Module 7

Software Engineering Issues

Lesson 33

Introduction to Software Engineering

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Get an introduction to software engineering
- Understand the need for software engineering principles
- Identify the causes of and solutions for software crisis
- Differentiate a piece of program from a software product
- Understand the evolution of software design techniques over last 50 years
- Identify the features of a structured program and its advantages
- Identify the features of various design techniques
- Differentiate between the exploratory style and modern styles of software development
- Explain what a life cycle model is
- Understand the need for a software life cycle model
- Identify the different phases of the classical waterfall model and related activities
- Identify the phase-entry and phase-exit criteria of each phase
- Explain what a prototype is
- Explain the need for prototype development
- State the activities carried out during each phase of a spiral model

1. Introduction

With the advancement of technology, computers have become more powerful and sophisticated. The more powerful a computer is, the more sophisticated programs it can run. Thus, programmers have been tasked to solve larger and more complex problems. They have coped with this challenge by innovating and by building on their past programming experience. All those past innovations and experience of writing good quality programs in efficient and cost-effective ways have been systematically organized into a body of knowledge. This body of knowledge forms the basis of software engineering principles. Thus, we can view software engineering as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines.

1.1. The Need for Software Engineering

Alternatively, software engineering can be viewed an engineering approach to software development. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are indispensable to achieve a good quality software cost effectively. These definitions can be elaborated with the help of a building construction analogy.

Suppose you have a friend who asked you to build a small wall as shown in fig. 33.1. You would be able to do that using your common sense. You will get building materials like bricks; cement etc. and you will then build the wall.

Fig. 33.1 A Small Wall

But what would happen if the same friend asked you to build a large multistoried building as shown in fig. 33.2?

Fig. 33.2 A Multistoried Building

You don't have a very good idea about building such a huge complex. It would be very difficult to extend your idea about a small wall construction into constructing a large building. Even if you tried to build a large building, it would collapse because you would not have the requisite knowledge about the strength of materials, testing, planning, architectural design, etc. Building a small wall and building a large building are entirely different ball games. You can use your intuition and still be successful in building a small wall, but building a large building requires knowledge of civil, architectural and other engineering principles.

Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes as shown in fig. 33.3. For example, a program of size 1,000 lines of code has some complexity. But a program with 10,000 LOC is not just 10 times more difficult to develop, but may as well turn out to be 100 times more difficult unless software engineering principles are used. In such situations software engineering techniques come to the rescue. Software engineering helps to reduce programming complexity. Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition.

The principle of abstraction (in fig. 33.4) implies that a problem can be simplified by omitting irrelevant details. Once the simpler problem is solved then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on.

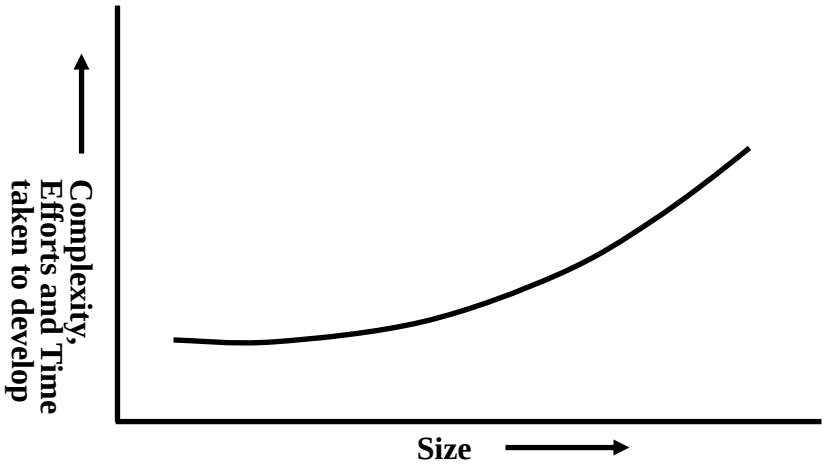

Fig. 33.3 Increase in development time and effort with problem size

1.1.1. Abstraction and Decomposition

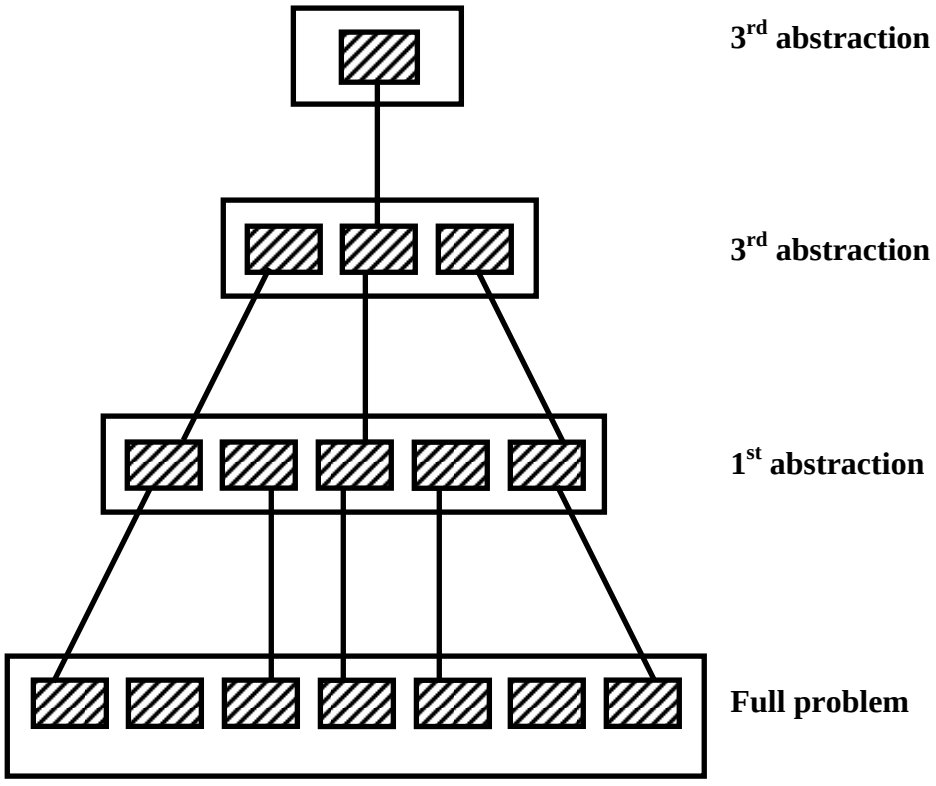

Fig. 33.4 A hierarchy of abstraction

The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different

components can be combined to get the full solution. A good decomposition of a problem as shown in fig. 33.5 should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

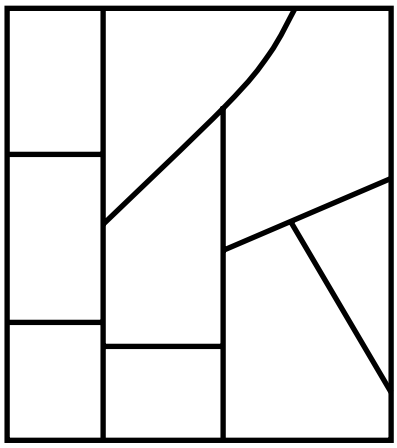

Fig. 33.5 Decomposition of a large problem into a set of smaller problems

1.2. The Software Crisis

Software engineering appears to be among the few options available to tackle the present software crisis.

To explain the present software crisis in simple words, consider the following. The expenses that organizations all around the world are incurring on software purchases compared to those on hardware purchases have been showing a worrying trend over the years (as shown in fig. 33.6)

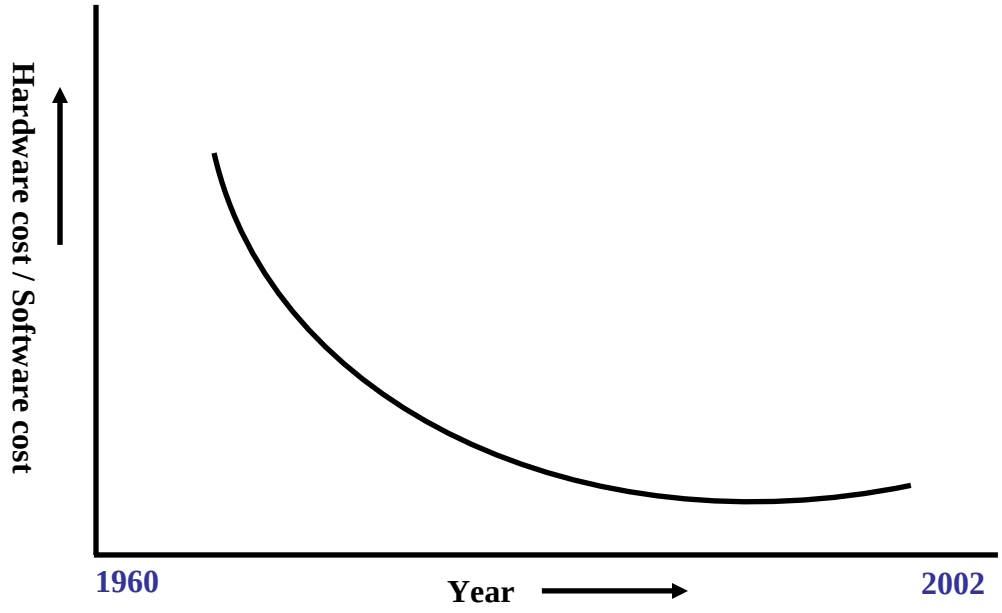

Fig. 33.6 Change in the relative cost of hardware and software over time

Organizations are spending larger and larger portions of their budget on software. Not only are the software products turning out to be more expensive than hardware, but they also present a host of other problems to the customers: software products are difficult to alter, debug, and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late. Among these, the trend of increasing software costs is probably the most important symptom of the present software crisis. Remember that the cost we are talking of here is not on account of increased features, but due to ineffective development of the product characterized by inefficient resource usage, and time and cost over-runs.

There are many factors that have contributed to the making of the present software crisis. Factors are larger problem sizes, lack of adequate training in software engineering, increasing skill shortage, and low productivity improvements.

It is believed that the only satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the engineers, coupled with further advancements to the software engineering discipline itself.

1.3. Program vs. Software Product

Programs are developed by individuals for their personal use. They are therefore, small in size and have limited functionality but software products are extremely large. In case of a program, the programmer himself is the sole user but on the other hand, in case of a software product, most users are not involved with the development. In case of a program, a single developer is involved but in case of a software product, a large number of developers are involved. For a program, the user interface may not be very important, because the programmer is the sole user. On the other hand, for a software product, user interface must be carefully designed and implemented because developers of that product and users of that product are totally different. In case of a program, very little documentation is expected, but a software product must be well documented. A program can be developed according to the programmer's individual style of development, but a software product must be developed using the accepted software engineering principles.

2. Evolution of Program Design Techniques

During the 1950s, most programs were being written in assembly language. These programs were limited to about a few hundreds of lines of assembly code, i.e. were very small in size. Every programmer developed programs in his own individual style - based on his intuition. This type of programming was called Exploratory Programming.

The next significant development which occurred during early 1960s in the area computer programming was the high-level language programming. Use of high-level language programming reduced development efforts and development time significantly. Languages like FORTRAN, ALGOL, and COBOL were introduced at that time.

2.1. Structured Programming

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. To cope with this problem, experienced programmers advised other programmers to pay particular attention to the design of the program's control flow structure (in late 1960s). In the late 1960s, it was found that the "GOTO" statement was the main culprit which makes control structure of a program complicated and messy. At that time most of the programmers used assembly languages extensively. They considered use of "GOTO" statements in high-level languages were very natural because of their familiarity with JUMP statements which are very frequently used in assembly language programming. So they did not really accept that they can write programs without using GOTO statements, and considered the frequent use of GOTO statements inevitable. At this time, **Dijkstra [1968]** published his (now famous) article "GOTO Statements Considered Harmful". Expectedly, many programmers were enraged to read this article. They published several counter articles highlighting the advantages and inevitable use of GOTO statements. But, soon it was conclusively proved that only three programming constructs – sequence, selection, and iteration – were sufficient to express any programming logic. This formed the basis of the structured programming methodology.

2.1.1. Features of Structured Programming

A structured program uses three types of program constructs i.e. selection, sequence and iteration. Structured programs avoid unstructured control flows by restricting the use of GOTO statements. A structured program consists of a well partitioned set of modules. Structured programming uses single entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, the structured programming principle emphasizes designing neat control structures for programs.

2.1.2. Advantages of Structured Programming

Structured programs are easier to read and understand. Structured programs are easier to maintain. They require less effort and time for development. They are amenable to easier debugging and usually fewer errors are made in the course of writing such programs.

2.2. Data Structure-Oriented Design

After structured programming, the next important development was data structure-oriented design. Programmers argued that for writing a good program, it is important to pay more attention to the design of data structure, of the program rather than to the design of its control structure. Data structure-oriented design techniques actually help to derive program structure from the data structure of the program. Example of a very popular data structure-oriented design technique is Jackson's Structured Programming (JSP) methodology, developed by Michael Jackson in the1970s.

2.3. Data Flow-Oriented Design

Next significant development in the late 1970s was the development of data flow-oriented design technique. Experienced programmers stated that to have a good program structure, one has to study how the data flows from input to the output of the program. Every program reads data and then processes that data to produce some output. Once the data flow structure is identified, then from there one can derive the program structure.

2.4. Object-Oriented Design

Object-oriented design (1980s) is the latest and very widely used technique. It has an intuitively appealing design approach in which natural objects (such as employees, pay-roll register, etc.) occurring in a problem are first identified. Relationships among objects (such as composition, reference and inheritance) are determined. Each object essentially acts as a data hiding entity.

2.5. Changes in Software Development Practices

An important difference is that the exploratory software development style is based on error correction while the software engineering principles are primarily based on error prevention. Inherent in the software engineering principles is the realization that it is much more cost-effective to prevent errors from occurring than to correct them as and when they are detected. Even when errors occur, software engineering principles emphasize detection of errors as close to the point where the errors are committed as possible. In the exploratory style, errors are detected only during the final product testing. In contrast, the modern practice of software development is to develop the software through several well-defined stages such as requirements specification, design, coding, testing, etc., and attempts are made to detect and fix as many errors as possible in the same phase in which they occur.

In the exploratory style, coding was considered synonymous with software development. For instance, exploratory programming style believed in developing a working system as quickly as possible and then successively modifying it until it performed satisfactorily.

In the modern software development style, coding is regarded as only a small part of the overall software development activities. There are several development activities such as design and testing which typically require much more effort than coding.

A lot of attention is being paid to requirements specification. Significant effort is now being devoted to develop a clear specification of the problem before any development activity is started.

Now, there is a distinct design phase where standard design techniques are employed.

Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is phase containment of errors, i.e. detect and correct errors as soon as possible. Defects are usually not detected as soon as they occur, rather they are noticed much later in the life cycle. Once a defect is detected, we have to go back to the phase

where it was introduced and rework those phases - possibly change the design or change the code and so on.

Today, software testing has become very systematic and standard testing techniques are available. Testing activity has also become all encompassing in the sense that test cases are being developed right from the requirements specification stage.

There is better visibility of design and code. By visibility we mean production of good quality, consistent and standard documents during every phase. In the past, very little attention was paid to producing good quality and consistent documents. In the exploratory style, the design and test activities, even if carried out (in whatever way), were not documented satisfactorily. Today, consciously good quality documents are being developed during product development. This has made fault diagnosis and maintenance smoother.

Now, projects are first thoroughly planned. Project planning normally includes preparation of various types of estimates, resource scheduling, and development of project tracking plans. Several techniques and tools for tasks such as configuration management, cost estimation, scheduling, etc. are used for effective software project management.

Several metrics are being used to help in software project management and software quality assurance.

3. Software Life Cycle Model

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to its retirement. Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out. For example, the design phase might consist of the structured analysis activity followed by the structured design activity.

3.1. The Need for a Life Cycle Model

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using a particular life cycle model, the development of a software product would not be in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. Let us try to illustrate this problem using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to

prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure.

A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without a software life cycle model, the entry and exit criteria for a phase cannot be recognized. Without models (such as classical waterfall model, iterative waterfall model, prototyping model, evolutionary model, spiral model etc.), it becomes difficult for software project managers to monitor the progress of the project.

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

- Classical Waterfall Model
- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

3.2. Classical Waterfall Model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, we will see that it is not a practical model in the sense that it can not be used in actual software development projects. Thus, we can consider this model to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models, we must first learn the classical waterfall model.

Classical waterfall model divides the life cycle into the following phases as shown in fig. 33.7:

- Feasibility study
- Requirements analysis and specification
- Design
- Coding and unit testing
- Integration and system testing
- Maintenance

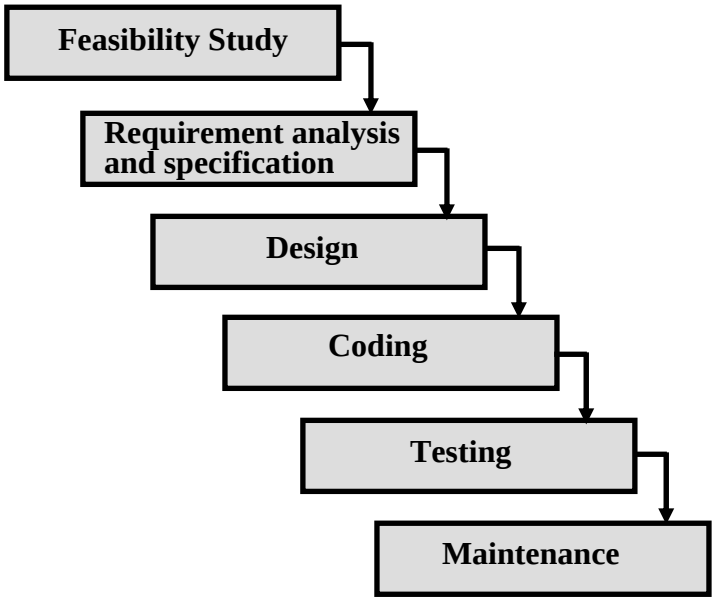

Fig. 33.7 Classical Waterfall Model

3.2.1. Feasibility Study

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product

- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behaviour of the system.
- After they have an overall understanding of the problem, they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kinds of resources are required, what would be the cost of development and what would be the development time for each solution.
- Based on this analysis, they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

The following is an example of a feasibility study undertaken by an organization. It is intended to give one a feel of the activities and issues involved in the feasibility study phase of a typical software project.

Case Study

A mining company named Galaxy Mining Company Ltd. (GMC) has mines located at various places in India. It has about fifty different mine sites spread across eight states. The company employs a large number of mines at each mine site. Mining being a risky profession, the company intends to operate a special provident fund, which would exist in addition to the standard provident fund that the miners already enjoy. The main objective of having the special provident fund (SPF) would be to quickly distribute some compensation before the standard provident amount is paid. According to this scheme, each mine site would deduct SPF instalments from each miner every month and deposit the same with the CSPFC (Central Special Provident Fund Commissioner). The CSPFC will maintain all details regarding the SPF instalments collected from the miners. GMC employed a reputed software vendor Adventure Software Inc. to undertake the task of developing the software for automating the maintenance of SPF records of all employees. GMC realized that besides saving manpower on bookkeeping work, the software would help in speedy settlement of claim cases. GMC indicated that the amount it could afford for this software to be developed and installed was 1 million rupees.

Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed the matter with the top managers of GMC to get an overview of the project. He also discussed the issues involved with the several field PF officers at various mine sites to determine the exact details of the project. The project manager identified two broad approaches to solve the problem. One was to have a central database which could be accessed and updated via a satellite connection to various mine sites. The other approach was to have local databases at each mine site and to update the central database periodically through a dial-up connection. These periodic updates could be done on a daily or hourly basis depending on the delay acceptable to GMC in invoking various functions of the software. The project manager found that the second approach was very affordable and more fault-tolerant as the local mine sites could still operate even when the communication link to the central database temporarily failed. The project manager quickly analyzed the database functionalities required, the user-interface issues, and the software handling communication with the mine sites. He arrived at a cost to develop from the analysis. He found that the solution involving maintenance of local databases at the mine sites and periodic updating of a central database was financially and technically feasible. The project manager discussed his solution with the GMC management and found that the solution was acceptable to them as well.

3.2.2. Requirements Analysis and Specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis, and
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed with a view to clearly understand the customer requirements and weed out the incompleteness and inconsistencies in these requirements.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document.

The customer requirements identified during the requirements gathering and analysis activity are organized into an SRS document. The important components of this document are functional requirements, the non-functional requirements, and the goals of implementation.

3.2.3. Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

Traditional design approach: Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

Object-oriented design approach: In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

3.2.4. Coding and Unit Testing

The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested.

During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

3.2.5. Integration and System Testing

Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner.

The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to the requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

- α – testing: It is the system testing performed by the development team.
- β – testing: It is the system testing performed by a friendly set of customers.
- Acceptance testing: It is the system testing performed by the customer himself after product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

3.2.6. Maintenance

Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

3.2.7. Shortcomings of the Classical Waterfall Model

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

3.2.8. Phase-Entry and Phase-Exit Criteria

At the start of the feasibility study, project managers or team leaders try to understand what the actual problem is, by visiting the client side. At the end of that phase, they pick the best solution and determine whether the solution is feasible financially and technically.

At the start of requirements analysis and specification phase, the required data is collected. After that requirement specification is carried out. Finally, SRS document is produced.

At the start of design phase, context diagram and different levels of DFDs are produced according to the SRS document. At the end of this phase module structure (structure chart) is produced.

During the coding phase each module (independently compilation unit) of the design is coded. Then each module is tested independently as a stand-alone unit and debugged separately. After this each module is documented individually. The end product of the implementation phase is a set of program modules that have been tested individually but not tested together.

After the implementation phase, different modules which have been tested individually are integrated in a planned manner. After all the modules have been successfully integrated and tested, system testing is carried out.

Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use.

3.3. Prototyping Model

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

3.3.1. The Need for a Prototype

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs.

- how screens might look like
- how the user interface would behave
- how the system would produce outputs, etc.

This is something similar to what the architectural designers of a building do; they show a prototype of the building to their customer. The customer can evaluate whether he likes it or not and the changes that he would need in the actual product. A similar thing happens in the case of a software product and its prototyping model.

32.

3.4. Spiral Model

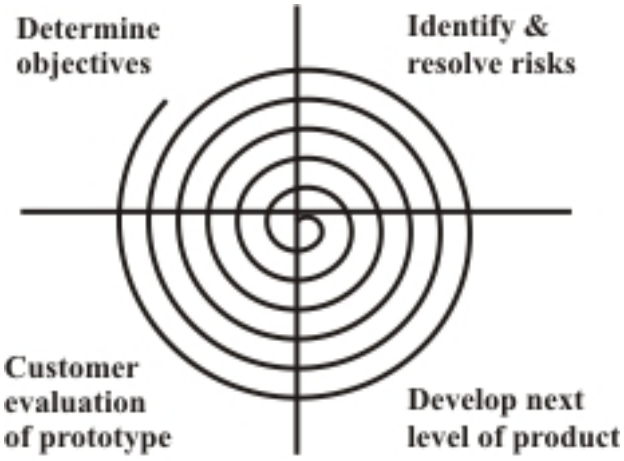

Fig. 33.8 Spiral Model

The Spiral model of software development is shown in fig. 33.8. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study; the next loop with requirements specification; the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. 33.8. The following activities are carried out during each phase of a spiral model.

First quadrant (Objective Setting):
- During the first quadrant, we need to identify the objectives of the phase.
- Examine the risks associated with these objectives

Second quadrant (Risk Assessment and Reduction):
- A detailed analysis is carried out for each identified project risk.
- Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed

Third quadrant (Objective Setting):
- Develop and validate the next level of the product after resolving the identified risks.

Fourth quadrant (Objective Setting):
- Review the results achieved so far with the customer and plan the next iteration around the spiral.
- With each iteration around the spiral, progressively a more complete version of the software gets built.

3.4.1. A Meta Model

The spiral model is called a meta-model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of

technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models. This is probably a factor deterring its use in ordinary projects.

3.5. Comparison of Different Life Cycle Models

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model can not be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta-model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models. This is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the long development process, customer confidence normally drops, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

3.6. Exercises

1. Mark the following as True or False. Justify your answer.
   a. All software engineering principles are backed by either scientific basis or theoretical proof.
   b. There are well defined steps through which a problem is solved using an exploratory style.
   c. Evolutionary life cycle model is ideally suited for development of very small software products typically requiring a few months of development effort.

d. Prototyping life cycle model is the most suitable one for undertaking a software development project susceptible to schedule slippage.
e. Spiral life cycle model is not suitable for products that are vulnerable to a large number of risks.

2. For the following, mark all options which are true.
   a. Which of the following problems can be considered to be contributing to the present software crisis?
      - large problem size
      - lack of rapid progress of software engineering
      - lack of intelligent engineers
      - shortage of skilled manpower
   b. Which of the following are essential program constructs (i.e. it would not be possible to develop programs for any given problem without using the construct)?
      - Sequence
      - Selection
      - Jump
      - Iteration
   c. In a classical waterfall model, which phase precedes the design phase?
      - Coding and unit testing
      - Maintenance
      - Requirements analysis and specification
      - Feasibility study
   d. Among development phases of software life cycle, which phase typically consumes the maximum effort?
      - Requirements analysis and specification
      - Design
      - Coding
      - Testing
   e. Among all the phases of software life cycle, which phase consumes the maximum effort?
      - Design
      - Maintenance
      - Testing
      - Coding
   f. In the classical waterfall model, during which phase is the Software Requirement Specification (SRS) document produced?
      - Design
      - Maintenance
      - Requirements analysis and specification
      - Coding
   g. Which phase is the last development phase in the classical waterfall software life cycle?
      - Design
      - Maintenance
      - Testing
      - Coding

h. Which development phase in classical waterfall life cycle immediately follows coding phase?
   - Design
   - Maintenance
   - Testing
   - Requirement analysis and specification

3. Identify the problem one would face, if he tries to develop a large software product without using software engineering principles.
4. Identify the two important techniques that software engineering uses to tackle the problem of exponential growth of problem complexity with its size.
5. State five symptoms of the present software crisis.
6. State four factors that have contributed to the making of the present software crisis.
7. Suggest at least two possible solutions to the present software crisis.
8. Identify at least four basic characteristics that differentiate a simple program from a software product.
9. Identify two important features of that a program must satisfy to be called as a structured program.
10. Explain exploratory program development style.
11. Show at least three important drawbacks of the exploratory programming style.
12. Identify at least two advantages of using high-level languages over assembly languages.
13. State at least two basic differences between control flow-oriented and data flow-oriented design techniques.
14. State at least five advantages of object-oriented design techniques.
15. State at least three differences between the exploratory style and modern styles of software development.
16. Explain the problems that might be faced by an organization if it does not follow any software life cycle model.
17. Differentiate between structured analysis and structured design.
18. Identify at least three activities undertaken in an object-oriented software design approach.
19. State why it is a good idea to test a module in isolation from other modules.
20. Identify why different modules making up a software product are almost never integrated in one shot.
21. Identify the necessity of integration and system testing.
22. Identify six different phases of a classical waterfall model. Mention the reasons for which classical waterfall model can be considered impractical and cannot be used in real projects.
23. Explain what a software prototype is. Identify three reasons for the necessity of developing a prototype during software development.
24. Explain the situations under which it is beneficial to develop a prototype during software development.
25. Identify the activities carried out during each phase of a spiral model. Discuss the advantages of using spiral model.