Fundamentals of Software Engineering

RAJIB MALL

Contents

1 INTRODUCTION

Commercial usage of computers now spans the last sixty years. Computers were very slow in the initial years and lacked sophistication. Since then, their computational power and sophistication increased rapidly, while their prices dropped dramatically. To get an idea of the kind of improvements that have occurred to computers, consider the following analogy. If similar improvements could have occurred to aircrafts, now personal mini-airplanes should have become available, costing as much as a bicycle, and flying at over 1000 times the speed of the supersonic jets. To say it in other words, the rapid strides in computing technologies are unparalleled in any other field of human endeavour.

Let us now reflect the impact of the astounding progress made to the hardware technologies on the software. The more powerful a computer is, the more sophisticated programs can it run. Therefore, with every increase in the raw computing capabilities of computers, software engineers have been called upon to solve increasingly larger and complex problems, and that too in cost-effective and efficient ways. Software engineers have coped up with this challenge by innovating and building upon their past programming experiences.

1.1 EVOLUTION—FROM AN ART FORM TO A N ENGINEERING DISCIPLINE

In this section, we review how starting from an esoteric art form, the software engineering discipline has evolved over the years.

1.1.1 Evolution of an Art into an Engineering Discipline

Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals. Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline.

1.2 SOFTWARE DEVELOPMENT PROJECTS

Before discussing about the various types of development projects that are being undertaken by software development companies, let us first understand the important ways in which professional software differs from toy software such as those written by a student in his first programming assignment.

1.2.1 Types of Software Development Projects

A software development company is typically structured into a large number of teams that handle various types of software development projects. These software development projects concern the development of either software product or some software service. In the following subsections, we distinguish between these two types of software development projects.

2 SOFTWARE LIFE CYCLE MODELS

In this chapter, w e first discuss a few basic concepts associated with life cycle models. Subsequently, we discuss the important activities that have been prescribed to be carried out in the classical waterfall model. This is intended to provide an insight into the activities that are carried out as part of every life cycle model.

2.1 A FEW BASIC CONCEPTS

In this section, we present a few basic concepts concerning the life cycle models.

2.1.1 Software life cycle

It is well known that all living organisms undergo a life cycle. For example when a seed is planted, it germinates, grows into a full tree, and finally dies. Based on this concept of a biological life cycle, the term software life cycle has been defined to imply the different stages (or phases) over which a software evolves from an initial customer request for it, to a fully developed software, and finally to a stage where it is no longer useful to any user, and then it is discarded.

2.2 WATERFALL MODEL AND ITS EXTENSIONS

The waterfall model and its derivatives were extremely popular in the 1970s and still are heavily being used across many development projects. The waterfall model is possibly the most obvious and intuitive way in which software can be developed through team effort. We can think of the waterfall model as a generic model that has been extended in many ways for catering to certain specific software development situations to realise all other software life cycle models. For this reason, after discussing the classical and iterative waterfall models, we discuss its various extensions.

2.2.1 Classical Waterfall Model

Classical waterfall model is intuitively the most obvious way to develop software. It is simple but idealistic. In fact, it is hard to put this model into use in any non-trivial software development project. One might wonder if this model is hard to use in practical development projects, then why study it at all? The reason is that all other life cycle models can be thought of as being extensions of the classical waterfall model.

Therefore, it makes sense to first understand the classical waterfall model, in order to be able to develop a proper understanding of other life cycle models. Besides, we shall see later in this text that this model though not used for software development; is implicitly used while documenting software.

3 SOFTWARE PROJECT MANAGEMENT

Effective project management is crucial to the success of any software development project. In the past, several projects have failed not for want of competent technical professionals neither for lack of resources, but due to the use of faulty project management practices. Therefore, it is important to carefully learn the latest software project management techniques.

3.1 SOFTWARE PROJECT MANAGEMENT COMPLEXITIES

Management of software projects is much more complex than management of many other types of projects. The main factors contributing to the complexity of managing a software project, as identified by [Brooks75], are the following:

3.1.1 Invisibility

: Software remains invisible, until its development is complete and it is operational. Anything that is invisible, is difficult to manage and control. Consider a house building project. For this project, the project manger can very easily assess the progress of the project through a visual examination of the building under construction. Therefore, the manager can closely monitor the progress of the project, and take remedial actions whenever he finds that the progress is not as per plan. In contrast, it becomes very difficult for the manager of a software project to assess the progress of the project due to the invisibility of software. The best that he can do perhaps is to monitor the milestones that have been completed by the development team and the documents that have been produced—which are rough indicators of the progress achieved.

3.2 RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

In this section, we examine the principal job responsibilities of a project manager and the skills necessary to accomplish those.

3.2.1 Job Responsibilities for Managing Software Projects

A software project manager takes the overall responsibility of steering a project to success. This surely is a very hazy job description. In fact, it is very difficult to objectively describe the precise job responsibilities of a project manager. The job responsibilities of a project manager ranges from invisible activities like building up of team morale to highly visible customer presentations. Most managers take the responsibilities for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, managerial report writing and presentation, and interfacing with clients. These activities are certainly numerous and varied. We can still broadly classify these activities into two major types—project planning and project monitoring and control.

4 REQUIREMENTS ANALYSIS AND SPECIFICATION

All plan-driven life cycle models prescribe that before starting to develop a software, the exact requirements of the customer must be understood and documented. In the past, many projects have suffered because the developers started to implement something without determining whether they were building what the customers exactly wanted. Starting development work without properly understanding and documenting the requirements increases the number of iterative changes in the later life cycle phases, and thereby alarmingly pushes up the development costs. This also sets the ground for customer dissatisfaction and bitter customer-developer disputes and protracted legal battles. No wonder that experienced developers consider the requirements analysis and specification to be a very important phase of software development life cycle and undertake it with utmost care.

4.1 An overview of requirements analysis and specification phase

The requirements analysis and specification phase starts after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible.

The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed. The requirements specification document is usually called as the software requirements specification (SRS) document. The goal of the requirements analysis and specification phase can be stated in a nutshell as follows.

4.2 Who carries out requirements analysis and specification?

Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site. The engineers who gather a n d analyse customer requirements and then write the requirements specification document are known as system analysts in the software industry parlance. System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done. After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the software requirements specification (SRS) document.

5 SOFTWARE DESIGN

During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. We can state the main objectives of the design phase, in other words, as follows.

This view of a design process has been shown schematically in Figure 5.1. As shown in Figure 5.1, the design process starts using the SRS document and completes with the production of the design document. The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.

5.1 OVERVIEW OF THE DESIGN PROCESS

The design process essentially transforms the SRS document into a design document. In the following sections and subsections, we will discuss a few important issues associated with the design process.

5.1.1 Outcome of the Design Process

The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs. For example, in an academic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named handle student registration.

A control relationship between two modules essentially arises due to function calls across the two modules. The control relationships existing among various modules should be identified in the design document.

5.2 HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

5.2.1 Correctness

A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

5.2.2 Understandability

A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

6 FUNCTION-ORIENTED SOFTWARE DESIGN

Function-oriented design techniques were proposed nearly four decades ago. These techniques are at the present time still very popular and are currently being used in many software development organisations. These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software. These services provided by a software (e.g., issue book, serach book, etc., for a Library Automation Software to its users are also known as the high-level functions supported by the software. During the design process, these high-level functions are successively decomposed into more detailed functions.

6.1 OVERVIEW OF SA/SD METHODOLOGY

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:
• Structured analysis (SA)
• Structured design (SD)
• During structured analysis, the SRS document is transformed into a data flow diagram (DFD) model.
• During structured design, the DFD model is transformed into a structure chart.

6.2 STRUCTURED ANALYSIS

We have already mentioned that during structured analysis, the major processing tasks (high-level functions) of the system are analysed, and t h e data flow among these processing tasks are represented graphically. Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978]. The structured analysis technique is based on the following underlying principles:
• Top-down decomposition approach
• Application of divide and conquer principle. Through this each highlevel function is independently decomposed into detailed functions.

7 OBJECT MODELLING USING UML

In recent years, the object-oriented software development style has become very popular and is at present being widely used in industry as well as in academic circles. Since its inception in the early eighties, the object technology has made rapid progress. From a modest beginning in the early eighties, the advancements to the object technology gathered momentum in the nineties and the technology is now nearing maturity. Considering the widespread use and popularity of the object technology in both industry and academia, it is important to learn this technology well.

7.1 BASIC OBJECT-ORIENTATION CONCEPTS

The principles of object-orientation have been founded on a few simple concepts. These concepts are pictorially shown in Figure 7.1. After discussing these basic concepts, we examine a few related technical terms.

7.2 UNIFIED MODELLING LANGUAGE (UML)

As the name itself implies, UML is a language for documenting models. As is the case with any other language, UML has its syntax (a set of basic symbols and sentence formation rules) and semantics (meanings of basic symbols and sentences). It provides a set of basic graphical notations (e.g. rectangles, lines, ellipses, etc.) that can be combined in certain ways to document the design and analysis results.

It is important to remember that UML is neither a system design or development methodology by itself, nor is tied to any specific methodology. UML is merely a language for documenting models. Before the advent of UML, every design methodology not only prescribed entirely different design steps, but each was tied to some specific design modelling language.

8 OBJECT-ORIENTED SOFTWARE DEVELOPMENT

In this Chapter, we shall build upon the object-modelling concepts introduced in the last Chapter to discuss an object-oriented analysis and design (OOAD) methodology. We shall realise that object-oriented analysis and design (OOAD) techniques advocate a radically different approach compared to the traditional function-oriented design approaches. Recall our discussions in Chapters 5 and 6, where we had pointed out that the traditional function-oriented design approaches essentially suggest that while developing a system, all the functionalities that the system needs to support should be identified and implemented. In contrast, the OOAD paradigm suggests that the objects (i.e., entities) associated with a problem should be identified and implemented.

8.1 Object-oriented analysis (OOA) ver sus object-oriented design (OOD)

Before discussing the details of OOA and OOD, let us understand the primary differences between their intents and the performed activities. The term object-oriented analysis (OOA) refers to developing an initial model of a software product from an analysis of its requirements specification.

Analysis involves constructing a model (called the analysis model) by analysing and elaborating the user requirements documented in the SRS document rather than determining how to define a solution that can be easily implemented. While developing the analysis model, implementation-specific decisions (such as in what sequence the classes would invoke each others' methods, specific hardware used, database used, etc.) are avoided. Therefore, the analysis model remains valid, even if the implementation aspects change later. However, it is very difficult to directly translate an analysis model into code. On the other hand, a design model can be easily translated into code. At present, many computer aided software engineering (CASE) tools support automatic generation of code templates from design models, thereby greatly reducing the programmer's work.

8.2 An OOAD methodology

In this chapter, we shall discuss a generic methodology for developing object-oriented designs starting from initial problem descriptions. This methodology consists of first constructing a use case model from the initial problem analysis. Subsequently, the domain model is constructed. These two analysis models are iteratively refined into a design model. The design model can straight away be implemented using a programming language. It must, however, be kept in mind that though the design methodology that we shall discuss is easy to master, it is useful only to solve simple problems. However, once we are able to understand this simple design method, approaches for solving more complex problems can be comprehended with only incremental effort.

9 USER INTERFACE DESIGN

The user interface portion of a software product is responsible for all interactions with the user. Almost every software product has a user interface (can you think of a software product that does not have any user interface?). In the early days of computer, no software product had any user interface. The computers those days were batch systems and no interactions with the users were supported. Now, we know that things are very different—almost every software product is highly interactive. The user interface part of a software product is responsible for all interactions with the end-user. Consequently, the user interface part of any software product is of direct concern to the end-users. No wonder then that many users often judge a software product based on its user interface. Aesthetics apart, an interface that is difficult to use leads to higher levels of user errors and ultimately leads to user dissatisfaction. Users become particularly irritated when a system behaves in an unexpected ways, i.e., issued commands do not carry out actions according to the intuitive expectations of the user. Normally, when a user starts using a system, he builds a mental model of the system and expects the system behaviour to conform to it. For example, if a user action causes one type of system activity and response under some context, then the user would expect similar system activity and response to occur for similar user actions in similar contexts. Therefore, sufficient care and attention should be paid to the design of the user interface of any software product.

9.1 CHARACTERISTICS OF A GOOD USER INTERFACE

Before we start discussing anything about how to develop user interfaces, it is important to identify the different characteristics that are usually desired of a good user interface. Unless we know what exactly is expected of a good user interface, we cannot possibly design one. In the following subsections, we identify a few important characteristics of a good user interface:

Speed of learning: A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memorise commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following three issues are crucial to enhance the speed of learning.

9.2 BASIC CONCEPTS

In this section, we first discuss some basic concepts in user guidance and on-line help system. Next, we examine the concept of a mode-based and a modeless interface and the advantages of a graphical interface.

9.2.1 User Guidance and On-line Help

Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system. This is different from the guidance and error messages which are flashed automatically without the user asking for them. The guidance messages prompt the user regarding the options he has regarding the next command, and the status of the last command, etc

10 CODING AND TESTING

In this chapter, we will discuss the coding and testing phases of the software life cycle. In the coding phase, every module specified in the design document is coded and unit tested. During unit testing, each module is tested in isolation from other modules. That is, a module is tested independently as and when its coding is complete.

10.1 CODING

The input to the coding phase is the design document produced at the end of the design phase. Please recollect that the design document contains not only the high-level design of the system in the form of a module structure (e.g., a structure chart), but also the detailed design. The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified. During the coding phase, different modules identified in the design document are coded according to their respective module specifications. We can describe the overall objective of the coding phase to be the following.

10.1.1 Coding Standards and Guidelines

Good software development organisations usually develop their own coding standards and guidelines depending on what suits their organisation best and based on the specific types of software they develop. To give an idea about the types of coding standards that are being used, we shall only list some general coding standards and guidelines that are commonly adopted by many software development organisations, rather than trying to provide an exhaustive list.

10.2 CODE REVIEW

Testing is an effective defect removal mechanism. However, testing is applicable to only executable code. Review is a very effective technique to remove defects from source code. In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing. Over the years, review techniques have become extremely popular and have been generalised for use with other work products.

10.2.1 Code Walkthrough

Code walkthrough is an informal code analysis technique. In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated. A few members of the development team are given the code a couple of days before the walkthrough meeting. Each member selects some test cases and simulates execution of the code by hand (i.e., traces the execution through different statements and functions of the code).