimated end date}}{\text{Planned End Date} - \text{Planned Start Date}} \right) \times 100$

## 8.6. Software Productivity

You can measure productivity in a manufacturing system by counting the number of units that are produced and dividing this by the number of person-hours required to produce them. However, for any software problem, there are many different solutions, each of which has different attributes. One solution may execute more efficiently while another may be more readable and easier to maintain. When solutions with different attributes are produced, comparing their production rates is not really meaningful.

Nevertheless, as a project manager, you may be faced with the problem of estimating, the productivity of software engineers. You may need these productivity estimates to help define the project cost or schedule, to inform investment decisions or to assess, whether processor technology improvements are effective.

Productivity estimates are usually based on measuring attributes of the software and dividing this by the total effort required for development. There are two types of metric that have been used:

1. **Size-related metrics:** These are related to the size of some output from an activity. The most commonly used size-related metric is lines of delivered source code. Other metrics that may be used are the number of delivered object code instructions or the number of pages of system documentation.
2. **Function-related metrics:** These are related to the overall functionality of the delivered software. Productivity is expressed in terms of the amount of useful functionality produced in some given time. Function points and object points are the best-known metrics of this type.

**Lines of source code per programmer-month (LOC/pm)** - is a widely used software productivity metric. You can compute LOC/pm by counting the total number of lines of source code that are delivered, then divide the count by the total time in programmer-months required to complete the project. This time therefore includes the time required for all other activities (requirements, design, coding, testing and documentation) involved in software development.

This approach was first developed when most programming was in FORTRAN, assembly language or COBOL. Then, programs were typed on cards, with one statement on each card. The number of lines of code was easy to count. It corresponded to the number of cards in the program deck. However, programs in languages such as Java or C++ consist of declarations, executable statements and commentary. They may include macroinstructions that expand to several lines of code. There may be more than one statement per line. There is not, therefore, a simple relationship between program statements and lines on a listing.

Comparing productivity across programming languages can also give misleading impressions of programmer productivity. The more expressive the programming language is, the lower the apparent productivity will be. This anomaly arises because all software development activities are considered together when computing the development time, but the LOC metric applies only to the programming process. Therefore, if one language requires more lines than another to implement the same functionality, productivity estimates will be anomalous.

For example, consider an embedded real-time system that might be coded in 5,000 lines of assembly code or 1,500 lines of C. The development time for the various phases is shown in Figure 8.18. The assembler programmer has a productivity of 714 lines /month and the high-level language programmer less than half of this – 300 lines /month. Yet the development costs for the system developed in C are lower and it is delivered earlier.

An alternative to using code size as the estimated product attribute is to use some measure of the functionality of the code. This avoids the above anomaly, as functionality is independent of implementation language. Some researchers have described and compared several function-based measures. The best known of these measures is the function-point count, i.e., the practical use of function points in software projects.

Productivity is expressed as the number of function points that are implemented per person-month. A function point is not a single characteristic but is computed by combining several different measurements or estimates. You compute the total number of function points in a program by measuring or estimating the following program features:



- external inputs and outputs
- user interactions
- external interfaces
- files used by the system

Obviously, some inputs and outputs, interactions and so on are more complex than others and take longer to implement. The function-point metric takes this into account by multiplying the initial function-point estimate by a complexity-weighting factor. You should assess each of these features for complexity and then assign the weighting factor that varies from 3 (for simple external inputs) to 15 for complex internal files. Either the weighting values proposed by Albrecht or values based on local experience may be used.

You can then compute the so-called unadjusted function-point count (UFC) by multiplying each initial count by the estimated weight and summing all values.

$$\text{UFC} = \sum (\text{number of elements of given type}) \times (\text{weight})$$

You then modify this unadjusted function-point count by additional complexity factors that are related to the complexity of the system as a whole. This takes into account the degree of distributed processing, the amount of reuse, the performance, and so on. The unadjusted function-point count is multiplied by these project complexity factors to produce a final function-point count for the overall system.

The subjective nature of complexity estimates means that the function-point count in a program depends on the estimator. Different people have different notions of complexity. There are therefore wide variations in function-point count depending on the estimator's judgment and the type of system being developed. Furthermore, function points are biased towards data-processing systems that are dominated by input and output operations. It is harder to estimate function-point counts for event-driven systems. For this reason some people think that function points are not a very useful way to measure software productivity. However, users of function points argue that in spite of their laws, they are effective in practical situations.

Object points are an alternative to function points. They can be used with languages such as database programming languages or scripting languages. Object points are not object classes that may be produced when an object oriented approach is taken to software development. Rather, the number of object points in a program is a weighted estimate of:

- ***The number of separate screens that are displayed:*** Simple screens count as 1 object point, moderately complex screens count as 2, and very complex screens count as 3 object points.
- ***The number of reports that are produced:*** For simple reports, count 2 object points, for moderately complex reports, count 5, and for reports that are likely to be difficult to produce, count 8 object points.
- ***The number of modules in imperative programming languages such as Java or C++ that must be developed to supplement the database programming code:*** Each of these modules counts as 10 object points.

Object points are used in the COCOMO II estimation model (where they are called application points). The advantage of object points over function points is that they are easier to estimate from a high-level

software specification. Object points are only concerned with screens, reports and modules in conventional programming languages. They are not concerned with implementation details, and the complexity factor estimation is much simpler.

If function points or object points are used, they can be estimated at an early stage in the development process before decisions that affect the program size have been made. Estimates of these parameters can be made as soon as the external interactions of the system have been designed. At this stage, it is very difficult to produce an accurate estimate of the size of a program in lines of source code.

Function-point and object-point counts can be used in conjunction with lines of code-estimation models. The final code size is calculated from the number of function points. Using historical data analysis, the average number of lines of code, AVC, in a particular language required to implement a function point can be estimated.

Values of AVC vary from 200 to 300 LOC/FP in assembly language to 2 to 40 LOC/FP for a database programming language such as SQL. The estimated code size for a new application is then computed as follows:

**Code size** = AVC X Number of function points

| Factor                        | Description  |
|-------------------------------|--|
| Application domain experience | Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive. |
| Process quality               | The development process used can have a significant effect on productivity.  |
| Project size                  | The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.                  |
| Technology support            | Good support technology such as CASE tools and configuration management systems can improve productivity.  |
| Working environment           | A quiet working environment with private work areas contributes to improved productivity.  |

Table 8.3: Factors affecting software engineering productivity [1]

The programming productivity of individuals working in an organization is affected by a number of factors. Some of the most important of these are summarized in Table 8.3. However, individual differences in ability are usually more significant than any of these factors. In a nearly assessment of productivity, it has been found that some programmers were more than 10 times more productive than others. Large teams are likely to have a mix of abilities and experience and so will have average productivity. In small teams, however, overall productivity is mostly dependent on individual aptitudes and abilities.

Software development productivity varies dramatically across application domains and organizations. For large, complex, embedded systems, productivity has been estimated to be low as 30 LOC/pm. For straight forward, well-understood application systems, written in a language such as Java, it may be as high as 900 LOC/pm. When measured in terms of object points, it has been suggested that productivity varies from 4 object points per month to 50 per month, depending on the type of application, tool support and developer capability.

The problem with measures that rely on the amount produced in a given time period is that they take no account of quality characteristics such as reliability and maintainability. They imply that more always

means better. For example in terms of extreme programming (XP), if your approach is based on continuous code simplification and improvement, then counting lines of code doesn't mean much.

These measures also do not take into account the possibility of reusing the software produced, using code generators and other tools that help create the software. What we really want to estimate is the cost of deriving a particular system with given functionality, quality, performance, maintainability, and so on. This is only indirectly related to tangible measures such as the system size.

As a manager, you should not use productivity measurements to make hasty judgments about the abilities of the engineers on your team. If you do, engineers may compromise on quality in order to become more 'productive'. It may be the case that the 'less-productive' programmer produces more reliable code-code that is easier to understand and cheaper to maintain. You should always, therefore, think of productivity measures as providing partial information about programmer productivity. You also need to consider other information about the quality of the programs that are produced.

### 8.7. Estimation Techniques

There is no simple way to make an accurate estimate of the effort required to develop a software system. You may have to make initial estimates on the basis of a high level user requirements definition. The software may have to run on unfamiliar computers or use new development technology. The people involved in the project and their skills will probably not be known. All of these mean that it is impossible to estimate system development costs accurately at an early stage in a project.

Furthermore, there is a fundamental difficulty in assessing the accuracy of different approaches to cost-estimation techniques. Project cost estimates are often self fulfilling. The estimate is used to define the project budget, and the product is adjusted so that the budget figure is realized. A controlled experiment would not reveal the cost estimate to the project manager. The actual costs would then be compared with the estimated project costs.

However, such an experiment is probably impossible because of the high costs involved and the number of variables that cannot be controlled. Nevertheless, organizations need to make software effort and cost estimates. To do so, one or more of the techniques described in Table 8.4, may be used. All of these techniques rely on experience-based judgments by project managers who use their knowledge of previous projects to arrive at an estimate of the resources required for the project. However, there may be important differences between past and future projects. Many new development methods and techniques have been introduced in the last 10 years. Some examples of the changes that may affect estimates based on experience include:

- Distributed object systems rather than mainframe-based systems
- Use of web services
- Use of ERP or database-centered systems
- Use of off-the-shelf software rather than original system development
- Development for and with reuse rather than new development of all parts of a system
- Development using scripting languages such as TCL or Perl
- The use of CASE tools and program generators rather than unsupported software development

| Technique                 | Description   |
|---------------------------|---|
| Algorithmic cost modeling | A model is developed using historical cost information that relates some software metric (usually its size) to the project cost. An estimate is made of that metric and the model predicts the effort required.   |
| Expert judgment           | Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.                               |
| Estimation by analogy     | This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects.   |
| Parkinson's law           | Parkinson's Law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months. |
| Pricing to win            | The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.  |

Table 8.4: Cost estimation techniques [1]

If project managers have not worked with these techniques, their previous experience may not help them estimate software project costs. This makes it more difficult for them to produce accurate costs and schedule estimates.

You can tackle the approaches to cost estimation shown in Table 8.5 using either top-down or a bottom-up approach. *A top-down approach* starts at the system level. You start by examining the overall functionality of the product and how that functionality is provided by interacting sub-functions. The costs of system-level activities, such as integration, configuration management and documentation are taken into account.

*The bottom-up approach*, by contrast, starts at the component level. The system is decomposed into components, and you estimate the effort required to develop each of these components. You then add these component costs to compute the effort required for the whole system development.

The disadvantages of the top-down approach are the advantages of the bottom-up approach and vice versa. Top-down estimation can underestimate the costs of solving difficult technical problems associated with specific components such as interfaces to non standard hardware. There is no detailed justification of the estimate that is produced. By contrast, bottom-up estimation produces such a justification and considers each component. However, this approach is more likely to underestimate the costs of system activities such as integration. Bottom-up estimation is also more expensive. There must be an initial system design to identify the components to be costed.

Each estimation technique has its own strengths and weaknesses. Each of them uses different information about the project and the development team, so if you use a single model and this information is not accurate, your final estimate will be wrong.

For large projects, therefore, you should use several cost estimation techniques and compare their results. If these predict radically different costs, you probably do not have enough information about the product or the development process. You should look for more information about the product, processor team and repeat the costing process until the estimates converge.

These estimation techniques are applicable where a requirements document for the system has been produced. This should define all users and system requirements. You can therefore make a reasonable

estimate of the system functionality that is to be developed. In general, large systems engineering projects will have such a requirements document.

However, in many cases, the costs of many projects must be estimated using only incomplete user requirements for the system. This means that the estimators have very little information with which to work. Requirements analysis and specification is expensive, and the managers in a company may need an initial cost estimate for the system before they can have a budget approved to develop more detailed requirements or a system prototype.

Under these circumstances, 'pricing to win' is a commonly used strategy. The notion of pricing to win may seem unethical and unbusinesslike. However, it does have some advantages. A project cost is agreed on the basis of an outline proposal.

Negotiations then take place between client and customer to establish the detailed project specification. This specification is constrained by the agreed cost. The buyer and seller must agree on what is acceptable system functionality. The fixed factor in many projects is not the project requirements but the cost. The requirements may be changed so that the cost is not exceeded.

For example, say a company is bidding for a contract to develop a new fuel delivery system for an oil company that schedules deliveries of fuel to its service stations. There is no detailed requirements document for this system so the developers estimate that a price of \$900,000 is likely to be competitive and within the oil company's budget. After they are granted the contract, they negotiate the detailed requirements of the system so that basic functionality is delivered; then they estimate the additional costs for other requirements. The oil company does not necessarily lose here because it has awarded the contract to accompany that it can trust. The additional requirements may be funded from a future budget, so that the oil company's budgeting is not disrupted by a very high initial software cost.

## 8.8. Algorithmic Cost Modeling

Algorithmic cost modeling uses a mathematical formula to predict project costs based on estimates of the project size, the number of software engineers, and other process and product factors. An algorithmic cost model can be built by analyzing the costs and attributes of completed projects and finding the closest fit formula to actual experience.

Algorithmic cost models are primarily used to make estimates of software development costs, but there are also a range of other uses for algorithmic cost estimates, including estimates for investors in software companies, estimates of alternative strategies to help assess risks, and estimates to inform decisions about reuse, redevelopment or outsourcing.

In its most general form, an algorithmic cost estimate for software cost can be expressed as:

$$\mathbf{Effort} = A \times \mathbf{Size}^B \times M$$

**A** is a constant for that depends on local organizational practices and the type of software that is developed. **Size** may be either an assessment of the code size of the software or a functionality estimate expressed in functions or object points. The value of exponent **B** usually lies between 1 - 1.5. **M** is a multiplier made by combining process, product and development attributes, such as the dependability requirements for the software and the experience of the development team.

Most algorithmic estimation models have an exponential component ( $B$  in the above equation) that is associated with the size estimate. This reflects the fact that costs do not normally increase linearly with project size. As the size of the software increases, extra costs are incurred because of the communication overhead of larger teams, more complex configuration management, more difficult system integration, and so on. Therefore, the larger the system is, the larger the value of this exponent will be.

Unfortunately, all algorithmic models suffer from the same fundamental difficulties:

- *It is often difficult to estimate **Size** at an early stage in a project when only a specification is available.* Function-point and object-point estimates are easier to produce than estimates of code size but are often still inaccurate.
- *The estimates of the factors contributing to **B** and **M** are subjective.* Estimates vary from one person to another, depending on their background and experience with the type of system that is being developed.

The number of lines of source code in the delivered system is the basic metric used in many algorithmic cost models. Size estimation may involve estimation by analogy with other projects, estimation by converting function or object points to code size, estimation by ranking the sizes of system components and using a known reference component to estimate the component size, or it may simply be a question of engineering judgment.

Accurate code size estimation is difficult at an early stage in a project because the code size is affected by design decisions that have not yet been made. For example, an application that requires complex data management may either use a commercial database or implement its own data-management system. If a commercial database is used, the code size will be smaller but additional effort may be needed to overcome the performance limitations of the commercial product.

The programming language used or system development also affects the number of lines of code to be developed. A language such as Java might mean that more lines of code are necessary than if C (say) were used. However, this extra code allows more compile-time checking so validation costs are likely to be reduced. *How should this be taken into account?* Furthermore, it may be possible to reuse a significant amount of code from previous projects and the size estimate has to be adjusted to take this into account.

*If you use an algorithmic cost estimation model, you should develop a range of estimates (worst, expected and best) rather than a single estimate and apply the costing formula to all of them.* Estimates are most likely to be accurate when you understand the type of software that is being developed, when you have calibrated the costing model using local data and when programming language and hardware choices are predefined.

The accuracy of the estimates produced by an algorithmic model depends on the system information that is available. As the software process proceeds, more information becomes available so estimates become more and more accurate. If the initial estimate of effort required is  $x$  months of effort, this range may be from  $0.25x$  to  $4x$  when the system is first proposed. This narrows during the development process, as shown in Figure 8.18. This figure reflects experience of a large number of software development projects. Of course, just before the system is delivered, a very accurate estimate can be made.

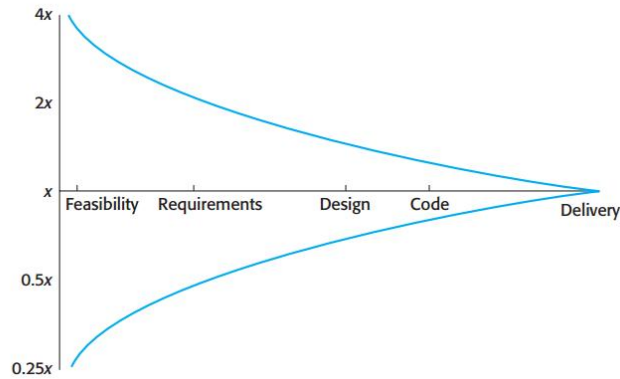


Figure 8.18: Estimate uncertainty [1]

## THE *COCOMO* MODEL

A number of algorithmic models have been proposed as the basis for estimating the *effort, schedule and costs of a software project*. These are conceptually similar but use different parameter values. The model that is chosen here is the COCOMO model.

The COCOMO model is an empirical model that was derived by collecting data from a large number of software projects. These data were analyzed to discover formulae that were the best fit to the observations. These *formulae link the size of the system and product, project and team factors to the effort to develop the system*.

The reasons for choosing the COCOMO model are:

- It is well documented, available in the public domain and supported by public domain and commercial tools.
- It has been widely used and evaluated in a range of organizations.
- It has a long pedigree from its first instantiation in 1981, through a refinement tailored to Ada software development to its most recent version, COCOMO II, published in 2000.

The COCOMO models are comprehensive with a large number of parameters that can each take a range of values.

The first version of the COCOMO model (COCOMO 81) was a three-level model where the levels corresponded to the detail of the analysis of the cost estimate. The **first level (basic)** provided an initial rough estimate; the **second level** modified this using a number of project and process multipliers; and the **most detailed level** produced estimates for different phases of the project. Table 8.5 shows the basic COCOMO formula for different types of projects. The multiplier **M** reflects product, project and team characteristics.

| Project complexity | Formula                         | Description   |
|--------------------|---------------------------------|---|
| Simple             | $PM=2.4 (KDSI)^{1.05} \times M$ | Well-understood applications developed by small teams   |
| Moderate           | $PM=3.0 (KDSI)^{1.12} \times M$ | More complex projects where team members may have limited experience of related systems   |
| Embedded           | $PM=3.6 (KDSI)^{1.2} \times M$  | Complex projects where the software is part of a strongly coupled complex hardware, software, regulations and operational procedures. |

Table 8.5: The basic COCOMO 81 model [1]

**COCOMO81 assumed that the software would be developed according to a waterfall process using standard imperative programming languages such as C or FORTRAN.** However, there have been radical changes to software developments since this initial version was proposed. Prototyping and incremental development are commonly used process models. Software is now often developed by assembling reusable components with off-the-shelf systems and 'gluing' them together with scripting language. Data-intensive systems are developed using a database programming language such as SQL and a commercial database management system. Existing software is re-engineered to create new software. CASE tool support for most software process activities is now available.

To take these changes into account, the COCOMOII model recognizes different approaches to software development such as prototyping, development by component composition and use of database programming. **COCOMOII supports a spiral model of development and embeds several sub-models that produce increasingly detailed estimates. These can be used in successive rounds of the development spiral.** Figure 8.19 shows COCOMO II sub-models and where they are used.

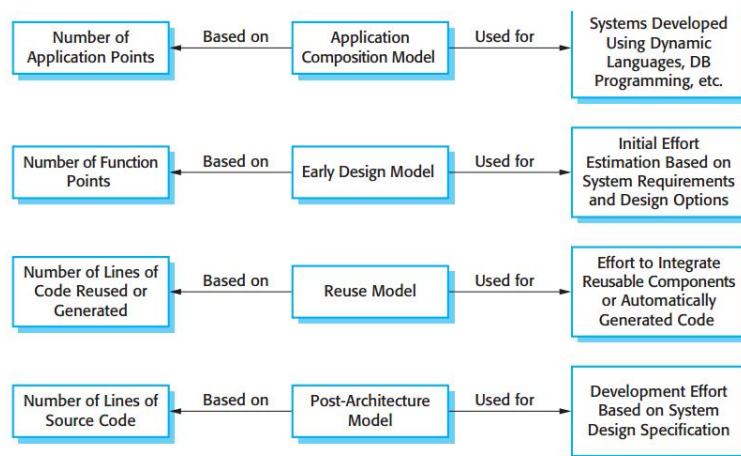


Figure 8.19: The COCOMO II models [1]

The sub-models that are part of the COCOMOII model are:

1. **An application-composition model:** This assumes that systems are created from reusable components, scripting or database programming. It is designed to make estimates of prototype development. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required. Application points are the same as object points, but the name was changed to avoid confusion with objects in object-oriented development.



2. ***A nearly design model:*** This model is used during early stages of the system design after the requirements have been established. Estimates are based on function points, which are then converted to number of lines of source code. The formula follows the standard form discussed above with a simplified set of seven multipliers.
3. ***A reuse model:*** This model is used to compute the effort required to integrate reusable components and /or program code that is automatically generated by design or program translation tools. It is usually used in conjunction with the post-architecture model.
4. ***A post-architecture model:*** Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again this model uses the standard formula for cost estimation. However, it includes a more extensive set of 17 multipliers reflecting personnel capability and product and project characteristics.

Of course, in large systems, different parts may be developed using different technologies and you may not have to estimate all parts of the system to the same level of accuracy. In such cases, you can use the appropriate sub-model for each part of the system and combine the results to create a composite estimate.

### The Application-Composition Model

The application-composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing existing components. It is based on an estimate of weighted application points (object points) divided by a standard estimate of application-point productivity. The estimate is then adjusted according to the difficulty of developing each object point. Programmer productivity also depends on the developers experience and capability as well as the capabilities of the CASE tools used to support development. Table 8.6 shows the levels of object-point productivity suggested by the model developers.

|                                       |          |     |         |      |           |
|---------------------------------------|----------|-----|---------|------|-----------|
| Developer's experience and capability | Very low | Low | Nominal | High | Very high |
| CASE maturity and capability          | Very low | Low | Nominal | High | Very high |
| PROD (NOP/month)                      | 4        | 7   | 13      | 25   | 50        |

Table 8.6: Object-Point Productivity [1]

Application composition usually involves significant software reuse, and some of the total number of application points in the system may be implemented with reusable components. Consequently, you have to adjust the estimate based on the total number of application points to take into account the percentage of reuse expected.

Therefore, the final formula for effort computation for system prototypes is:

$$PM = (NAP \times (1 - \% \text{ reuse}/100)) / PROD$$

***PM*** is the effort estimate in person-months. ***NAP*** is the total number of application points in the delivered system. ***%reuse*** is an estimate of the amount of reused code in the development. ***PROD*** is the object-point productivity as shown in Table 8.7. ***The model simplistically assumes that there is no additional effort involved in reuse.***

## The Early Design Model

This model is used once user requirements have been agreed and initial stages of the system design process are underway. However, you don't need a detailed architectural design to make these initial estimates. Your goal at this stage should be to make an approximate estimate without undue effort. Consequently, you make various simplifying assumptions, such as that the effort involved in integrating reusable code is zero. Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements.

The estimates produced at this stage are based on the standard formula for algorithmic models, namely

$$\mathbf{Effort} = A \times \text{Size}^B \times M$$

The coefficient  $A$  is 2.94. The *Size* of the system is expressed in KSLOC, which is the number of thousands of lines of source code. You calculate KSLOC by estimating the number of function points in the software. You then use standard tables that relate software size to function points for different programming languages to compute an initial estimate of the system size in KSLOC.

The exponent  $B$  reflects the increased effort required as the size of the project increases. This is not fixed for different types of systems, as in COCOMO81, but can vary from 1.1 to 1.24 depending on the novelty of the project, the development flexibility, the risk resolution processes used, the cohesion of the development team and the process maturity level of the organization.

The multiplier  $M$  in COCOMOII is based on a simplified set of seven project and process characteristics that influence the estimate. These can increase or decrease the effort required. These characteristics used in the early design model are product reliability and complexity ( $RCPX$ ), reuse required ( $RUSE$ ), platform difficulty ( $PDIF$ ), personnel capability ( $PERS$ ), personnel experience ( $PREX$ ), schedule ( $SCED$ ) and support facilities ( $FCIL$ ). You estimate values for these attributes using a six-point scale where 1 corresponds to very low values for these multipliers and 6 corresponds to very high values.

This results in an effort computation as follows:

$$\mathbf{PM} = 2.94 \times \text{Size}^B \times M$$

Where,

$$M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED$$

## The Reuse Model

The reuse model is used to estimate the effort required to integrate reusable or generated code. COCOMO II considers reused code to be of two types. Black-box code is code that can be reused without understanding the code or making changes to it. The development effort for black-box code is taken to be zero. Code that has to be adapted to integrate it with new code or other reused components is called white-box code. Some development effort is required to reuse this because it has to be understood and modified before it can work correctly in the system.

In addition, many systems include automatically generated code from program translators that generate code from system models. This is a form of reuse where standard templates are embedded in the generator. The system model is analyzed, and code based on these standard templates with additional details from the system model is generated. The COCOMO II reuse model includes a separate model to estimate the costs associated with this generated code.

For code that is automatically generated, the model estimates the number of person months required to integrate this code. The formula for effort estimation is:

$$PM_{\text{Auto}} = (ASLOC \times AT/100) / ATPROD \quad // \text{Estimate for generated code}$$

*AT* is the percentage of adapted code that is automatically generated and *ATPROD* is the productivity of engineers in integrating such code. The value of ATPROD is about 2,400 source statements per month.

Therefore, if there is a total of 20,000 lines of white-box reused code in a system and 30% of this is automatically generated, then the effort required to integrate this generated code is:

$$(20,000 \times 30 / 100) / 2400 = 2.5 \text{ person months} \quad // \text{Generated code example}$$

The other component of the reuse model is used when a system includes some new code and some reused white-box components that have to be integrated. In this case, the reuse model does not compute the effort-directly. Rather, based on the number of lines of code that are reused, it calculates a figure that represents the equivalent number of lines of new code.

Therefore, if 30,000 lines of code are to be reused, the new equivalent size estimate might be 6,000. Essentially, reusing 30,000 lines of code is taken to be equivalent to writing 6,000 lines of new code. This calculated figure is added to the number of lines of new code to be developed in the COCOMOII post-architecture model.

The estimates in this reuse model are:

**ASLOC**-the number of lines of code in the components that have to be adapted

**ESLOC**-the equivalent number of lines of new source code

The formula used to compute ESLOC takes into account the effort required for software understanding, for making changes to the reused code and for making changes to the system to integrate that code. It also takes into account the amount of code that is automatically generated where the development effort is calculated, as explained earlier in this section.

The following formula is used to calculate the number of equivalent lines of source code:

$$ESLOC = ASLOC \times (1 - AT/100) \times AAM$$

*ASLOC* is reduced according to the percentage of automatically generated code. *AAM* is the Adaptation Adjustment Multiplier, which takes into account the effort required to reuse code. Simplistically, AAM is the sum of three components:

- **An adaptation component (referred to as AAF)** that represents the costs of making changes to the reused code. This includes components that take into account design, code and integration changes.
- **An understanding component (referred to as SU)** that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code. SU ranges from 50 for complex unstructured code to 10 for well-written, object-oriented code.
- **An assessment factor (referred to as AA)** that represents the costs of reuse decision making. That is, some analysis is always required to decide whether code can be reused, and this is included in the cost as AA. AA varies from 0 to 8 depending on the amount of analysis effort required.

The reuse model is a nonlinear model. Some effort is required if reuse is considered to make an assessment of whether reuse is possible. Furthermore, as more and more reuse is contemplated, the costs per code unit reused drop as the fixed understanding and assessment costs are spread across more lines of code.

### The Post-Architecture Model

The post-architecture model is the most detailed of the COCOMO II models. It is used once an initial architectural design for the system is available so the sub-system structure is known.

The estimates produced at the post-architecture level are based on the same basic formula ( $PM = A \times Size^B \times M$ ) used in the early design estimates. However, the size estimate for the software should be more accurate by this stage in the estimation process. In addition, a much more extensive set of product, process and organizational attributes (17 rather than 7) are used to refine the initial effort computation. It is possible to use more attributes at this stage because you have more information about the software to be developed and the development process.

The estimate of the code size in the post-architecture model is computed using three components:

1. An estimate of the total number of lines of new code to be developed
2. An, estimate of the equivalent number of source lines of code (ESLOC) calculated using the reuse model
3. An estimate of the number of lines of code that have to be modified because of changes to the requirements

These three estimates are added to give the total code size in KSLOC that you use in the effort computation formula. The final component in the estimate-the number of lines of modified code-reflects the fact that software requirements always change. The system programs have to reflect these requirements changes so additional code has to be developed. Of course, estimating the number of lines of code that will change is not easy and there will often be even more uncertainty in this figure than in development estimates.

The exponent term (**B**) in the effort computation formula had three possible values in COCOMO I. These were related to the levels of project complexity. As projects become more complex, the effects of increasing system size become more significant. However, good organizational practices and procedures can control this '*diseconomy of scale*'. This is recognized in COCOMOII, where the range of values for

the exponent B is continuous rather than discrete. The exponent is based on five scale factors, as shown in Table 8.7. These factors are rated on a six-point scale from Very low to Extra high (5 to 0). You should then add the ratings, divide them by 1100 and add the result to 1.01 to get the exponent that should be used.

| Scale factor                 | Explanation  |
|------------------------------|--|
| Precedentedness              | Reflects the previous experience of the organization with this type of project. Very low means no previous experience; Extra high means that the organization is completely familiar with this application domain. |
| Development flexibility      | Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client sets only general goals.  |
| Architecture/risk resolution | Reflects the extent of risk analysis carried out. Very low means little analysis; Extra high means a complete and thorough risk analysis.  |
| Team cohesion                | Reflects how well the development team knows each other and work together. Very low means very difficult interactions; Extra high means an integrated and effective team with no communication problems.           |
| Process maturity             | Reflects the process maturity of the organization. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.  |

Table 8.7: Scale factors used in COCOMO II exponent computation [1]

To illustrate this, imagine that an organization is taking on a project in a domain where it has little previous experience. The project client has not defined the process to be used and has not allowed time in the project schedule for significant risk analysis. A new development team must be put together to implement this system. The organization has recently put a process improvement program in place and has rated as Level 2 organization according to the CMM model. Possible values for the rating used in exponent calculation are:

- **Precedentedness:** This is a new project for the organization- rated Low (4)
- **Development flexibility:** No client involvement-rated Very high (1)
- **Architecture/risk resolution:** No risk analysis carried out-rated Very low (5)
- **Team cohesion:** New team so no information-rated Nominal (3)
- **Process maturity:** Some process control in place-rated Nominal (3)

The sum of these values is 16, so you calculate the exponent by adding 0.16 to 1.01, getting a value of 1.17.

The attributes (Table 8.8) that are used to adjust the initial estimates and create multiplier **M** in the post-architecture model fall into four classes:

1. **Product attributes** are concerned with required characteristics of the software product being developed.
2. **Computer attributes** are constraints imposed on the software by the hardware platform.
3. **Personnel attributes** are multipliers that take the experience and capabilities of the people working on the project into account.
4. **Project attributes** are concerned with the particular characteristics of the software development project.

| Attribute | Type      | Description  |
|-----------|-----------|--|
| RELY      | Product   | Required system reliability  |
| CPLX      | Product   | Complexity of system modules   |
| DOCU      | Product   | Extent of documentation required                                     |
| DATA      | Product   | Size of database used  |
| RUSE      | Product   | Required percentage of reusable components                           |
| TIME      | Computer  | Execution time constraint  |
| PVOL      | Computer  | Volatility of development platform                                   |
| STOR      | Computer  | Memory constraints   |
| ACAP      | Personnel | Capability of project analysts                                       |
| PCON      | Personnel | Personnel continuity   |
| PCAP      | Personnel | Programmer capability  |
| PEXP      | Personnel | Programmer experience in project domain                              |
| AEXP      | Personnel | Analyst experience in project domain                                 |
| LTEX      | Personnel | Language and tool experience   |
| TOOL      | Project   | Use of software tools  |
| SCED      | Project   | Development schedule compression                                     |
| SITE      | Project   | Extent of multisite working and quality of inter-site communications |

Table 8.8: Attributes to Adjust the Initial Estimates and Create Multiplier M [1]

Figure 8.20 shows how these cost drivers influence effort estimates. In this, the value for the exponent is taken as 1.17 and assume that RELY, CPLX, STOR, TOOL and SCED are the key cost drivers in the project. All of the other cost drivers have a nominal value of 1, so they do not affect the computation of the effort.

|   |                                |
|---|--------------------------------|
| Exponent value  | 1.17                           |
| System size (including factors for reuse and requirements volatility) | 128,000 DSI                    |
| <b>Initial COCOMO estimate without cost drivers</b>                   | <b>730 person-months</b>       |
| Reliability   | Very high, multiplier = 1.39   |
| Complexity  | Very high, multiplier = 1.3    |
| Memory constraint   | High, multiplier = 1.21        |
| Tool use  | Low, multiplier = 1.12         |
| Schedule  | Accelerated, multiplier = 1.29 |
| <b>Adjusted COCOMO estimate</b>                                       | <b>2,306 person-months</b>     |
| Reliability   | Very low, multiplier = 0.75    |
| Complexity  | Very low, multiplier = 0.75    |
| Memory constraint   | None, multiplier = 1           |
| Tool use  | Very high, multiplier = 0.72   |
| Schedule  | Normal, multiplier = 1         |
| <b>Adjusted COCOMO estimate</b>                                       | <b>295 person-months</b>       |

Figure 8.20: The effect of cost drivers on effort estimates [1]

In Figure 8.20, maximum and minimum values to the key cost drivers are assigned to show how they influence the effort estimate. The values taken are those from the COCOMOII reference manual. You can see that high values for the cost drivers lead to an effort estimate that is more than three times the initial

estimate, whereas low values reduce the estimate to about one third of the original. This highlights the vast differences between different types of project and the difficulties of transferring experience from one application domain to another.

These formulae proposed by the developers of the COCOMOII model reflects their experience and data, but it is an extremely complex model to understand and use. There are many attributes and considerable scope for uncertainty in estimating their values. In principle, each user of the model should calibrate the model and the attribute values according to its own historical project data, as this will reflect local circumstances that affect the model.

In practice, however, few organizations have collected enough data from past projects in a form that supports model calibration. Practical use of COCOMOII therefore has to start with the published values for the model parameters, and it is impossible for a user to know how closely these relate to their own situation. This means that the practical use of the COCOMO model is limited. Very large organizations may have the resources to employ a cost-modeling expert to adapt and use the COCOMOII models. However, for the majority of companies, the cost of calibrating and learning to use an algorithmic model such as the COCOMO model is so high that they are unlikely to introduce this approach.

## ALGORITHMIC COST MODELS IN PROJECT PLANNING

One of the most valuable uses of algorithmic cost modeling is to compare different ways of investing money to reduce project costs. This is particularly important where you have to make hardware/software cost trade-offs and where you may have to recruit new staff with specific project skills. The algorithmic code model helps you assess the risks of each option. Applying the cost model reveals the financial exposure that is associated with different management decisions.

Consider an embedded system to control an experiment that is to be launched into space. Spaceborne experiments have to be very reliable and are subject to stringent weight limits. The number of chips on a circuit board may have to be minimized. In terms of the COCOMO model, the multipliers based on computer constraints and reliability are greater than 1.

There are three components to be taken into account in costing this project:

- The cost of the target hardware to execute the system
- The cost of the platform (computer plus software) to develop the system
- The cost of the effort required to develop the software

Table 8.9 shows some possible options for this project. These include spending more on target hardware to reduce software costs or investing in better development tools.

| Option | RELY | STOR | TIME | TOOLS | LTEX | Total Effort | Software Cost | Hardware Cost | Total Cost |
|--------|------|------|------|-------|------|--------------|---------------|---------------|------------|
| A      | 1.39 | 1.06 | 1.11 | 0.86  | 1    | 63           | 949393        | 100000        | 1049393    |
| B      | 1.39 | 1    | 1    | 1.12  | 1.22 | 88           | 1313550       | 120000        | 1402025    |
| C      | 1.39 | 1    | 1.11 | 0.86  | 1    | 60           | 895653        | 105000        | 1000653    |
| D      | 1.39 | 1.06 | 1.11 | 0.86  | 0.84 | 51           | 769008        | 100000        | 897490     |
| EX     | 1.39 | 1    | 1    | 0.72  | 1.22 | 56           | 844425        | 220000        | 1044159    |
| F      | 1.39 | 1    | 1    | 1.12  | 0.84 | 57           | 851180        | 120000        | 1002706    |

Table 8.9: Cost of management options [1]

Additional hardware costs may be acceptable because the system is a specialized system that does not have to be mass-produced. If hardware is embedded in consumer products, however, investing in target hardware to reduce software costs increases the unit cost of the product, irrespective of the number sold, which is usually undesirable.

Table 8.9 shows the hardware, software and total costs for the options A-F shown in Figure 8.21. Applying the COCOMOII model without cost drivers predicts an effort of 45 person-months to develop an embedded software system for this application. The average cost for one person-month of effort is \$15,000.

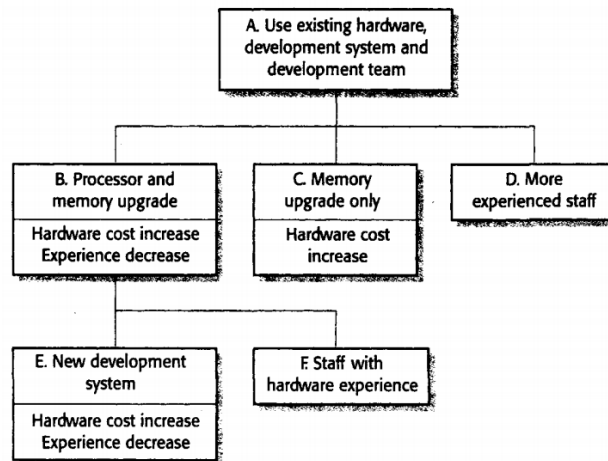


Figure 8.21: Management options [1]

The relevant multipliers are based on storage and execution time constraints (TIME and STOR), the availability of tool support (cross-compilers, etc.) for the development system (TOOL), and development team's experience platform experience (LTEX). In all options, the reliability multiplier (RELY) is 1.39, indicating that significant extra effort is needed to develop a reliable system.

The software cost (SC) is computed as follows:

$$SC = \text{Effort estimate} \times \text{RELY} \times \text{TIME} \times \text{STOR} \times \text{TOOL} \times \text{EXP} \times \$15,000$$

Option A represents the cost of building the system with existing support and staff. It represents a baseline for comparison. All other options involve either more hardware expenditure or the recruitment (with associated costs and risks) of new staff. Option B shows that upgrading hardware does not necessarily reduce costs. The staff lack experience with the new hardware so the increase in the experience multiplier negates the reduction in the STOR and TIME multipliers. It is actually more cost-effective to upgrade memory rather than the whole computer configuration.

Option D appears to offer the lowest costs for all basic estimates. No additional hardware expenditure is involved but new staff must be recruited on to the project. If these are already available in the company, this is probably the best option to choose. If not, they must be recruited externally, which involves significant costs and risks. These may mean that the cost advantages of this option are much less significant than suggested by Table 8.10. Option C offers a saving of almost \$50,000 with virtually no



associated risk. Conservative project managers would probably select this option rather than the riskier Option D.

The comparisons show the importance of staff experience as a multiplier. If good quality people with the right experience are recruited, this can significantly reduce project costs. It also reveals that investment in new hardware and tools may not be cost-effective. Some engineers may prefer this option because it gives them an opportunity to learn about and work with new systems. However, the loss of experience is a more significant effect on the system cost than the savings that arise from using the new hardware system.

## 8.9. Project Duration and Staffing

As well as estimating the effort required to develop a software system and the overall project costs, project managers must also estimate how long the software will take to develop and when staff will be needed to work on the project. The development time for the project is called the *project schedule*. Increasingly, organizations are demanding shorter development schedules so that their products can be brought to market before their competitors.

The relationship between the number of staff working on a project, the total effort required and the development time is not linear. As the number of staff increases, more effort may be needed. The reason for this is that people spend more time communicating and defining interfaces between the parts of the system developed by other people. Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved.

The COCOMO model includes a formula to estimate the calendar time (TDEV) required to complete a project. The time computation formula is the same for all COCOMO levels:

$$\mathbf{TDEV} = 3 \times (\mathbf{PM})^{(0.33+0.2 \times (\mathbf{B}-1.01))}$$

*PM* is the effort computation and *B* is the exponent computed, (*B is 1 for the early prototyping model*). This computation predicts the nominal schedule for the project.

However, the predicted project schedule and the schedule required by the project plan are not necessarily the same thing. The planned schedule may be shorter or longer than the nominal predicted schedule. However, there is obviously a limit to the extent of schedule changes, and the COCOMO II model predicts this:

$$\mathbf{TDEV} = 3 \times (\mathbf{PM})^{(0.33+0.2 \times (\mathbf{B}-1.01))} \times \mathbf{SCED\ Percentage} / 100$$

*SCED Percentage* is the percentage increase or decrease in the nominal schedule. If the predicted figure then differs significantly from the planned schedule, it suggests that there is a high risk of problems delivering the software as planned.

To illustrate the COCOMO development schedule computation, assume that 60 months of effort are estimated to develop a software system. Assume that the value of exponent B is 1.17. From the schedule equation the time required to complete the project is:

$$\mathbf{TDEV} = 3 \times (60)^{0.36} = 13 \text{ months}$$

In this case, there is no schedule compression or expansion, so the last term in the formula has no effect on the computation.

An interesting implication of the COCOMO model is that the time required to complete the project is a function of the total effort required for the project. It does not depend on the number of software engineers working on the project. ***This confirms the notion that adding more people to a project that is behind schedule is unlikely to help that schedule to be regained.*** Projects are likely to run into significant problems if they try to develop software without allowing sufficient calendar time.

## REFERENCES

- [1] I. Sommerville , 2008. Software Engineering. Eighth Edition, Addison-Wesley
- [2] Guru99, 2018, <https://www.guru99.com/software-testing-metrics-complete-tutorial.html>
- [3] P.B. Nirpal & K.V. Kale, 2011, A Brief Overview of Software Testing Metrics. International Journal on Computer Science and Engineering, Vol. 3 No. 1 Jan 2011.

## Appendix:

### Basic COCOMO [\[ edit \]](#)

Basic COCOMO compute software development effort (and cost) as a function of program size. Program size is expressed in estimated thousands of source lines of code (SLOC, KLOC).

COCOMO applies to three classes of software projects:

- Organic projects - "small" teams with "good" experience working with "less than rigid" requirements
- Semi-detached projects - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements
- Embedded projects - developed within a set of "tight" constraints. It is also combination of organic and semi-detached projects.(hardware, software, operational, ...)

The basic COCOMO equations take the form

Effort Applied (E) =  $a_b(KLOC)^{b_b}$  [ man-months ]

Development Time (D) =  $c_b(\text{Effort Applied})^{d_b}$  [months]

People required (P) = Effort Applied / Development Time [count]

where, KLOC is the estimated number of delivered lines (expressed in thousands ) of code for project. The coefficients  $a_b$ ,  $b_b$ ,  $c_b$  and  $d_b$  are given in the following table (note: the values listed below are from the original analysis, with a modern reanalysis<sup>[4]</sup> producing different values):

| Software project | $a_b$ | $b_b$ | $c_b$ | $d_b$ |
|------------------|-------|-------|-------|-------|
| Organic          | 2.4   | 1.05  | 2.5   | 0.38  |
| Semi-detached    | 3.0   | 1.12  | 2.5   | 0.35  |
| Embedded         | 3.6   | 1.20  | 2.5   | 0.32  |

Basic COCOMO is good for quick estimate of software costs. However it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and so on.

## 9. QUALITY MANAGEMENT

Problems with software quality were initially discovered in the 1960s with the development of the first large software systems, and continued to plague software engineering throughout the 20<sup>th</sup> century. Delivered software was slow and unreliable, difficult to maintain and hard to reuse. Dissatisfaction with this situation led to the adoption of formal techniques of software quality management, which have been developed from methods used in the manufacturing industry. These quality management techniques, in conjunction with new software technologies and better software testing, have led to significant improvements in the general level of software quality.

Software quality is a complex concept that is not directly comparable with quality in manufacturing. In manufacturing, the notion of quality has been that the developed product should meet its specification. In an ideal world this definition should apply to all products but, for software systems, there are problems with this:

- The specification should be oriented towards the characteristics of the product that the customer wants. However, the development organization may also have requirements (such as maintainability requirements) that are not included in the specification.
- We do not know how to specify certain quality characteristics (e.g., maintainability) in an unambiguous way.
- It is very difficult to write complete software specifications. Therefore, although a software product may conform to its specification, users may not consider it to be a high-quality product because it does not meet their expectations.

You have to recognize the problems with existing software specifications and therefore design quality procedures that do not rely on having a perfect specification. In particular, software attributes such as maintainability, security or efficiency cannot be specified explicitly. However, they have a large effect on the perceived quality of the system.

Software quality management for software systems has three principal concerns:

1. ***At the organizational level***, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high quality software. This means that the quality management team should take responsibility for defining the software development processes to be used and standards that should apply to the software and related documentation, including the system requirements, design, and code.
2. ***At the project level***, quality management involves the application of specific quality processes, checking that these planned processes have been followed, and ensuring that the project outputs are conformant with the standards that are applicable to that project.
3. ***Quality management at the project level is also concerned with establishing a quality plan for a project***. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

Software quality management for large systems can be structured into three main activities:

1. ***Quality assurance***: The establishment of a framework of organizational procedures and standards that lead to high-quality software

2. **Quality planning:** The selection of appropriate procedures and standards from this framework, adapted for a specific software project
3. **Quality control:** The definition and enactment of processes that ensure the software development team have followed project quality procedures and standards

The terms ‘*quality assurance*’ and ‘*quality control*’ are widely used in manufacturing industry. Quality assurance (QA) is the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process. Quality control is the application of this quality processes to weed out products that are not of the required level of quality.

In the software industry, different companies and industry sectors interpret **quality assurance and quality control in different ways**. Sometimes, *quality assurance simply means the definition of procedures, processes, and standards that are aimed at ensuring that software quality is achieved*. In other cases, *quality assurance also includes all configuration management, verification, and validation activities that are applied after a product has been handed over by a development team*. In this chapter, the term ‘*quality assurance*’ is used to include verification and validation and the processes of checking that quality procedures have been properly applied. The term ‘*quality control*’ is avoided as this term is not widely used in the software industry. The QA team in most companies is responsible for managing the release testing process. This means that they manage the testing of the software before it is released to customers. They are responsible for checking that the system tests provide coverage of the requirements and that proper records are maintained of the testing process.

**Quality management** provides an independent check on the software development process. The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals, as shown in Figure 9.1. The QA team should be independent from the development team so that they can take an objective view of the software. This allows them to report on software quality without being influenced by software development issues.

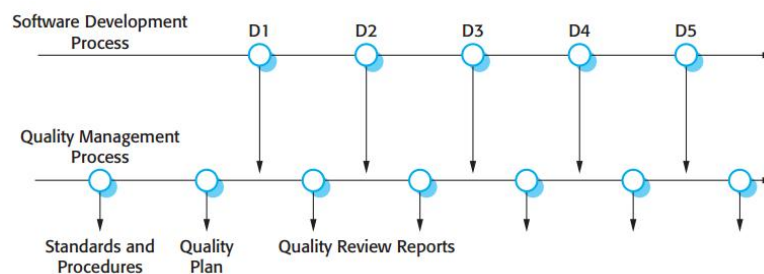


Figure 9.1: Quality management and software development [1]

Ideally, **the quality management team** should not be associated with any particular development group, but should rather have organization-wide responsibility for quality management. They should be independent and report to management above the project manager level. The reason for this is that project managers have to maintain the project budget and schedule. If problems arise, they may be tempted to compromise on product quality so that they meet their schedule. An independent quality management team ensures that the organizational goals of quality are not compromised by short-term budget and schedule considerations. In smaller companies, however, this is practically impossible. Quality

management and software development are inevitably intertwined with people having both development and quality responsibilities.

**Quality planning** is the process of developing a quality plan for a project. The quality plan should set out the desired software qualities and describe how these are to be assessed. It therefore defines what ‘high-quality’ software actually means for a particular system. Without this definition, engineers may make different and sometimes conflicting assumptions about which product attributes reflect the most important quality characteristics. Formalized quality planning is an integral part of plan-based development processes. Agile methods, however, adopt a less formal approach to quality management.

An outline structure for a quality plan includes:

1. **Product introduction:** A description of the product, its intended market, and the quality expectations for the product.
2. **Product plans:** The critical release dates and responsibilities for the product, along with plans for distribution and product servicing
3. **Process descriptions:** The development and service processes and standards that should be used for product development and management.
4. **Quality goals:** The quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. **Risks and risk management:** The key risks that might affect product quality and the actions to be taken to address these risks.

Quality plans, which are developed as part of the general project planning process, differ in detail depending on the size and the type of system that is being developed. However, when writing quality plans, you should try to keep them as short as possible. If the document is too long, people will not read it and this will defeat the purpose of producing the quality plan.

Some people think that software quality can be achieved through prescriptive processes that are based around organizational standards and associated quality procedures that check that these standards are followed by the software development team. Their argument is that standards embody good software engineering practice and that following this good practice will lead to high-quality products. In practice, however, there is much more to quality management than standards and the associated bureaucracy to ensure that these have been followed.

Standards and processes are important but quality managers should also aim to develop a ‘quality culture’ where everyone responsible for software development is committed to achieving a high level of product quality. They should encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement. Although standards and procedures are the basis of quality management, good quality managers recognize that there are intangible aspects to software quality (elegance, readability, etc.) that cannot be embodied in standards. They should support people who are interested in the intangible aspects of quality and encourage professional behavior in all team members.

Formalized quality management is particularly important for teams that are developing large, long-lifetime systems that take several years to develop. Quality documentation is a record of what has been done by each subgroup in the project. It helps people check that important tasks have not been forgotten

or that one group has not made incorrect assumptions about what other groups have done. The quality documentation is also a means of communication over the lifetime of a system. It allows the groups responsible for system evolution to trace the tests and checks that have been implemented by the development team.

For smaller systems, quality management is still important but a more informal approach can be adopted. Not as much paperwork is needed because a small development team can communicate informally. The key quality issue for small systems development is establishing a quality culture and ensuring that all team members have a positive approach to software quality.

### 9.1. Software Quality

The fundamentals of quality management were established by manufacturing industry in a drive to improve the quality of the products that were being made. As part of this, they developed a definition of *'quality'*, which was based on conformance with a detailed product specification and the notion of *tolerances*. The underlying assumption was that products could be completely specified and procedures could be established that could check a manufactured product against its specification. Of course, products will never exactly meet a specification so some tolerance was allowed. If the product was 'almost right', it was classed as acceptable.

*Software quality is not directly comparable with quality in manufacturing.* The idea of tolerances is not applicable to digital systems and, for the following reasons, it may be impossible to come to an objective conclusion about whether or not a software system meets its specification:

1. It is difficult to write complete and unambiguous software specifications. Software developers and customers may interpret the requirements in different ways and it may be impossible to reach agreement on whether or not software conforms to its specification.
2. Specifications usually integrate requirements from several classes of stakeholders. These requirements are inevitably a compromise and may not include the requirements of all stakeholder groups. The excluded stakeholders may therefore perceive the system as a poor quality system, even though it implements the agreed requirements.
3. It is impossible to measure certain quality characteristics (e.g., maintainability) directly and so they cannot be specified in an unambiguous way.

Because of these problems, the assessment of software quality is a subjective process where the quality management team has to use their judgment to decide if an acceptable level of quality has been achieved. The quality management team has to consider whether or not the software is fit for its intended purpose. This involves answering questions about the system's characteristics. For example:

1. *Have programming and documentation standards been followed in the development process?*
2. *Has the software been properly tested?*
3. *Is the software sufficiently dependable to be put into use?*
4. *Is the performance of the software acceptable for normal use?*
5. *Is the software usable?*
6. *Is the software well structured and understandable?*

There is a general assumption in software quality management that the system will be tested against its requirements. The judgment on whether or not it delivers the required functionality should be based on the results of these tests. Therefore, the QA team should review the tests that have been developed and examine the test records to check that testing has been properly carried out. In some organizations, the quality management team is responsible for system testing but, sometimes, a separate system testing group is made responsible for this.

The subjective quality of a software system is largely based on its non-functional characteristics. This reflects practical user experience—if the software’s functionality is not what is expected, then users will often just work around this and find other ways to do what they want to do. However, if the software is unreliable or too slow, then it is practically impossible for them to achieve their goals.

Therefore software quality is not just about whether the software functionality has been correctly implemented, but also depends on non-functional system attributes. The 15 important software quality attributes are shown in Table 9.1. These attributes relate to the software dependability, usability, efficiency, and maintainability. Dependability attributes are usually the most important quality attributes of a system. However, the software’s performance is also very important. Users will reject software that is too slow.

|             |                   |              |             |              |
|-------------|-------------------|--------------|-------------|--------------|
| Safety      | Resilience        | Testability  | Complexity  | Reusability  |
| Security    | Robustness        | Adaptability | Portability | Efficiency   |
| Reliability | Understandability | Modularity   | Usability   | Learnability |

Table 9.1: Software quality attributes [1]

It is not possible for any system to be optimized for all of these attributes, for example, improving robustness may lead to loss of performance. The quality plan should therefore define the most important quality attributes for the software that is being developed. It may be that efficiency is critical and other factors have to be sacrificed to achieve this. If you have stated this in the quality plan, the engineers working on the development can cooperate to achieve this. The plan should also include a definition of the quality assessment process. This should be an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

An assumption that underlies software quality management is that the quality of software is directly related to the quality of the software development process. This again comes from manufacturing systems where product quality is intimately related to the production process. *A manufacturing process involves configuring, setting up, and operating the machines involved in the process.* Once the machines are operating correctly, product quality naturally follows. You measure the quality of the product and change the process until you achieve the quality level that you need. Figure 9.2 illustrates this process-based approach to achieving product quality.

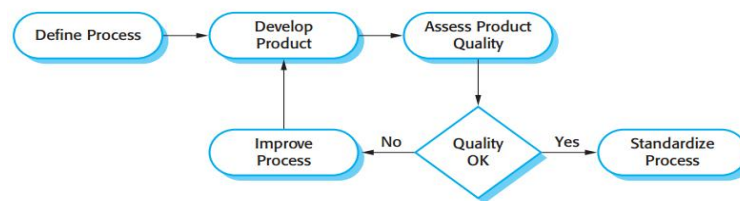


Figure 9.2: Process based quality [1]



There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor. Once manufacturing systems are calibrated, they can be run again and again to output high-quality products. However, software is not manufactured—it is designed. In software development, therefore, the relationship between process quality and product quality is more complex. Software development is a creative rather than a mechanical process, so the influence of individual skills and experience is significant. External factors, *such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.*

There is no doubt that the development process used has a significant influence on the quality of the software and that good processes are more likely to lead to good quality software. Process quality management and improvement can lead to fewer defects in the software being developed. However, it is difficult to assess software quality attributes, such as maintainability, without using the software for a long period. Consequently, it is hard to tell how process characteristics influence these attributes. Furthermore, because of the role of design and creativity in the software process, process standardization can sometimes stifle creativity, which leads to poorer rather than better quality software.

## 9.2. Software Quality Assurance

Quality assurance is the process of defining how software quality can be achieved and how the development organization knows that the software has the required level of quality. The QA process is primarily concerned with defining or selecting standards that should be applied to the software development processor software product. As part of the QA process, you may select and procure tools and methods to support these standards.

The two types of standards that may be established as part of the quality assurance process are:

1. **Product standards:** These standards apply to the software product being developed. They include document standards, such as the structure of requirements documents; documentation standards, such as a standard comment header for an object class definition; and coding standards that define how a programming language should be used.
2. **Process standards:** These standards define the processes that should be followed during software development. They may include definitions of specification, design and validation processes and a description of the documents that should be written in the course of these processes.

There is a close link between product and process standards. Product standards apply to the output of the software process and, in many cases; process standards include specific process activities that ensure that product standards are followed.

Software standards are important for several reasons:

- They are based on knowledge about the best or most appropriate practice for the company. This knowledge is often only acquired after a great deal of trial and error. Building it into a standard helps the company avoid repeating past mistakes. Standards capture wisdom that is of value to the organization.



- They provide a framework for implementing the quality assurance process. Given that standards encapsulate best practice, quality assurance involves ensuring that appropriate standards have been selected and are used.
- They assist in continuity where work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices. Consequently, learning effort when starting new work is reduced.

The development of software engineering project standards is a difficult and time consuming process. National and international bodies such as the USDoD, ANSI, BSI, NATO and the IEEE have been active in the production of standards. These are general standards that can be applied across a range of projects. Bodies such as NATO and other defense organizations may require that their own standards are followed in software contracts.

National and international standards have been developed covering software engineering terminology, programming languages such as Java and C++, notations such as charting symbols, procedures for deriving and writing software requirements, quality assurance procedures, and software verification and validation processes.

Quality assurance teams that are developing standards for a company should normally base their organizational standards on national and international standards. Using these standards as a starting point, the quality assurance team should draw up a standards 'handbook'. This should define the standards that are needed by their organization. Examples of standards that might be included in such a handbook are shown in Table 9.2.

Software engineers sometimes consider standards to be bureaucratic and irrelevant to the technical activity of software development. This is particularly likely when the standards require tedious form filling and work recording. Although they usually agree about the general need for standards, engineers often find good reasons why standards are not necessarily appropriate to their particular project.

| Product standards               | Process standards             |
|---------------------------------|-------------------------------|
| Design review form              | Design review conduct         |
| Requirements document structure | Submission of documents to CM |
| Method header format            | Version release process       |
| Java programming style          | Project plan approval process |
| Project plan format             | Change control process        |
| Change request form             | Test recording process        |

Table 9.2: Product and process standards [1]

To avoid these problems, quality managers who set the standards need to be adequately resourced and should take the following steps:

- ***Involve software engineers in the selection of product standards.*** They should understand why standards have been designed and so are more likely to be committed to these standards. The standards document should not simply state a standard to be followed but should include a rationale of why particular standardization decisions have been made.
- ***Review and modify standards regularly to reflect changing technologies.*** Once standards are developed, they tend to be enshrined in a company standards handbook, and management is often

reluctant to change them. A standards handbook is essential but it should evolve to reflect changing circumstances and technology.

- ***Provide software tools to support standards wherever possible.*** Clerical standards are the cause of many complaints because of the tedious work involved in implementing them. If tool support is available, you don't need much extra effort to follow the software development standards.

Process standards may cause difficulties if an impractical process is imposed on the development team. Different types of software need different development processes. There is no point in prescribing a particular way of working if it is inappropriate for a project or project team. Each project manager should therefore have the authority to modify process standards according to individual circumstances. However, standards that relate to product quality and the post-delivery process should be changed only after careful consideration.

The project manager and the quality manager can avoid the problems of inappropriate standards by careful quality planning early in the project. They should decide which of the standards in the handbook should be used without change, which should be modified and which should be ignored. New standards may have to be created in response to a particular project requirement. For example, standards for formal specifications may be required if these have not been used in previous projects. As the team gains experience with them, you should plan to modify and extend these new standards.

### 9.3. Software Review (9.2) Formal Technical Review

Reviews are the most widely used method of validating the quality of a process or product. This involves examining the software, its documentation and records of the process to discover errors and omissions and to see if quality standards have been followed. The conclusions of the review are formally recorded and passed to the author or whoever is responsible for correcting the discovered problems. Table 9.3 briefly describes several types of review, including reviews for quality management.

| Review type                   | Principle purpose  |
|-------------------------------|--|
| Design or program inspections | To detect detailed errors in the requirements, design or code. A checklist of possible errors should drive the review.   |
| Progress reviews              | To provide information for management about the overall progress of the project. This is both a process and a product review and is concerned with costs, plans and schedules.   |
| Quality reviews               | To carry out a technical analysis of product components or documentation to find mismatches between the specification and the component design, code or documentation and to ensure that defined quality standards have been followed. |

Table 9.3: Types of review [1]

During a review, a group of people examine the software and its associated documentation, looking for potential problems and non-conformance with standards. The review team makes informed judgments about the level of quality of a system or project deliverable. Project managers may then use these assessments to make planning decisions and allocate resources to the development process.

The review team should have a core of three to four people who are selected as principal reviewers. One member should be a senior designer who can take the responsibility for making significant technical decisions. The principal reviewers may invite other project members, such as the designers of related sub-systems, to contribute to the review. They may not be involved in reviewing the whole document. Rather,

they concentrate on those parts that affect their work. Alternatively, the review team may circulate the document being reviewed and ask for written comments from a broad spectrum of project members.

Documents to be reviewed must be distributed well in advance of the review to allow reviewers time to read and understand them. Although this delay can disrupt the development process, reviewing is ineffective if the review team have not properly understood the documents before the review takes place.

The review itself should be relatively short (two hours at most). The author of the document being reviewed should '*walkthrough*' the document with the review team. One team member should chair the review and another should formally record all review decisions and actions to be taken. During the review, the chair is responsible for ensuring that all written comments are considered. The review chair should sign a record of comments and actions agreed upon during the review. This record is then filed as part of the formal project documentation. If only minor problems are discovered, a further review may be unnecessary. The chairman is responsible for ensuring that the required changes are made. If major changes are necessary, a follow-on review may be arranged.

Quality reviews are based on documents that have been produced during the software development process. As well as software specifications, designs, or code, process models, test plans, configuration management procedures, process standards, and user manuals may all be reviewed. The review should check the consistency and completeness of the documents or code under review and make sure that quality standards have been followed.

However, reviews are not just about checking conformance to standards. They are also used to help discover problems and omissions in the software or project documentation. The conclusions of the review should be formally recorded as part of the quality management process. If problems have been discovered, the reviewers' comments should be passed to the author of the software or whoever is responsible for correcting errors or omissions.

The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team. Reviewing is a public process of error detection, compared with the more private component-testing process. Inevitably, mistakes that are made by individuals are revealed to the whole programming team. To ensure that all developers engage constructively with the review process, project managers have to be sensitive to individual concerns. They must develop a working culture that provides support without blame when errors are discovered.

Although a quality review provides information for management about the software being developed, quality reviews are not the same as management progress reviews. Progress reviews compare the actual progress in a software project against the planned progress. Their prime concern is whether or not the project will deliver useful software on time and on budget. Progress reviews take external factors into account, and changed circumstances may mean that software under development is no longer required or has to be radically changed. Projects that have developed high-quality software may have to be canceled because of changes to the business or its operating environment

## THE REVIEW PROCESS

Although there are many variations in the details of reviews, the review process (Figure 9.3) is normally structured into three phases:

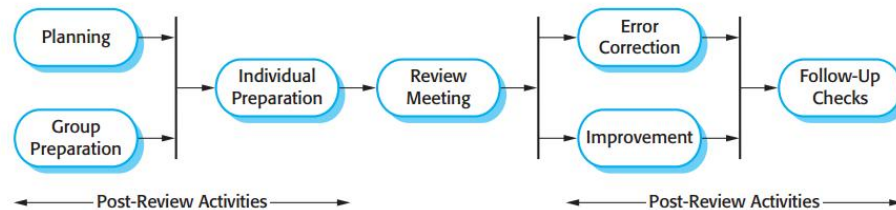


Figure 9.3: The software review process [1]

1. **Pre-review activities:** These are preparatory activities that are essential for the review to be effective. Typically, pre-review activities are concerned with review planning and review preparation. Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed. During review preparation, the team may meet to get an overview of the software to be reviewed. Individual review team members read and understand the software or documents and relevant standards. They work independently to find errors, omissions, and departures from standards. Reviewers may supply written comments on the software if they cannot attend the review meeting.
2. **The review meeting:** During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team. The review itself should be relatively short—two hours at most. One team member should chair the review and another should formally record all review decisions and actions to be taken. During the review, the chair is responsible for ensuring that all written comments are considered. The review chair should sign a record of comments and actions agreed during the review.
3. **Post-review activities:** After a review meeting has finished, the issues and problems raised during the review must be addressed. This may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents. Sometimes, the problems discovered in a quality review are such that a management review is also necessary to decide if more resources should be made available to correct them. After changes have been made, the review chair may check that the review comments have all been taken into account. Sometimes, a further review will be required to check that the changes made cover all of the previous review comments

Review teams should normally have a core of three to four people who are selected as principal reviewers. One member should be a senior designer who will take the responsibility for making significant technical decisions. The principal reviewers may invite other project members, such as the designers of related subsystems, to contribute to the review. They may not be involved in reviewing the whole document but should concentrate on those sections that affect their work.

Alternatively, the review team may circulate the document and ask for written comments from a broad spectrum of project members. The project manager need not be involved in the review, unless problems are anticipated that require changes to the project plan.

The above review process relies on all members of a development team being co-located and available for a team meeting. However, project teams are now often distributed, sometimes across countries or continents, so it is often impractical for team members to meet in the same room. In such situations, document editing tools may be used to support the review process. Team members use these to annotate the document or software source code with comments. These comments are visible to other team members who may then approve or reject them. A phone discussion may only be required when disagreements between reviewers have to be resolved.

The review process in agile software development is usually informal. In Scrum, for example, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed. In extreme programming, pair programming ensures that code is constantly being examined and reviewed by another team member.

General quality issues are also considered at daily team meetings but XP relies on individuals taking the initiative to improve and refactor code. Agile approaches are not usually standards-driven, so issues of standards compliance are not usually considered.

The lack of formal quality procedures in agile methods means that there can be problems in using agile approaches in companies that have developed detailed quality management procedures. Quality reviews can slowdown the pace of software development and they are best used within a plan-driven development process. In a plan-driven process, reviews can be planned and other work scheduled in parallel with them. This is impractical in agile approaches that focus single-mindedly on code development.

#### **9.4. Formal Approaches to SQA (9.5) Statistical Software Quality Assurance**

Software Quality Assurance (SQA) is 'a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements' (ANSI/IEEE Std 730.1-1989). Software Quality Assurance is synonymous with Software 'Product Assurance' (PA) and the terms are used interchangeably in these Standards.

The SQA perspective of a review report is twofold. First, SQA must ensure that all action items on the review report are discussed. SQA should also be concerned with analyzing the data on the review forms and classifying defects to improve the software development and review processes. Many possible classifications of defects exist. There is also a variety of interpretations that can be made from accurate review data reporting that must be discussed. For example, a high number of specification errors may suggest a lack of rigor or time in the requirements specifications phase of the project. Information regarding the type of review, its participants, its agenda and its scheduling should also be recorded in order to facilitate improved review planning activities. In essence, the information review report should be utilized to evaluate both the software and its development process. This is most often performed during some sort of post mortem review of a project.

- Assumes that a rigorous syntax and semantics can be defined for every programming language
- Allows the use of a rigorous approach to the specification of software requirements
- Applies mathematical proof of correctness techniques to demonstrate that a program conforms to its specification

### ***Proof of Correctness***

- Treat a program as a mathematical object.
- Developed with a language with a rigorous syntax.
- Are attempts at developing a rigorous approach to specification of software requirements?
- With both can attempt to develop a mathematical proof that a program conforms exactly to its specification.
- In the code, can at selected statements formulate assertions on the set of correct values for program variables.
- Can then show that the statements between these assertions do the correct transformation of the values in the assertions.

### ***Statistical quality assurance:***

- Information about software defects is collected and categorized.
- An attempt is made to trace each defect to its underlying cause (e.g., not conforming to the specification, design error, violation of standards, and poor communication with customer ...)
- Using the 'Pareto principle' (80% of defects can be traced to 20% of all possible causes), isolate the 20% of causes (the "vital few").
- Once the "vital few" causes have been identified, correct the problems that have caused the defects.

### ***The Cleanroom Process (See also Cleanroom Software Development in Chapter 7.3)***

- Use statistical quality control and formal program verification.
- Attempt to prevent defects rather than find defects.
- In projects attempted so far with this method (size between 1000 and 50,000 LOC), 90% of all defects were found before the first execution tests were conducted.
- Has not been widely applied in industry.
- Requires significant change in both management and technical approaches to software development.

## **9.6. Software Reliability**

Software Reliability is the application of statistical techniques to data collected during system development and operation to specify, predict, estimate, and assess the reliability of software-based systems. Software Reliability Engineering (SRE) is a standard, proven best practice that makes testing more reliable, faster, and cheaper. It can be applied to any system using software and to frequently-used members of software component libraries."

In measuring reliability of software the old timers applied the principles of measuring hardware reliability. They transplanted the hardware reliability measurement standards on to the software. But that was a mismatch. Though a mismatch, there is a good deal of discussion still going on about the applicability of hardware reliability techniques to software. Hardware failures are prompted more by wear and tear and the impact of environmental factors like heat, dust etc. and less by design faults. The obverse is true in the case of software

It has been stated that in a computer system the measure of reliability is MTBF and

$$MTBF = MTTF + MTTR$$

Where MTBF = Mean Time between Failure

MTTF = Mean Time to Failure

MTTR = Mean Time to Repair

A point worth noting in the field of software reliability is that a software program may contain within its ambit several errors. The end-user is not interested in the total count of the number of errors and cannot feel satisfied because the number of errors say is only 5. Each time he runs the program if the error produces failure the utility is nil. In other words each error does not produce the same failure rate. Some errors may remain dormant because that part of the program containing errors is not used quite often and hence looks to the user to be failure free. Only on the rare occasions when it is used it will end up in failure. The existence of errors and the rate of failure therefore do not carry a pro-rata link. Such errors may take longer MTBF to surface; perhaps several decades.

Consider that an error of this type triggers a failure once in 30 years. You are moving heaven and earth to eradicate all these types of errors. Obviously that will not improve MTBF in any sizable way.

Is the software available when you want it? Or is it failing when you need it badly? The measure of availability is stated to be

$$\text{Availability} = [MTTF / (MTTF + MTTR)] \times 100\%$$

This supplies us with the information that if the MTTR is low the software can be maintained by consuming little repair time – the maintainability is easy

## 9.7. A Framework for Software Metrics

Software metrics is a measure of some property of a piece of software or its specifications. Good quality, reliability and maintainability are important attributes of enterprise applications and have a huge impact in the success on the economics of the business powered. Some of the reasons why we have software product metrics are because they:

- Help software engineers to better understand the attributes of models and assess the quality of the software
- Help software engineers to gain insight into the design and construction of the software
- Focus on specific attributes of software engineering work products resulting from analysis, design, coding, and testing
- Provide a systematic way to assess quality based on a set of clearly defined rules
- Provide an “on-the-spot” rather than “after-the-fact” insight into the software development

In framework for software Metrics, Measures, Measurement, Metrics and Indicators are often used interchangeably but they can have subtle differences:

- **Measures:** Provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attributes of a product or process.
- **Measurement:** The act of determining a measure.
- **Metric:** A quantitative measure of the degree to which a system, component or process processes a given attribute.
- **Indicator:** A metric or combination of metrics that provide insight into the software process, a software project or the product itself.

The purposes of software product metrics are:

- Aid in the evaluation of analysis and design models
- Provide an indication of the complexity of procedural designs and source code
- Facilitate the design of more effective testing techniques
- Assess the stability of a fielded software product

Activities of a *Measurement Process* are:

- **Formation:** The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
- **Collection:** The mechanism used to accumulate data required to derive the formulated metrics.
- **Analysis:** The computation of metrics and application of mathematical tools.
- **Interpretation:** The evaluation of metrics in an effort to gain insight into the quality of the representation.
- **Feedback:** Recommendations derived from the interpretation of product metrics and passed on to the software development team.

Characterizing and validating metrics

- A metric should have desirable mathematical properties
  - It should have a meaningful range (e.g., zero to ten)
  - It should not be set on a rational scale if it is composed of components measured on an ordinal scale
- If a metric represents software characteristic that increases when positive traits occur or decreases when undesirable traits are encountered, the value of the metric should increase or decrease in the same manner
- Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions
  - It should measure the factor of interest independently of other factors
  - It should scale up to large systems
  - It should work in a variety of programming languages and system domains

Data collection and analysis guidelines for software product metrics

- Whenever possible, data collection and analysis should be automated



- Valid statistical techniques should be applied to establish relationships between internal product attributes and external quality characteristics
- Interpretative guidelines and recommendations should be established for each metric

## GOAL-ORIENTED SOFTWARE MEASUREMENT

- Goal/Question/Metric (GQM) paradigm
- GQM technique identifies meaningful metrics for any part of the software process
- GQM emphasizes the need to
  - Establish an explicit measurement goal that is specific to the process activity or product characteristic that is to be assessed
  - Define a set of questions that must be answered in order to achieve the goal
  - Identify well-formulated metrics that help to answer these questions
- GQM utilizes a goal definition template to define each measurement goal
- Example use of goal definition template
 

*Analyze the SafeHome software architecture for the purpose of evaluating architecture components. Do this with respect to the ability to make SafeHome more extensible from the viewpoint of the software engineers, who are performing the work in the context of product enhancement over the next three years.*
- Example questions for this goal definition
  - Are architectural components characterized in a manner that compartmentalizes function and related data?
  - Is the complexity of each component within bounds that will facilitate modification and extension?

## ATTRIBUTES OF EFFECTIVE SOFTWARE METRICS

- ***Simple and computable:*** It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time
- ***Empirically and intuitively persuasive:*** The metric should satisfy the engineer's intuitive notions about the product attribute under consideration
- ***Consistent and objective:*** The metric should always yield results that are unambiguous
- ***Consistent in the use of units and dimensions:*** The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units
- ***Programming language independent:*** Metrics should be based on the analysis model, the design model, or the structure of the program itself
- ***An effective mechanism for high-quality feedback:*** The metric should lead to a higher-quality end product

### 9.8. Metrics for Analysis and Design Model

#### METRICS FOR THE ANALYSIS

The metrics for the analysis are:

- **Functionality delivered:** It provides an indirect measure of the functionality that is packaged within the software.
- **System Size:** It measures the overall size of the system defined in terms of information available as part of the analysis model.
- **Specification quality:** It provides an indication of the specific and completeness of a requirements specification.

An example of metric for the analysis is “**Function Points**”.

### ***FUNCTION POINTS***

Function points can be used effectively as a means for measuring the functionality delivered by a system. Using historical data, function points can be used to:

- Estimate the cost or effort required to design, code, and test the software
- Predict the number of errors that will be encountered during testing
- Forecast the number of components and/or the number of projected source code lines in the implemented system

It is derived using an empirical relationship based on

1. Countable (direct) measures of the software’s *information domain*
2. Assessments of the software’s complexity

### ***Information Domain Values***

- Number of external inputs
  - Each external input originates from a user or is transmitted from another application
  - They provide distinct application-oriented data or control information
  - They are often used to update internal logical files
  - They are not inquiries (those are counted under another category)
- Number of external outputs
  - Each external output is derived within the application and provides information to the user
  - This refers to reports, screens, error messages, etc.
  - Individual data items within a report or screen are not counted separately
- Number of external inquiries
  - An external inquiry is defined as an online input that results in the generation of some immediate software response
  - The response is in the form of an on-line output
- Number of internal logical files
  - Each internal logical file is a logical grouping of data that resides within the application’s boundary and is maintained via external inputs
- Number of external interface files
  - Each external interface file is a logical grouping of data that resides external to the application but provides data that may be of use to the application

### Function Points Computation

1. Identify/collect the information domain values
2. Complete the table shown below to get the count total
  - a. Associate a weighting factor (i.e., complexity value) with each count based on criteria established by the software development organization
3. Evaluate and sum up the adjustment factors (see the next two slides)
  - a. "Fi" refers to 14 value adjustment factors, with each ranging in value from 0 (not important) to 5 (absolutely essential)
4. Compute the number of function points (FP)
  - a.  $FP = \text{count total} * [0.65 + 0.01 * \text{sum}(F_i)]$

Figure 9.4 illustrates how function points are calculated from information domain values.

| Information<br>Domain Value | Weighting Factor |   |        |         |           |
|-----------------------------|------------------|---|--------|---------|-----------|
|                             | Count            |   | Simple | Average | Complex   |
| External Inputs             | 3                | x | 3      | 4       | 6 = 9     |
| External Outputs            | 2                | x | 4      | 5       | 7 = 8     |
| External Inquiries          | 2                | x | 3      | 4       | 6 = 6     |
| Internal Logical Files      | 1                | x | 7      | 10      | 15 = 7    |
| External Interface Files    | 4                | x | 5      | 7       | 10 = 20   |
| <b>Count total</b>          |                  |   |        |         | <b>50</b> |

- $FP = \text{count total} * [0.65 + 0.01 * \text{sum}(F_i)]$
- $FP = 50 * [0.65 + (0.01 * 46)]$
- $FP = 55.5$  (rounded up to 56)

Figure 9.4: Example of function points computation

### Value Adjustment Factors

- Does the system require reliable backup and recovery?
- Are specialized data communications required to transfer information to or from the application?
- Are there distributed processing functions?
- Is performance critical?
- Will the system run in an existing, heavily utilized operational environment?
- Does the system require on-line data entry?
- Does the on-line data entry require the input transaction to be built over multiple screens or operations?
- Are the internal logical files updated on-line?
- Are the inputs, outputs, files, or inquiries complex?
- Is the internal processing complex?
- Is the code designed to be reusable?
- Are conversion and installation included in the design?
- Is the system designed for multiple installations in different organizations?
- Is the application designed to facilitate change and for ease of use by the user?

### ***Interpretation of the Function Points Number***

- Assume that past project data for a software development group indicates that
  - One FP translates into 60 lines of object-oriented source code
  - 12 FPs are produced for each person-month of effort
  - An average of three errors per function point are found during analysis and design reviews
  - An average of four errors per function point are found during unit and integration testing
- This data can help project managers revise their earlier estimates
- This data can also help software engineers estimate the overall implementation size of their code and assess the completeness of their review and testing activities

### **METRICS FOR DESIGN MODEL:**

- ***Architectural metrics:*** Provide an indication of the quality of the architectural design.
- ***Component-level metrics:*** It measures the complexity of software components and other characteristics that have a bearing on quality.
- ***Interface design metrics:*** It focuses primarily on usability.
- ***Specialized object oriented design metrics:*** It measures characteristics of classes and their communication and collaboration characteristics.

### ***HIERARCHICAL ARCHITECTURE METRICS***

- Fan out: the number of modules immediately subordinate to the module  $i$ , that is, the number of modules directly invoked by module  $i$
- Structural complexity
  - $S(i) = f_{out}^2(i)$ , where  $f_{out}(i)$  is the “fan out” of module  $i$
- Data complexity
  - $D(i) = v(i) / [f_{out}(i) + 1]$ , where  $v(i)$  is the number of input and output variables that are passed to and from module  $i$
- System complexity
  - $C(i) = S(i) + D(i)$
- As each of these complexity values increases, the overall architectural complexity of the system also increases
- This leads to greater likelihood that the integration and testing effort will also increase
- Shape complexity
  - $size = n + a$ , where  $n$  is the number of nodes and  $a$  is the number of arcs
  - Allows different program software architectures to be compared in a straightforward manner
- Connectivity density (i.e., the arc-to-node ratio)
  - $r = a/n$
  - May provide a simple indication of the coupling in the software architecture

## 9.9. ISO Standards

There is an international set of standards that can be used in the development of quality management systems in all industries, called ISO 9000. ISO 9000 standards can be applied to a range of organizations from manufacturing through to service industries. ISO 9001, the most general of these standards, applies to organizations that design, develop, and maintain products, including software. The ISO 9001 standard was originally developed in 1987, with its most recent revision in 2008.

*The ISO 9001 standard is not itself a standard for software development but is a framework for developing software standards.* It sets out general quality principles, describes quality processes in general, and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

The major revision of the ISO 9001 standard in 2000 reoriented the standard around nine core processes (Figure 9.5). If an organization is to be ISO 9001 conformant, it must document how its processes relate to these core processes. It must also define and maintain records that demonstrate that the defined organizational processes have been followed. The company quality manual should describe the relevant processes and the process data that has to be collected and maintained.

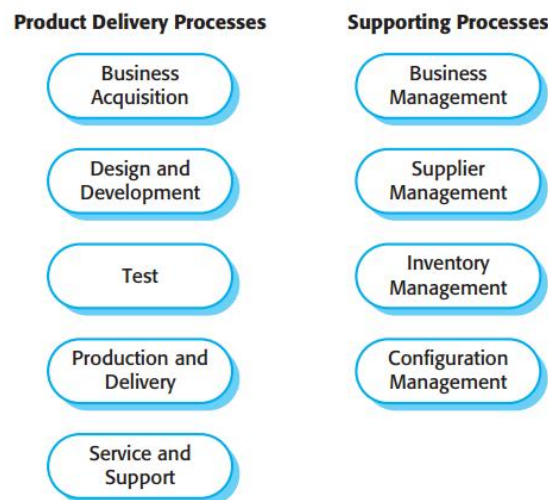


Figure 9.5: ISO 9001 core processes [1]

The ISO 9001 standard does not define or prescribe the specific quality processes that should be used in a company. To be conformant with ISO 9001, a company must have defined the types of process shown in Figure 9.5 and have procedures in place that demonstrate that its quality processes are being followed. This allows flexibility across industrial sectors and company sizes. Quality standards can be defined that are appropriate for the type of software being developed. Small companies can have unbureaucratic processes and still be ISO 9001 compliant. However, this flexibility means that you cannot make assumptions about the similarities or differences between the processes in different ISO 9001-compliant companies. Some companies may have very rigid quality processes that keep detailed records, whereas others may be much less formal, with minimal additional documentation.

The relationships between ISO 9001, organizational quality manuals, and individual project quality plans are shown in Figure 9.6. This diagram explains how the general ISO 9001 standard can be used as a basis for software quality management processes.

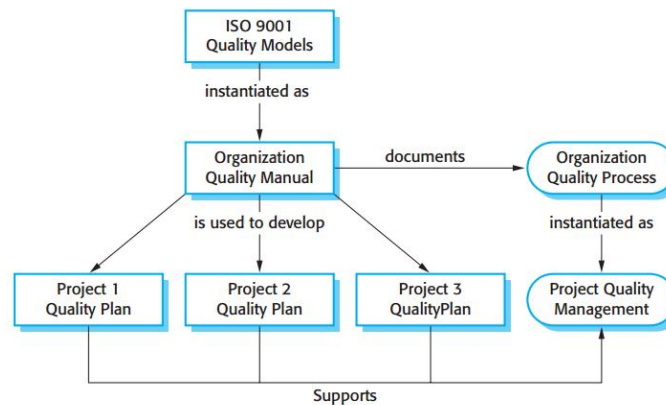


Figure 9.6: ISO 9001 and quality management [1]

Some software customers demand that their suppliers should be ISO 9001 certified. The customers can then be confident that the software development company has an approved quality management system in place. Independent accreditation authorities examine the quality management processes and process documentation and decide if these processes cover all of the areas specified in ISO 9001. If so, they certify that a company's quality processes, as defined in the quality manual, conform to the ISO 9001 standard.

Some people think that ISO 9001 certification means that the quality of the software produced by certified companies will be better than that from uncertified companies. This is not necessarily true. The ISO 9001 standard focuses on ensuring that the organization has quality management procedures in place and it follows these procedures. There is no guarantee that ISO 9001 certified companies use the best software development practices or that their processes lead to high-quality software.

For example, a company could define test coverage standards specifying that all methods in objects must be called at least once. Unfortunately, this standard can be met by incomplete software testing, which does not run tests with different method parameters. So long as the defined testing procedures were followed and records kept of the testing carried out, the company could be ISO 9001 certified. The ISO 9001 certification defines quality to be the conformance to standards, and takes no account of the quality as experienced by users of the software.

Agile methods, which avoid documentation and focus on the code being developed, have little in common with the formal quality processes that are discussed in ISO 9001. There has been some work done on reconciling these approaches, but the agile development community is fundamentally opposed to what they see as the bureaucratic overhead of standards conformance. For this reason companies that use agile development methods are rarely concerned with ISO 9001 certification.

## DOCUMENTATION STANDARDS

Documentation standards in a software project are important because documents are the only tangible way of representing the software and the software process. Standardized documents have a consistent appearance, structure and quality, and should therefore be easier to read and understand.

There are three types of documentation standards

1. **Documentation process standards:** These standards define the process that should be followed for document production.
2. **Document standards:** These standards govern the structure and presentation of documents.
3. **Document interchange standards:** These standards ensure that all electronic copies of documents are compatible.

Documentation process standards define the process used to produce documents. This means that you set out the procedures involved in document development and the software tools used for document production. You should also define checking and refinement procedures to ensure that high-quality documents are produced.

Document process quality standards must be flexible and able to cope with all types of documents. For working papers or electronic memos, there is no need for explicit quality checking. However, for formal documents—that is, those that will be used for further development or released to customers—you should use a formal quality process. Figure 9.7 is a model of one possible documentation process.

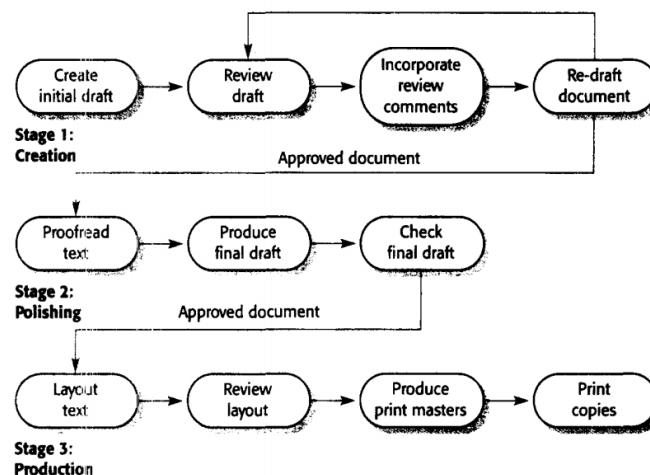


Figure 9.7: A document production process including quality checks [1]

Drafting, checking, revising and redrafting is an iterative process. It should continue until a document of acceptable quality is produced. The acceptable quality level depends on the document type and the potential readers of the document.

Document standards should apply to all documents produced during a software development project. Documents should have a consistent style and appearance, and documents of the same type should have a

consistent structure. Although document standards should be adapted to the needs of a specific project, it is good practice for the same 'house style' to be used in all of the documents produced by an organization.

Examples of document standards that may be developed are:

1. **Document identification standards:** As large system development projects may produce thousands of documents, each document should be uniquely identified. For formal documents, this identifier may be the formal identifier defined by the configuration manager. For informal documents, the project manager may define the form of the document.
2. **Document structure standards:** Each class of document produced during a software project should follow some standard structure. Structure standards should define the sections to be included and should specify the conventions used for page numbering, page header and footer information, and section and sub-section numbering.
3. **Document presentation standards:** Document presentation standards define a 'house style' for documents and contribute significantly to document consistency. They include the definition of fonts and styles used in the document, the use of logos and, company names, the use of color to highlight document structure, and so on.
4. **Document update standards:** As a document evolves to reflect changes in the system, a consistent way of indicating document changes should be used. You can use cover color to indicate document version and change bars in the margin to indicate modified or added paragraphs.

Document interchange standards are important as electronic copies of documents are interchanged. The use of interchange standards allows documents to be transferred electronically and re-created in their original form.

Assuming that the use of standard tools is mandated in the process standards, interchange standards define the conventions for using these tools. Examples of interchange standards include the use of a standard style sheet if a word processor is used or limitations on the use of document macros to avoid possible virus infection. Interchange standards may also limit the fonts and text styles used because of differing printer and display capabilities.

## 9.10. CMMI

The U.S. Software Engineering Institute (SEI) was established to improve the capabilities of the American software industry. In the mid-1980s, the SEI initiated a study of ways to assess the capabilities of software contractors. The outcome of this capability assessment was the SEI Software Capability Maturity Model (CMM). This has been tremendously influential in convincing the software engineering community to take process improvement seriously. The Software CMM was followed by a range of other capability maturity models, including the People Capability Maturity Model (P-CMM) and the Systems Engineering Capability Model.

Other organizations have also developed comparable process maturity models. The SPICE approach to capability assessment and process improvement is more flexible than the SEI model. It includes maturity levels comparable with the CMM levels, but also identifies processes, such as customer supplier processes, that cut across these levels. As the level of maturity increases, the performance of these cross-cutting processes must also improve.



The Bootstrap project in the 1990s had the goal of extending and adapting the SEI maturity model to make it applicable across a wider range of companies. This model uses the SEI's maturity levels. It also proposes a base process model (based on the model used in the European Space Agency) that may be used as a starting point for local process definition. It includes guidelines for developing a company-wide quality system to support process improvement.

In an attempt to integrate the plethora of capability models based on the notion of process maturity (including its own models), the SEI embarked on a new program to develop an integrated capability model (CMMI). The CMMI framework supersedes the Software and Systems Engineering CMMs and integrates other capability maturity models. It has two instantiations, staged and continuous, and addresses some of the reported weaknesses in the Software CMM.

The CMMI model is intended to be a framework for process improvement that has broad applicability across a range of companies. Its staged version is compatible with the Software CMM and allows an organization's system development and management processes to be assessed and assigned a maturity level from 1 to 5. Its continuous version allows for a finer-grain classification of process maturity. This model provides a way of rating 22 process areas (see Table 9.4) on a scale from 0 to 5.

| Category           | Process area   |
|--------------------|--|
| Process management | Organizational process definition (OPD)<br>Organizational process focus (OPF)<br>Organizational training (OT)<br>Organizational process performance (OPP)<br>Organizational innovation and deployment (OID)    |
| Project management | Project planning (PP)<br>Project monitoring and control (PMC)<br>Supplier agreement management (SAM)<br>Integrated project management (IPM)<br>Risk management (RSKM)<br>Quantitative project management (QPM) |
| Engineering        | Requirements management (REQM)<br>Requirements development (RD)<br>Technical solution (TS)<br>Product integration (PI)<br>Verification (VER)<br>Validation (VAL)   |
| Support            | Configuration management (CM)<br>Process and product quality management (PPQA)<br>Measurement and analysis (MA)<br>Decision analysis and resolution (DAR)<br>Causal analysis and resolution (CAR)              |

Table 9.4: Process areas in the CMMI [1]

The CMMI model is very complex, with more than 1,000 pages of description. A radically simplified version is discussed here. The principal model components are:

1. ***A set of process areas that are related to software process activities:*** The CMMI identifies 22 process areas that are relevant to software process capability and improvement. These are

organized into four groups in the continuous CMMI model. These groups and related process areas are listed in Table 9.4.

2. ***A number of goals, which are abstract descriptions of a desirable state that should be attained by an organization:*** The CMMI has specific goals that are associated with each process area and define the desirable state for that area. It also defines generic goals that are associated with the institutionalization of good practice. Table 9.5 shows examples of specific and generic goals in the CMMI.
3. ***A set of good practices, which are descriptions of ways of achieving a goal:*** Several specific and generic practices may be associated with each goal within a process area. Some examples of recommended practices are shown in Table 9.6. However, the CMMI recognizes that it is the goal rather than the way that the goal is reached that is important. Organizations may use any appropriate practices to achieve any of the CMMI goals—they do not have to adopt the practices recommended in the CMMI.

| Goal   | Process area                                   |
|--|--|
| Corrective actions are managed to closure when the project's performance or results deviate significantly from the plan. | Project monitoring and control (specific goal) |
| Actual performance and progress of the project are monitored against the project plan.                                   | Project monitoring and control (specific goal) |
| The requirements are analyzed and validated, and a definition of the required functionality is developed.                | Requirements development (specific goal)       |
| Root causes of defects and other problems are systematically determined.   | Causal analysis and resolution (specific goal) |
| The process is institutionalized as a defined process.   | Generic goal                                   |

Table 9.5: Process areas in the CMMI [1]

| Goal  | Associated practices  |
|---|---|
| The requirements are analyzed and validated, and a definition of the required functionality is developed. | Analyze derived requirements systematically to ensure that they are necessary and sufficient.   |
|   | Validate requirements to ensure that the resulting product will perform as intended in the user's environment, using multiple techniques as appropriate.          |
| Root causes of defects and other problems are systematically determined.                                  | Select the critical defects and other problems for analysis.  |
|   | Perform causal analysis of selected defects and other problems and propose actions to address them.   |
| The process is institutionalized as a defined process.  | Establish and maintain an organizational policy for planning and performing the requirements development process.   |
|   | Assign responsibility and authority for performing the process, developing the work products, and providing the services of the requirements development process. |

Table 9.6: Goals and associated practices in the CMMI [1]

Generic goals and practices are not technical but are associated with the institutionalization of good practice. What this means depends on the maturity of the organization. At an early stage of maturity

development, institutionalization may mean ensuring that plans are established and processes are defined for all software development in the company. However, for an organization with more mature, advanced processes, institutionalization may mean introducing process control using statistical and other quantitative techniques across the organization.

A CMMI assessment involves examining the processes in an organization and rating these processes or process areas on a six-point scale that relates to the level of maturity in each process area. The idea is that the more mature a process, the better it is. The six-point scale assigns a level of maturity to a process area as follows:

1. ***Incomplete***: At least one of the specific goals associated with the process area is not satisfied. There are no generic goals at this level as institutionalization of an incomplete process does not make sense.
2. ***Performed***: The goals associated with the process area are satisfied, and for all processes the scope of the work to be performed is explicitly set out and communicated to the team members.
3. ***Managed***: At this level, the goals associated with the process area are met and organizational policies are in place that define when each process should be used. There must be documented project plans that define the project goals. Resource management and process monitoring procedures must be in place across the institution.
4. ***Defined***: This level focuses on organizational standardization and deployment of processes. Each project has a managed process that is adapted to the project requirements from a defined set of organizational processes. Process assets and process measurements must be collected and used for future process improvements.
5. ***Quantitatively managed***: At this level, there is an organizational responsibility to use statistical and other quantitative methods to control sub-processes; that is, collected process and product measurements must be used in process management.
6. ***Optimizing***: At this highest level, the organization must use the process and product measurements to drive process improvement. Trends must be analyzed and the processes adapted to changing business needs

This is a very simplified description of the capability levels and, to put these into practice, you need to work with more detailed descriptions. The levels are progressive, with explicit process descriptions at the lowest levels, through process standardization, to process change and improvement driven by measurements of the process and the software at the highest level. To improve its processes, a company should aim to increase the maturity level of the process groups that are relevant to its business.

## THE STAGED CMMI MODEL

The staged CMMI model is comparable with the Software Capability Maturity Model in that it provides a means to assess an organization's process capability at one of five levels, and prescribes the goals that should be achieved at each of these levels. Process improvement is achieved by implementing practices at each level, moving from the lower to the higher levels in the model.

The five levels in the staged CMMI model are shown in Figure 9.8. They correspond to capability levels 1 to 5 in the continuous model. The key difference between the staged and the continuous CMMI models is

that the staged model is used to assess the capability of the organization as a whole, whereas the continuous model measures the maturity of specific process areas within the organization.

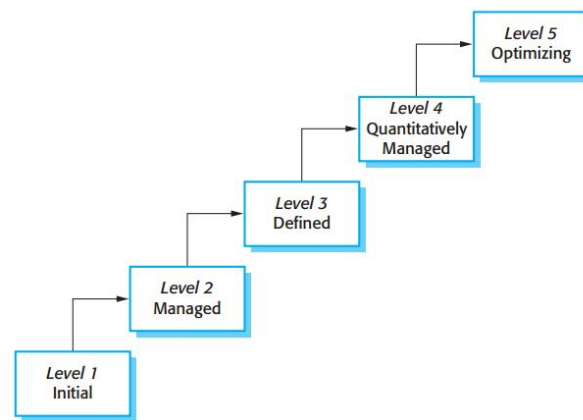


Figure 9.8: The CMMI staged maturity model [1]

Each maturity level has an associated set of process areas and generic goals. These reflect good software engineering and management practice and the institutionalization of process improvement. The lower maturity levels may be achieved by introducing good practice; however, higher levels require a commitment to process measurement and improvement.

For example, the process areas as defined in the model associated with the second level (the managed level) are:

1. **Requirements management:** Manage the requirements of the project's products and product components, and identify inconsistencies between those requirements and the project's plans and work products.
2. **Project planning:** Establish and maintain plans that define project activities.
3. **Project monitoring and control:** Provide understanding into the project's progress so that appropriate corrective actions can be taken when the project's performance deviates significantly from the plan.
4. **Supplier agreement management:** Manage the acquisition of products and services from suppliers external to the project for which a formal agreement exists.
5. **Measurement and analysis:** Develop and sustain a measurement capability that is used to support management information needs.
6. **Process and product quality assurance:** Provide staff and management with objective insight into the processes and associated work products.
7. **Configuration management:** Establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits

As well as these specific practices, organizations operating at the second level in the CMMI model should have achieved the generic goal of institutionalizing each of the processes as a managed process. Examples of institutional practices associated with project planning that lead to the project planning process being a managed process are:

- Establish and maintain an organizational policy for planning and performing the project planning process.
- Provide adequate resources for performing the project management process, developing the work products, and providing the services of the process.
- Monitor and control the project planning process against the plan and take appropriate corrective action.
- Review the activities, status, and results of the project planning process with high-level management, and resolve any issues.

The **advantage** of the staged CMMI is that it is compatible with the software capability maturity model that was proposed in the late 1980s. Many companies understand and are committed to using this model for process improvement. It is therefore straightforward for them to make a transition from this to the staged CMMI model. Furthermore, the staged model defines a clear improvement pathway for organizations. They can plan to move from the second to the third level and so on.

The major **disadvantage** of the staged model (and of the Software CMM), however, is its prescriptive nature. Each maturity level has its own goals and practices. The staged model assumes that all of the goals and practices at one level are implemented before the transition to the next level. However, organizational circumstances may be such that it is more appropriate to implement goals and practices at higher levels before lower-level practices. When an organization does this, a maturity assessment will give a misleading picture of its capability.

## THE CONTINUOUS CMMI MODEL

Continuous maturity models do not classify an organization according to discrete levels. Rather, they are finer-grained models that consider individual or groups of practices and assess the use of good practice within each process group. The maturity assessment is not, therefore, a single value but a set of values showing the organization's maturity for each process or process group.

The continuous CMMI considers the process areas shown in Table 9.4 and assigns a capability assessment level from 0 to 5 (as described earlier) to each process area. Normally, organizations operate at different maturity levels for different process areas. Consequently, the result of a continuous CMMI assessment is a capability profile showing each process area and its associated capability assessment. A fragment of a capability profile that shows processes at different capability levels is shown in Figure 9.9. This shows that the level of maturity in configuration management, for example, is high, but that risk management maturity is low. A company may develop actual and target capability profiles where the target profile reflects the capability level that they would like to reach for that process area.

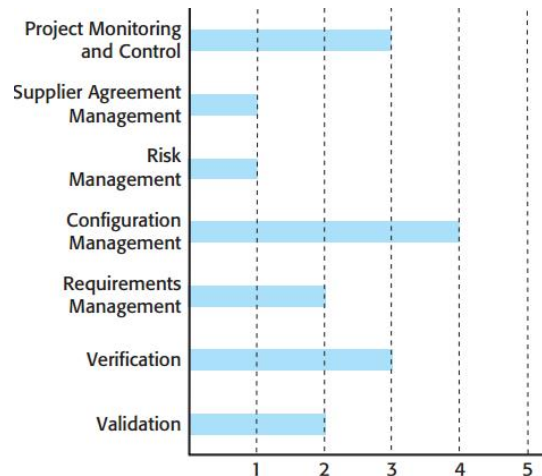


Figure 9.9: A process capability profile [1]

The principal advantage of the continuous model is that companies can pick and choose processes for improvement according to their own needs and requirements. Different types of organization have different requirements for process improvement. For example, a company that develops software for the aerospace industry may focus on improvements in system specification, configuration management, and validation, whereas a web development company may be more concerned with customer-facing processes. The staged model requires companies to focus on the different stages in turn. By contrast, the continuous CMMI permits discretion and flexibility, while still allowing companies to work within the CMMI improvement framework

### 9.11. SQA Plan

The Software Quality Assurance Plan (SQAP) defines how adherence to these standards will be monitored. The SQAP contents list is a checklist for activities that have to be carried out to assure the quality of the product. For each activity, those with responsibility for SQA should describe the plans for monitoring it.

- **Activities:** Objective evidence of adherence to these standards should be sought during all phases of the life cycle. Documents called for by this standard should be obtained and examined. Source code should be checked for adherence to coding standards. Where possible, aspects of quality (e.g. complexity, reliability, maintainability, safety, number of defects, number of problems, and number of RIDs) should be measured quantitatively, using well-established metrics. Subsequent sections list activities derived from ANSI/IEEE Std 730.1-1989 that are necessary if a software item is to be fit for its purpose. Each section discusses how the activity can be verified.
- **Management:** Analysis of the managerial structure that influences and controls the quality of the software is an SQA activity. The existence of an appropriate organizational structure should be verified. It should be confirmed that the individuals defined in that structure have defined tasks and responsibilities. The organization, tasks and responsibilities will have been defined in the Software Project Management Plan (SPMP).

- **Documentation:** The documentation plan that has been defined in the SPMP should be analyzed. Any departures from the documentation plan defined in these standards should be scrutinized and discussed with project management.
- **Standards, practices, conventions and metrics:** Adherence to all standards, practices and conventions should be monitored.
- **Reviews and audits:** These Standards call for reviews of the reports like User Requirement Document (URD), the System Requirement Document (DSRD), the Software Configuration Management Plan (SCMP), and others. It also calls for the review and audit of the code during production. The review and audit arrangements described in the Server Virtualization Validation Program (SVVP) should be examined. Many kinds of reviews are possible (e.g. technical, inspection and walkthrough). It should be verified that the review mechanisms appropriate for the type of project. SQA personnel should participate in the review process.
- **Testing activities:** Unit, integration, system and acceptance testing of executable software is essential to assure its quality. Test plans, test designs, test case, test procedures and test reports are described in the SVVP. These should be reviewed by SQA personnel. They should monitor the testing activities carried out by the development team, including test execution. Additionally, other tests may be proposed in the SQAP. These may be carried out by SQA personnel.
- **Problem reporting and corrective action:** The problem handling procedure described in these standards is designed to report and track problems from identification until solution. SQA personnel should monitor the execution of the procedures, described in the SCMP, and examine trends in problem occurrence.
- **Tools, techniques and methods:** These Standards call for a tools, techniques and methods for software production to be defined at the project level. It is an SQA activity to check that appropriate tools, techniques and methods are selected and to monitor their correct application. SQA personnel may decide that additional tools, techniques and methods are required to support their monitoring activity. These should be described in the Software Quality Assurance Plan (SQAP).
- **Code and media control:** These Standards require that the procedures for the methods and facilities used to maintain, store, secure and document controlled versions of the identified software, be defined in the SCMP. SQA personnel should check that appropriate procedures have been defined in the SCMP and carried out.
- **Supplier control:** Software items acquired from external suppliers must always be checked against the standards for the project. An SQAP shall be produced by each contractor developing the software. An SQAP is not required for commercial software.
- **Records collection, maintenance and retention:** These standards define a set of documents that must be produced in any project. Additional documents, for example minutes of meetings and review records, may also be produced. SQA personnel should check that appropriate methods and facilities are used to assemble, safeguard, and maintain all this documentation for at least the life of the project. Documentation control procedures are defined in the SCMP.
- **Training:** SQA personnel should check that development staff are properly trained for their tasks and identify any training that is necessary. Training plans are documented in the SPMP.
- **Risk management:** All projects must identify the factors that are critical to their success and control these factors. This is called 'risk management'. Project management must always analyze the risks that affect the project. Their findings are documented in the SPMP. SQA personnel

should monitor the risk management activity, and advise project management on the methods and procedures to identify, assess, monitor, and control areas of risk

### 9.12. Software Certification

The verification and testing techniques lead to software components (and entire increments) that can be certified. Within the context of the cleanroom software engineering approach, certification implies that the reliability (measured by mean-time-to-failure, MTTF) can be specified for each component.

The potential impact of certifiable software components goes far beyond a single cleanroom project. Reusable software components can be stored along with their usage scenarios, program stimuli, and probability distributions. Each component would have a certified reliability under the usage scenario and testing regime described. This information is invaluable to others who intend to use the components.

The certification approach involves five steps:

1. Usage scenarios must be created.
2. A usage profile is specified.
3. Test cases are generated from the profile.
4. Tests are executed and failure data are recorded and analyzed.
5. Reliability is computed and certified.

Steps 1 through 4 have been discussed in an earlier section. In this section, we concentrate on reliability certification.

Certification for cleanroom software engineering requires the creation of three models:

- **Sampling model:** Software testing executes  $m$  random test cases and is certified if no failures or a specified numbers of failures occur. The value of  $m$  is derived mathematically to ensure that required reliability is achieved.
- **Component model:** A system composed of  $n$  components is to be certified. The component model enables the analyst to determine the probability that component  $i$  will fail prior to completion.
- **Certification model:** The overall reliability of the system is projected and certified.

At the completion of statistical use testing, the certification team has the information required to deliver software that has a certified MTTF computed using each of these models.

## REFERENCES

[1] I. Sommerville , 2008. Software Engineering. Eighth Edition, Addison-Wesley



## 10.CONFIGURATION MANAGEMENT

Configuration management (CM) is the development and use of standards and procedures for managing an evolving software system. System requirements always change during development and use, and you have to incorporate these requirements into new versions of the system. You need to manage evolving systems because it is easy to lose track of what changes have been incorporated into what system version. Versions incorporate proposals for change, corrections of faults and adaptations for different hardware and operating systems. There may be several versions under development and in use at the same time. If you don't have effective configuration management procedures in place, you may waste effort modifying the wrong version of a system, deliver the wrong version of a system to customers or lose track of where the software source code is stored.

Configuration management procedures define how to record and process proposed system changes, how to relate these to system components and the methods used to identify different versions of the system. Configuration management tools are used to store versions of system components, build systems from these components and track the releases of system versions to customers.

Configuration management is sometimes considered to be part of software quality management, with the same manager sharing quality management and configuration management responsibilities. The software is initially released by the development team for quality assurance. The QA team checks that the system is of acceptable quality. It then becomes a controlled system, which means that changes to the system have to be agreed on and recorded before they are implemented. Controlled systems are sometimes called baselines because they are a starting point for further, controlled evolution.

There are many reasons why systems exist in different configurations. Configurations may be produced for different computers, for different operating systems, incorporating client-specific functions and soon (See Figure 10.1). Configuration managers are responsible for keeping track of the differences between software versions for ensuring that new versions are derived in a controlled way and for releasing new versions to the right customers at the right time.

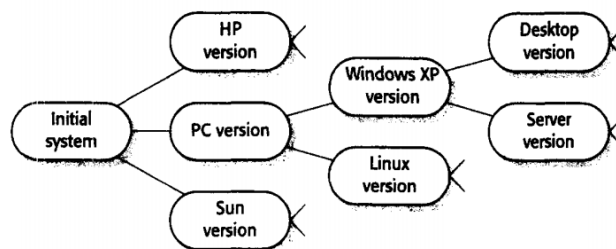


Figure 10.1: System families [1]

The definition and use of configuration management standards is essential for quality certification in both the ISO9000 and the CMM and CMMI standards. An example of such a standard is IEEE 828-1998, which is a standard for configuration management plans. Within a company, these standards should be incorporated into the quality handbook or configuration management guide. Of course, the generic external standards may be used as a basis for more detailed organizational standards that are tailored to a specific environment.

In a traditional software development process based on the 'waterfall' model, software is delivered to the configuration management team after development is complete and the individual software components have been tested. This team then takes over the responsibility for building the complete system and for managing system testing. Faults that are discovered during system testing are passed back to the development team for repair. After the faults have been repaired, the development team delivers a new version of the repaired component to the quality assurance team. If the quality is acceptable, this then may become the new baseline for further system development.

The model, where the CM team controls the system integration and testing processes, has influenced the development of configuration management standards. Most CM standards have an embedded assumption that a waterfall model will be used for system development. This means that the standards have to be adapted to modern software development approaches based on incremental specification and development. To cater for incremental development, some organizations have developed a modified approach to configuration management that supports concurrent development and system testing. This approach relies on a very frequent (at least daily) build of the whole system from its components:

- The development organization sets a delivery time (say 2 p.m.) for system components. If developers have new versions of the components that they are writing, they must deliver them by that time. Components may be incomplete but should provide some basic functionality that can be tested.
- A new version of the system is built from these components by compiling and linking them to form a complete system.
- This system is then delivered to the testing team, which carries out a set of predefined system tests. At the same time, the developers are still working on their components, adding to the functionality and repairing faults discovered in previous tests.
- Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

The advantages of using daily builds of software are that the chances of finding problems stemming from component interactions early in the process are increased. Furthermore, daily building encourages thorough unit testing of components. Psychologically, developers are put under pressure not to 'break the build', that is, deliver versions of components that cause the whole system to fail. They are therefore reluctant to deliver new component versions that have not been properly tested. Less system testing time is spent discovering and coping with software faults that should have been found during unit testing.

The successful use of daily builds requires a very stringent change management process to keep track of the problems that have been discovered and repaired. It also leads to a very large number of system and component versions that must be managed. Good configuration management is therefore essential for this approach to be successful.

Configuration management in agile and rapid development approaches cannot be based around rigid procedures and paper work. While these may be necessary for large, complex projects, they slow down the development process. Careful record-keeping is essential for large, complex systems developed across several sites, but it is unnecessary for small projects. In these projects, all team members work together in the same room, and the overhead involved in record-keeping slows down the development process. However, this does not mean that CM should be completely abandoned when rapid development is

required. Rather, agile processes use simple CM tools, such as version management and system-building tools that enforce some control. All team members have to learn to use these tools and conform to the disciplines that they impose.

### **10.1. Configuration Management Planning**

A configuration management plan describes the standards and procedures that should be used for configuration management. The starting point for developing the plan should be a set of configuration management standards, and these should be adapted to fit the requirements and constraints of each specific project. The CM plan should be organized into a number of sections that:

1. Define what is to be managed (the configuration items) and the scheme that you should use to identify these entities.
2. Set out who is responsible for the configuration management procedures and for submitting controlled entities to the configuration management team.
3. Define the configuration management policies that all team members must use for change control and version management.
4. Specify the tools that you should use for configuration management and the processes for using these tools.
5. Describe the structure of the configuration database that is used to record configuration information and the information that should be maintained in that database (the configuration records).

You may also include other information in the CM plan such as the management of software from external suppliers and the auditing procedures for the CM process in the CM plan.

An important part of the CM plan is the definition of responsibilities. The plan should define who is responsible for the delivery of each document or software component to quality assurance and configuration management. It may also define the reviewers of each document. The person responsible for document delivery need not be the same as the person responsible for producing the document. To simplify interfaces, project managers or team leaders are often responsible for all documents, produced by their team.

### **CONFIGURATION ITEM IDENTIFICATION**

In a large software system, there may be thousands of source code modules, test scripts, design documents and so on. These are produced by different people and, when created, may be assigned similar or identical names. To keep track of all this information so that the right file can be found when it is needed, you need a consistent Identification scheme for all items in the configuration management system.

During the configuration management planning process, you decide exactly which items (or classes of items) are to be controlled. Documents or groups of related documents under configuration control are formal documents or configuration items. Project plans, specifications, designs, programs and test data suites are normally maintained as configuration items. All documents that may be useful for future system evolution should be controlled by the configuration management system.

*However, this does not mean that every document or file produced must be placed under configuration control.* Documents such as technical working documents that present a snapshot of ideas for further development, minutes of group meetings, outline plans and proposals, and so on may not have long-term relevance and are not needed for future maintenance of the system.

The configuration item identification scheme must assign a unique name to all documents under configuration control. This name may reflect the type of item, the part of the system that it applies to, the creator of the item and so on. In your naming scheme, you may wish to reflect relationships between items by ensuring that related documents have a common root to their name. Therefore, you might define a hierarchical naming scheme with names such as

PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE

PCL-TOOLS/EDIT/HELP/QUERY/HELPPFRAMES/FR-I

The initial part of the name is the project name, PCL-TOOLS. In this project, there are a number of separate tools being developed, so the tool name (EDIT) is used as the next part of the name. Each tool includes differently named modules whose name makes up the next component of the item identifier (FORMS HELP). This decomposition process continues until the base-level formal documents are referenced (Figure 10.2). The leaves of the documentation hierarchy are the formal configuration items. Figure 10.2 shows that three formal items are required for each code component: an object description (OBJECTS), the source code of the component (CODE) and a set of tests for that component (TESTS). Items such as help frames are also managed and have different names (FR-I, in the preceding example).

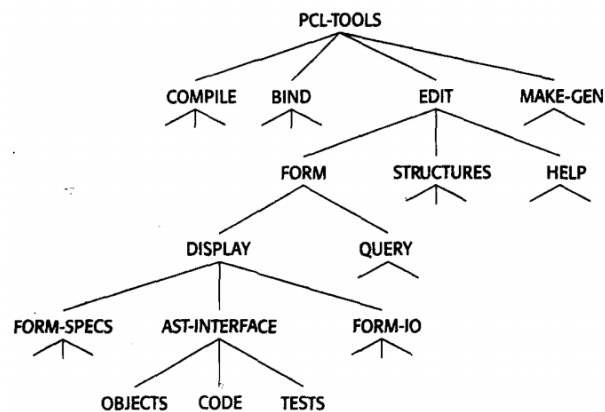


Figure 10.2: Hierarchical naming scheme [1]

Hierarchical naming schemes are simple and easily understood, and sometimes they can map to the directory structures used to store project files. However, they reflect the structure of the project where the software was developed. The configuration item names associate components with a particular project and so may reduce the opportunities for reuse. It can be very hard to find related components (e.g., all components developed by the same programmer) where the relationship is not reflected in the item-naming scheme.

## THE CONFIGURATION DATABASE

The configuration database is used to record all relevant information about system configurations and configuration items. You use the CM database to help assess the impact of system changes and to generate reports for management about the CM process. As part of the CM planning process, you should define the CM database schema, the forms to collect information to be recorded in the database and procedures for recording and retrieving project information.

A configuration database does not just include information about configuration items. It may also record information about users of components, system customers, execution platforms, proposed changes and so forth. It should be able to provide answers to a variety of queries about system configurations. Typical queries might be:

1. Which customers have taken delivery of a particular version of the system?
2. What hardware and operating system configuration is required to run a given system version?
3. How many versions of a system have been created and what were their creation dates?
4. What versions of a system might be affected if a particular component is changed?
5. How many change requests are outstanding on a particular version?
6. How many reported faults exist in a particular version?

Ideally, the configuration database should be integrated with the version management system that is used to store and manage the formal project documents. This approach, supported by some integrated CASE tools, makes it possible to link changes directly with the documents and components affected by the change. Links between documents (such as design documents) and program code may be maintained so that you can find everything that you have to modify when a change is proposed.

However, integrated CASE tools for configuration management are expensive. Many companies do not use them but maintain their configuration database separate from their version control systems. They store configuration items as files in a directory structure or in a version management system such as CVS.

The configuration database stores information about the configuration items and references their names in the version management system or file store. While this is a relatively cheap and flexible approach, the problem with it is that configuration items maybe changed without going through the configuration database. Therefore, you can't be completely sure that the configuration database is an up to-date description of the state of the system.

### 10.2. Change Management

Change is a fact of life for large software systems. Organizational needs and requirements change during the lifetime of a system. This means that you have to make corresponding changes to the software system. To ensure that the changes are applied to the system in a controlled way, you need a set of tool-supported, change management procedures.

Change management procedures are concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components of the system have been changed. The change management process (Figure 10.3) should come into effect when the software or associated documentation is baselined by the configuration management team.

```

Request change by completing a change request form
Analyze change request
if change is valid then
    Assess how change might be implemented
    Assess change cost
    Record change request in database
    Submit request to change control board
    if change is accepted then
        repeat
            make changes to software
            record changes and link to associated change request
            submit changed software for quality approval
        until software quality is adequate
        create new system version
    else
        reject change request
else
    reject change request

```

Figure 10.3: The change management process [1]

The first stage in the change management process is to complete a change request form (CRF) describing the change required to the system. As well as recording the change required, the CRF records the recommendations regarding the change, the estimated costs of the change and the dates when the change was requested, approved, implemented and validated. The CRF may also include a section where an analyst outline show the change is to be implemented.

| Change Request Form  |                                  |
|--|----------------------------------|
| <b>Project:</b> Proteus/PCL-Tools  | <b>Number:</b> 23/02             |
| <b>Change requester:</b> I. Sommerville  | <b>Date:</b> 1/12/02             |
| <b>Requested change:</b> When a component is selected from the structure, display the name of the file where it is stored.   |                                  |
| <b>Change analyser:</b> G. Dean  | <b>Analysis date:</b> 10/12/02   |
| <b>Components affected:</b> Display-Icon.Select, Display Icon.Display  |                                  |
| <b>Associated components:</b> FileTable  |                                  |
| <b>Change assessment:</b> Relatively simple to implement as a file name table is available. Requires the design and implementation of a display field. No changes to associated components are required. |                                  |
| <b>Change priority:</b> Low  |                                  |
| <b>Change implementation</b>   |                                  |
| <b>Estimated effort:</b> 0.5 days  |                                  |
| <b>Date to CCB:</b> 15/12/02   | <b>CCB decision date:</b> 1/2/03 |
| <b>CCB decision:</b> Accept change. Change to be implemented in Release 2.1.   |                                  |
| <b>Change implementor:</b>   | <b>Date of change:</b>           |
| <b>Date submitted to QA:</b>   | <b>QA decision:</b>              |
| <b>Date submitted to CM:</b>   |                                  |
| <b>Comments</b>  |                                  |

Figure 10.4: A partially completed change request form [1]

An example of a partially completed change request form is shown in Figure 10.4. The change request form is usually defined during the CM planning process. This is an example of a CRF that might be used in a large complex systems engineering project. For smaller projects, it is recommended that change requests should be formally recorded, but the CRF should focus on describing the change required with less focus on implementation issues. The engineer making the change decides how to implement that change in these situations.

Once a change request form has been submitted, it should be registered in the configuration database. The change request is then analyzed to check that the change requested is necessary. Some change requests may be due to misunderstandings rather than system faults and no system change is necessary. Others may refer to already known faults. If the analysis discovers that a change request is invalid, duplicated or has already been considered, the change is rejected. You should tell the person who submitted the change request why it has been rejected.

*For valid changes, the next stage of the process is change assessment and costing. The impact of the change on the rest of the system must be checked.* This involves identifying all of the components affected by the change using information from the configuration database and the source code of the software. If making the change means that further changes elsewhere in the system are needed, this clearly increases the cost of change implementation. Next, the required changes to the system modules are assessed. Finally, the cost of making the change is estimated, taking into account the costs of changing related components.

A change control board (CCB) should review and approve all change requests unless the changes simply involve correcting minor errors on screen displays, web pages or in documents. The CCB considers the impact of the change from a strategic and organizational rather than a technical point of view. The board should decide whether the change is economically justified and should prioritize the changes that have been accepted.

The term change control board implies a rather grand group that makes change decisions. Such formally structured CCBs, including senior client and contractor staff, are a requirement of military projects. However, for small or medium-sized projects, the CCB may simply consist of a project manager plus one or two engineers who are not directly involved in the software development. In some cases, the CCB may be a single change reviewer who gives advice on whether changes are justifiable.

Change management for generic, shrink-wrapped software products rather than systems that are tailored for a specific customer has to be handled in a slightly different way. In these systems, the customer is not directly involved so the relevance of the change to the customer's business is not an issue. Change requests in these products are usually associated with bugs in the system that have been discovered during system testing or by customers after the software has been released. Customers may use a web page or e-mail to report bugs. A bug management team then checks that the bug reports are valid and translates them into formal system change requests. As with other types of systems, changes have to be prioritized for implementation and bugs may not be repaired if the repair costs are too high.

During development, when new versions of the system are created through daily (or more frequent) system builds, a simpler change management process is used. Problems and changes must still be recorded, but changes that affect only individual components and modules need not be independently

assessed. They are passed directly to the system developer. The system developer either accepts them or makes a case why they are not required. Changes that affect system modules produced by different development teams, however, should be assessed by a change control authority who prioritizes them for implementation.

In some agile methods, such as extreme programming, customers are directly involved in deciding whether a change should be implemented. When they propose a change to the system requirements, they work with the team to assess the impact of that change and then decide whether the change should take priority over the features planned for the next increment of the system. However, changes that involve software improvement are left to the discretion of the programmers working on the system. Refactoring, where the software is continually improved, is not seen as an overhead but rather as a necessary part of the development process.

As software components are changed, a record of the changes made to each component should be maintained. This is sometimes called the derivation history of a component. A good way to keep the derivation history is in a standardized comment at the beginning of the component source code (see Figure 10.5). This comment should reference the change request that triggered the software change. You can then write simple scripts that scan all components and process the derivation histories to produce component change reports. A similar approach can be used for web pages. For published documents, records of changes incorporated in each version are usually maintained in a separate page at the front of the document.

### 10.3. Version and Release Management

The processes involved in version and release management are concerned with identifying and keeping track of the versions of a system. Version managers devise procedures to ensure that versions of a system may be retrieved when required and are not accidentally changed by the development team. For products, version managers work with marketing staff and, for custom systems with customers, to plan when new releases of a system should be created and distributed for deployment.

A system version is an instance of a system that differs, in some way, from other instances. Versions of the system may have different functionality, enhanced performance or repaired software faults. Some versions may be functionally equivalent but designed for different hardware or software configurations. Versions with only small differences are sometimes called *variants*.

A system release is a version that is distributed to customers. Each system release should either include new functionality or should be intended for a different hardware platform. There are normally many more versions of a system than releases. *Versions are created within an organization for internal development or testing and are not intended for release to customers.*

CASE tools are now always used to support version management. These tools manage the storage of each version of the software and control access to system components. Components must be checked out from the system for editing. Re-entering (checking in) the component creates a new version, and an identifier is assigned by the version management system. While tools obviously differ significantly in the features offered and their user interfaces, the general principles of version management covered here are the basis for all support tools.



## VERSION IDENTIFICATION

To create a particular version of a system, you have to specify the versions of the system components that should be included in it. In a large software system, there are hundreds of software components, each of which may exist in several different versions. There must therefore be an unambiguous way to identify each component version to ensure that the right components are included in the system. However, you cannot use the configuration item name for version identification because there may be several versions of each identified configuration item.

Instead, three basic techniques are used for component version identification:

1. **Version numbering:** The component is given an explicit, unique version number. This is the most commonly used identification scheme.
2. **Attribute-based identification:** Each component has a name (such as the configuration item name, which is not unique across versions) and an associated set of attributes for each version. Components are therefore identified by specifying their name and attribute values.
3. **Change-oriented identification:** Each component is named as in attribute-based identification but is also associated with one or more change requests. That is, it is assumed that each version of the component has been created in response to one or more change requests. The component version is identified by the set of change requests that apply to the component.

## VERSION NUMBERING

In a simple version-numbering scheme, a version number is added to the component or system name. Therefore, you might refer to Solaris 4.3 (version 4.3 of the Solaris system) and version 1.4 of component *getToken*. If the first version is called 1.0, subsequent versions are 1.1, 1.2, and so on. At some stage, a new release is created (release 2.0) and the process starts again at version 2.1. The scheme is linear, based on the assumption that system versions are created in sequence. Most version management tools such as RCS and CVS support this approach to version identification.

This approach and the derivation of a number of different system versions are illustrated in Figure 10.6. The arrows in this diagram point from the source version to the new version created from that source. Notice that the derivation of versions is not necessarily linear and versions with consecutive version numbers may be produced from different baselines. For example, in Figure 10.6, version 2.2 is created from version 1.2 rather than from version 2.1. In principle, any existing version may be used as the starting point for a new version of the system.

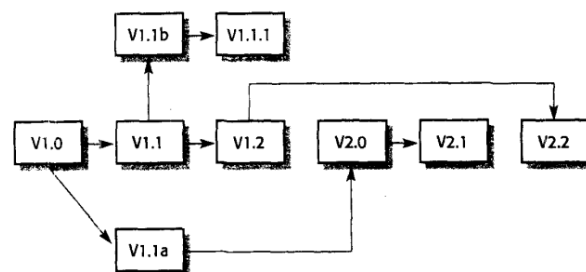


Figure 10.6: Different system versions [1]

This scheme is simple, but you need to maintain a lot of extra information to keep track of the differences between versions and the relationships between system change proposals and versions. For example, versions 1.1 and 1.2 of a system might differ because version 1.2 has been produced using a different graphics library.

*The name tells you nothing about the version or why it was created.* Consequently, you need to keep records in the configuration database that describe each version and why it was produced. You may also need to explicitly link change requests to the different versions of each component.

### **ATTRIBUTE-BASED IDENTIFICATION**

A fundamental problem with explicit version naming schemes is that they do not reflect the many attributes that may be used to identify versions. Examples of these identifying attributes are: *Customer, Development language, Development status, Hardware platform, and Creation date.*

If each version is identified by a unique set of attributes, it is easy to add new versions that are derived from any of the existing versions. These are identified using a unique set of attribute values. They share most of these values with their parent version so relationships between versions are maintained. You can retrieve specific versions by specifying the attribute values required. Functions on attributes support queries such as 'the most recently created version' or 'the version created between given dates'.

For example, the version of the software system AC3D developed in Java for Windows XP in January 2003 would be identified:

AC3D (language=Java, Platform= XP, date=Jan 2003)

Using a general specification of the components in AC3D, the version management tool selects the versions of components that have the attributes 'Java', 'XP' and 'Jan2003'. Attribute-based identification may be implemented directly by the version management system, with component attributes maintained in a system database.

Alternatively, the attribute identification system may be built as a layer on top of a hidden version-numbering scheme. The configuration database then maintains the links between identifying attributes and underlying system and component versions.

### **CHANGE-ORIENTED IDENTIFICATION**

Attribute-based identification of system versions removes some of the version retrieval problems of simple version numbering schemes. However, to retrieve a version, you still have to know its associated attributes. Furthermore, you still need to use a separate change management system to discover the relationships between versions and changes.

Change-oriented identification is used to identify system versions rather than components. The version identifiers of individual components are hidden from users of the CM system. Each system change that has been implemented has an associated change set that describes the requested changes made to the different system components. Change sets may be applied in sequence so that, in principle at least, the version may incorporate an arbitrary set of changes. For example, the set of changes to a system that were made to adapt it for Linux rather than Solaris could be applied, followed by the changes required to

incorporate a new system database. Equally, the Linux/Solaris changes could be followed by changes that converted the user interface language from English to Italian.

In practice, of course, it isn't possible to apply arbitrary sets of changes to a system. The change sets may be incompatible so that applying change set A followed by change set D may create an invalid system. Furthermore, change sets may conflict in that different changes affect the same code of the system. If the code has been changed by change set A then change set D may no longer work. To address these difficulties, version management tools that support change-oriented identification allow system consistency rules to be specified. These limit the ways in which change sets may be combined.

## **RELEASE MANAGEMENT**

A system release is a version of the system that is distributed to customers. System release managers are responsible for deciding when the system can be released to customers, managing the process of creating the release and the distribution media, and documenting the release to ensure that it may be re-created exactly as distributed if this is necessary.

A system release is not just the executable code of the system. The release may also include:

- Configuration files defining how the release should be configured for particular installations
- Data files that are needed for successful system operation
- An installation program that is used to help install the system on target hardware
- Electronic and paper documentation describing the system
- Packaging and associated publicity that have been designed for that release

*Release managers cannot assume that customers will always install new system releases. Some system users may be happy with an existing system.* They may consider it not worth the cost of changing to a new release. New releases of the system cannot, therefore, rely on the installation of previous releases. To illustrate this problem, consider the following scenario:

- Release 1 of a system is distributed and put into use.
- Release 2 requires the installation of new data files, but some customers do not need the facilities of release 2 so remain with release 1.
- Release 3 requires the data files installed in release 2 and has no new data files of its own.

*The software distributor cannot assume that the files required for release 3 have already been installed in all sites.* Some sites may go directly from release 1 to release 3, skipping release 2. Some sites may have modified the data files associated with release 2 to reflect local circumstances. Therefore, the data files must be distributed and installed with release 3 of the system.

## **RELEASE DECISION MAKING**

Preparing and distributing a system release is an expensive process, particularly for mass-market software products. If releases are too frequent, customers may not upgrade to the new release, especially if it is not free. If system releases are infrequent, market share may be lost as customers move to alternative systems. This, of course, does not apply to custom software developed specially for an organization. For custom

software, infrequent releases may mean increasing divergence between the software and the business processes that it is designed to support.

The various technical and organizational factors that you should take into account when deciding to create a new system release are shown in Table 10.1.

| Factor                          | Description   |
|---------------------------------|---|
| Technical quality of the system | If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. However, minor system faults may be repaired by issuing patches (often distributed over the Internet) that can be applied to the current release of the system. |
| Platform changes                | You may have to create a new release of a software application when a new version of the operating system platform is released.   |
| Lehman's fifth law              | This suggests that the increment of functionality that is included in each release is approximately constant. Therefore, a system release with significant new functionality may have to be followed by a repair release.   |
| Competition                     | A new system release may be necessary because a competing product is available.   |
| Marketing requirements          | The marketing department of an organization may have made a commitment for releases to be available at a particular date.   |
| Customer change proposals       | For customized systems, customers may have made and paid for a specific set of system change proposals, and they expect a system release as soon as these have been implemented.  |

Table 10.1: Factors influencing system release strategy [1]

## RELEASE CREATION

Release creation is the process of creating a collection of files and documentation that includes all of the components of the system release. The executable code of the programs and all associated data files must be collected and identified.

Configuration descriptions may have to be written for different hardware and operating systems and instructions prepared for customers who need to configure their own systems. If machine-readable manuals are distributed, electronic copies must be stored with the software. Scripts for the installation program may have to be written. Finally, when all information is available, the release directory is handed over for distribution.

The normal distribution medium for system releases is now optical disks (CD-ROM or DVD) that can store from 600 Megabytes to 4 Gigabytes of data. In addition, software may be released online, allowing customers to download it from the Internet, although many people find it takes too long to download large files and prefer CD-ROM distribution.

There are very high marketing and packaging costs associated with distributing new releases of software products, so product vendors usually create new releases only for new platforms or to add significant new functionality. They then charge users for this new software. When problems are discovered in an existing release, the vendors usually make patches to repair the existing software available on a website for downloading by customers.

Apart from the costs of finding and downloading the new release, the problem is that many customers may never discover the existence of these repairs or may not have the technical knowledge to install them. They may instead continue using their existing, faulty system with the consequent risks to their business. In some situations, where the patch is designed to repair security loopholes, the risks of failing to install the patch can mean that the business is susceptible to external attacks.

## RELEASE DOCUMENTATION

When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future. This is particularly important for customized, long life time embedded systems such as those controlling complex machines. Customers may use a single release of these systems for many years and may require specific changes to a particular software release long after its original release date.

To document a release, you have to record the specific versions of the source code components that were used to create the executable code. You must keep copies of the source and executable code and all data and configuration files. You should also record the versions of the operating system, libraries, compilers and other tools used to build the software. These may be required to build exactly the same system at some later date. This may mean that you have to store copies of the platform software and the tools used to create the system in the version management system along with the source code of the target system.

### 10.4. System Building

System building is the process of compiling and linking software components into a program that executes on a particular target configuration. When you are building a system from its components, you have to think about the following questions:

- Have all the components that make up a system been included in the build instructions?
- Have the appropriate version of each required component been included in the build instructions?
- Are all required data files available?
- If data files are referenced within a component; is the name used the same as the name of the data file on the target machine?
- Is the appropriate version of the compiler and other required tools available? Current versions of software tools may be incompatible with the older versions used to develop the system.

Nowadays, software configuration management tools or, sometimes, the programming environment are used to automate the system-building process. The CM team writes a build script that defines the dependencies between the system components. This script also defines the tools used to compile and link the system components. The system-building tool interprets the build script and calls other programs as required to build the executable system from its components. This is illustrated in Figure 10.7. In some programming environments (such as Java development environments), the build script is created automatically by parsing the source code and discovering which components are called. Of course, in this situation, the name of the stored component has to be the same as the name of the program component.

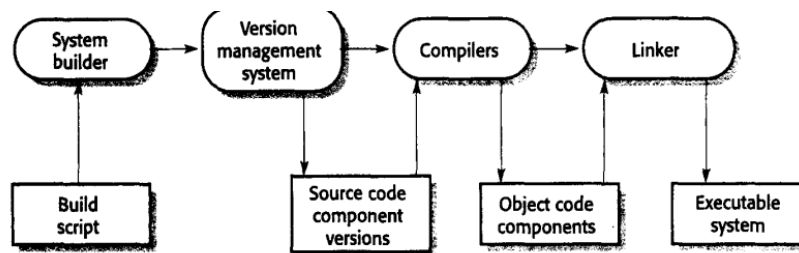


Figure 10.7: System building [1]

Dependencies between components are specified in the build script. This provides information so that the system-building tool can decide when the source code of components must be recompiled and when existing object code can be reused. In many tools, these build-script dependencies are often specified as dependencies between the physical files in which the source code and object code of components are stored. However, when there are multiple source code files representing multiple versions of components, it may be difficult to tell which source files were used to derive object-code components. This confusion is particularly likely when the correspondence between source and object code files relies on them having the same name but a different suffix (e.g., .c and .o). This problem can only be solved when the version management and system-building tools are integrated.

### 10.5. CASE Tools for Configuration Management

Configuration management processes are usually standardized and involve the application of predefined procedures. They require careful management of very large amounts of data, and attention to detail is essential. When a system is being built from component versions, a single configuration management mistake can mean that the software will not work properly. Consequently, CASE tool support is essential for configuration management, and, since the 1970s, many software tools covering different areas of configuration management have been produced.

1. **Open workbenches:** Tools for each stage in the CM process are integrated through standard organizational procedures for using these tools. There are many commercial and open-source CM tools available for specific purposes. Change management can be supported by bug-tracking tools such as *Bugzilla*, version management tools such as *RCS* or *CVS*, and system building by using tools such as *make* or *imake*. These are all open source tools that are freely available.
2. **Integrated workbenches:** These workbenches provide integrated facilities for version management, system building and change tracking. For example, *Rational's Unified Change Management* process relies on an integrated CM workbench incorporating *ClearCase* for system building and version management and *ClearQuest* for change tracking. The advantages of integrated CM workbenches are that data exchange is simplified, and the workbench includes an integrated CM database. *Integrated SCM workbenches* have been derived from earlier systems such as *Lifespan* for change management and *OSEE* for version management and system building. However, integrated *eM workbenches* are complex and expensive, and many organizations prefer to use cheaper and simpler individual tool support.

Many large systems are developed at different sites, and these need SCM tools that support multisite working with multiple data stores for configuration items. While most SCM tools are designed for single site working, some tools, such as *CVS*, have facilities for multisite support.

### SUPPORT FOR CHANGE MANAGEMENT

Each person involved in the change management process is responsible for some activity. They complete this activity, and then pass on the forms and associated configuration items to someone else. The procedural nature of this process means that a change process model can be designed and integrated with a version management system. This model may then be interpreted so that the right documents are passed to the right people at the right time.

There are several change management tools available, from relatively simple, open source tools such as Bugzilla to comprehensive integrated systems such as Rational ClearQuest. These tools provide some or all of the following facilities to support the process:

1. **A form editor** that allows change proposal forms to be created and completed by people making change requests.
2. **A workflow system** that allows the CM team to define who must process the change request form and the order of processing. This system will also automatically pass forms to the right people at the right time and inform the relevant team members of the progress of the change. E-mail is used to provide progress updates for those involved in the process.
3. **A change database** that is used to manage all change proposals and that may be linked to a version management system. Database query facilities allow the eM team to find specific change proposals.
4. **A change-reporting system** that generates management reports on the status of change requests that have been submitted.

## SUPPORT FOR VERSION MANAGEMENT

Version management involves managing large amounts of information and ensuring that system changes are recorded and controlled. Version management tools control a repository of configuration items where the contents of that repository are immutable (i.e., cannot be changed). To work on a configuration item, you must check it out of the repository into a working directory. After you have made the changes to the software, you check it back into the repository and a new version is automatically created. All version management systems provide a comparable basic set of capabilities although some have more sophisticated facilities than others. Examples of these capabilities are:

1. **Version and release identification:** Managed versions are assigned identifiers when they are submitted to the system. Different systems support the different types of version identification.
2. **Storage management:** To reduce the storage space required by multiple versions that are largely the same, version management systems provide storage management facilities so that versions are described by their differences from some master version. Differences between versions are represented as a delta, which encapsulates the instructions required to recreate the associated system version. This is illustrated in Figure 10.8, which shows how backward deltas may be applied to the latest version of a system to re-create earlier system versions. The latest version is version 1.3. To create version 1.2, you apply the change delta that re-creates that version.

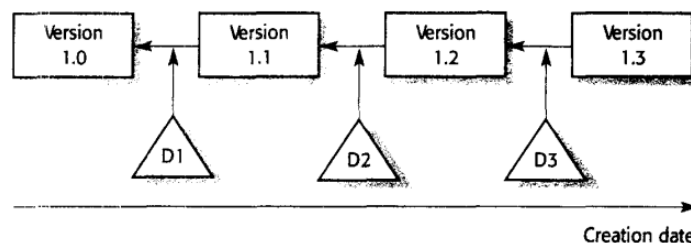


Figure 10.8: Delta based versioning [1]

3. ***Change history recording:*** All of the changes made to the code of a system or component are recorded and listed. In some systems, these changes may be used to select a particular system version.
4. ***Independent development:*** Multiple versions of a system can be developed in parallel and each version may be changed independently. For example, release 1 can be modified after development of release 2 is in progress by adding new level-1 deltas. The version management system keeps track of components that have been checked out for editing and ensures that changes made to the same component by different developers do not interfere. Some systems allow only one instance of a component to be checked out for editing; others resolve potential clashes when the edited components are checked back into the system
5. ***Project support:*** The system can support multiple projects as well as multiple file: In project support systems, such as CVS, it is possible to check in and check out all of the files associated with a project rather than having to work with one file at a time.

## SUPPORT FOR SYSTEM BUILDING

System building is a computationally intensive process. Compiling and linking all of the components of a large system can take several hours. There may be hundreds of files involved, with the consequent possibility of human error if these are compiled and linked manually. System-building tools automate the build process to reduce the potential for human error and, where possible, minimize the time required for system building.

System-building tools may be stand alone, such as derivatives of the Unix-make utility, or may be integrated with version management tools. Facilities provided by system-building CASE tools may include:

- ***A dependency specification language and associated interpreter:*** Component dependencies may be described and recompilation minimized.
- ***Tool selection and instantiation support:*** The compilers and other processing tool that is used to process the source code files may be specified and instantiated as required.
- ***Distributed compilation:*** Some system builders, especially those that are part of integrated CM systems, support distributed network compilation. Rather than all compilations are being carried out on a single machine, the system builder looks for idle processors on the network and sets off a number of parallel compilations. This significantly reduces the time required to build a system.
- ***Derived object management:*** Derived objects are objects created from other source objects. Derived object management links the source code and the derived objects and re-derives only an object when this is required by source code changes.

Most system-building tools use the file modification date as the key attribute in deciding whether recompilation is required. If a source code file is modified after its corresponding object code file, then the object code file is re-created. Essentially, there can only ever be one version of the object code corresponding to the most recently changed source code component. When a new version of a source code component is re-created, the object code for the previous version is lost.

However, some tools use a more sophisticated approach to derived object management. They tag derived objects with the version identifier of the source code used to generate these objects. Within the limits of



storage capacity, they maintain all derived objects. Therefore, it is usually possible to recover the object code of all versions of source code components without recompilation.

## **REFERENCES**

[1] I. Sommerville , 2008. Software Engineering. Eighth Edition, Addison-Wesley