Providing the basis of agreement and validation should be strong enough reasons for both the client and the developer to do a thorough and rigorous job of requirement understanding and specification, but there are other very practical and pressing reasons for having a good SRS.

Generally,many errors are made during the requirements phase. And an error in the SRS will manifest itself as an error in the final system implementing the SRS. Clearly, if we want a high-quality end product that has few errors, we must begin with a high-quality SRS. In other words, we can conclude that:

– A high-quality SRS is a prerequisite to high-quality software.

Finally, the quality of SRS has an impact on cost (and schedule) of the project. We know that errors can exist in the SRS. It is also known that the cost of fixing an error increases almost exponentially as time progresses [10, 12]. Hence, by improving the quality of requirements, we can have a huge savings in the future by having fewer expensive defect removals. In other words,

– A high-quality SRS reduces the development cost.

Requirement Process

The requirement process is the sequence of activities that need to be performed in the requirements phase and that culminate in producing a high-quality document containing the SRS. The requirements process typically consists of three basic tasks: problem or requirement analysis, requirements specification, and requirements validation.

Problem analysis often starts with a high-level "problem statement." During analysis the problem domain and the environment are modeled in an effort to understand the system behavior, constraints on the system, its inputs and outputs, etc. The basic purpose of this activity is to obtain a thorough understanding of what the software needs to provide. Frequently, during analysis, the analyst will have a series of meetings with the clients and end users. In the early meetings, the clients and end users will explain to the analyst about their work, their environment, and their needs as they perceive them. Any documents de- scribing the work or the organization may be given, along with outputs of the existing methods of performing the tasks. In these early meetings, the analyst is basically the listener, absorbing the information provided. Once the analyst understands the system to some extent, he uses the next few meetings to seek clarifications of the parts he does not understand. He may document the information or build some models, and he may do some brainstorming or thinking about what the system should do. In the final few meetings, the analyst essentially explains to the client what he understands the system should do and uses the meetings as a means of verifying if what he proposes the system should do is indeed consistent with the objectives of the clients. The understanding obtained by problem analysis forms the basis of requirements specification, in which the focus is on clearly specifying the requirements in a document. Issues such as representation, specification languages, and tools are addressed during this activity. As analysis produces large amounts of in- formation and knowledge with possible redundancies, properly organizing and describing the requirements is an important goal of this activity.

Requirements validation focuses on ensuring that what have been specified in the SRS are indeed all the requirements of the software and making sure that the SRS is of good quality. The requirements process terminates with the production of the validated SRS. We will discuss this more later in the chapter.

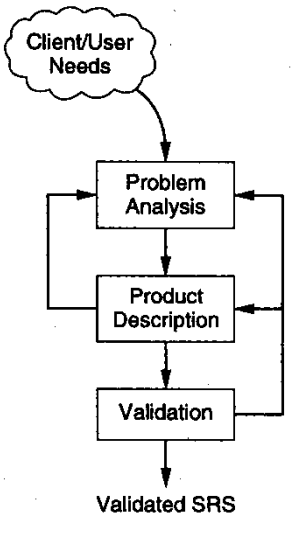


Figure 3.1: The requirement process.

It should be pointed out that the requirements process is not a linear se- quence of these three activities and there is considerable overlap and feedback between these activities. The overall requirement process is shown in Figure
3.1. As shown in the figure, from the specification activity we may go back to the analysis activity. This happens as frequently some parts of the problem are analyzed and then specified before other parts are analyzed and speci- fied. Furthermore, the process of specification frequently shows shortcomings in the knowledge of the problem, thereby necessitating further analysis. Once the specification is done, it goes through the validation activity. This activity may reveal problems in the specification itself, which requires going back to the specification step, or may reveal shortcomings in the understanding of the problem, which requires going back to the analysis activity.

Requirements Specification

The final output is the SRS document. As analysis precedes specification, the first question that arises is: If formal modeling is done during analysis, why are the outputs of modeling not treated as an SRS? The main reason is that modeling generally focuses on the problem structure, not its external behavior. Consequently, things like user interfaces are rarely modeled, whereas they fre- quently form a major component of the SRS. Similarly, for ease of modeling, frequently "minor issues" like erroneous situations (e.g., error in output) are rarely modeled properly, whereas in an SRS, behavior under such situations also has to be specified. Similarly, performance constraints, design constraints, standards compliance, recovery, etc., are not included in the model, but must be specified clearly in the SRS because the designer must know about these to properly design the system. It should therefore be clear that the outputs of a model cannot form a desirable SRS.

Desirable Characteristics of an SRS

To properly satisfy the basic goals, an SRS should have certain properties and should contain different types of requirements. Some of the desirable charac- teristics of an SRS are [53]:

1. Correct
2. Complete

3. Unambiguous

4. Verifiable

5. Consistent

6. Ranked for importance and/or stability

An SRS is correct if every requirement included in the SRS represents something required in the final system. It is complete if everything the software is supposed to do and the responses of the software to all classes of input data are specified in the SRS. It is unambiguous if and only if every requirement stated has one and only one interpretation. Requirements are often written in natural language, which is inherently ambiguous. If the requirements are specified in a natural language, the SRS writer has to be especially careful to ensure that there are no ambiguities.

An SRS is verifiable if and only if every stated requirement is verifiable. A requirement is verifiable if there exists some cost-effective process that can check whether the final software meets that requirement. Generally, all the requirements for software are not of equal importance. Some are critical, others are important but not critical, and there are some which are desirable but not very important. Similarly, some requirements are "core" requirements which are not likely to change as time passes, while others are more dependent on time. completeness is perhaps the most important and also the most difficult property to establish. One of the most common defects in requirements specification is incompleteness. Missing requirements necessitate additions and modifications to the requirements later in the development cycle, which are often expensive to incorporate.

For iterative development, as feedback is possible and opportunity for change is also there, the specification can be less detailed. And if an agile approach is being followed, then completeness should be sought only for top-level requirements, as details may not be required in written form, and are elicited when the requirement is being implemented. Together the performance and interface requirements and design constraints can be called nonfunctional requirements.

COMPONENTS OF AN SRS

Completeness of specifications is difficult to achieve and even more difficult to verify. Having guidelines about what different things an SRS should specify will help in completely specifying the requirements. The basic issues an SRS must address are:

– Functionality

– Performance

– Design constraints imposed on an implementation

– External interfaces

Functional requirements specify the expected behavior of the system—which outputs should be produced from the given inputs. They describe the relation- ship between the input and output of the system. An important part of the specification is the system behavior in abnormal situations, like invalid input (which can occur in many ways) or error during computation. The functional requirement must clearly state what the system should do if such situations occur.

The **performance requirements** part of an SRS specifies the performance constraints on the software system. All the requirements relating to the per- formance characteristics of the system must be clearly specified. There are two types of performance requirements: static and dynamic.

Static requirements are those that do not impose constraint on the execution characteristics of the system. These include requirements like the number of terminals to be supported, the number of simultaneous users to be supported, and the number of files that the system has to process and their sizes. These are also called capacity requirements of the system.

Dynamic requirements specify constraints on the execution behavior of the system. These typically include response time and throughput constraints on the system. Response time is the expected time for the completion of an op- eration under specified circumstances. Throughput is the expected number of operations that can be performed in a unit time. For example, the SRS may specify the number of transactions that must be processed per unit time, or what the response time for a particular command should be. Acceptable ranges of the different performance parameters should be specified, as well as accept- able performance for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms. Require- ments such as "response time should be good" or the system must be able to "process all the transactions quickly" are not desirable because they are impre- cise and not verifiable. Instead, statements like "the response time of command x should be less than one second 90% of the times" or "a transaction should be processed in less than one second 98% of the times" should be used to declare performance specifications.

An SRS should identify and specify all such constraints. Some examples of these are:

Standards Compliance: This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures. There may be audit requirements which may require logging of operations.

Hardware Limitations: The software may have to operate on some ex- isting or predetermined hardware, thus imposing restrictions on the design. Hardware limitations can include the type of machines to be used, operating system available on the system, languages supported, and limits on primary and secondary storage.

Reliability and Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed, as they make the system more complex and expensive. Recovery requirements are often an integral part here, detailing what the system should do if some failure occurs to ensure certain properties.

Security: Security requirements are becoming increasingly important. These requirements place restrictions on the use of certain commands, control access to data, provide different kinds of access requirements for different people, re- quire the use of passwords and cryptography techniques, and maintain a log of activities in the system. They may also require proper assessment of security threats, proper programming techniques, and use of tools to detect flaws like buffer overflow.

In the external interface specification part, all the interactions of the soft- ware with people, hardware, and other software should be clearly specified. For the user interface, the characteristics of each user interface of the software prod- uct should be specified. User interface is becoming increasingly important and must be given proper attention. A preliminary user manual should be created with all user commands, screen formats, an explanation of how the system will appear to the user, and feedback and error messages. Like other specifications, these requirements should be precise and verifiable. So, a statement like "the system should be user friendly" should be avoided and statements like "com- mands should be no longer than six characters" or "command names should reflect the function they perform" used.

For hardware interface requirements, the SRS should specify the logical characteristics of each interface between the software product and the hard- ware components. If the software is to execute on existing hardware or on pre- determined hardware, all the characteristics of the hardware, including memory restrictions, should be specified.

The interface requirement should specify the interface with other software the system will use or that will use the system. This includes the interface with the operating system and other applications. The message content and format of each interface should be specified.

Structure of a Requirements Document

Requirements have to be specified using some specification language. Though formal notations exist for specifying specific properties of the system, natural languages are now most often used for specifying requirements. When formal languages are employed, they are often used to specify particular properties or for specific parts of the system, as part of the overall SRS.

The introduction section contains the purpose, scope, overview, etc., of the requirements document. The key aspect here is to clarify the motivation and business objectives that are driving this project, and the scope of the project. The next section gives an overall perspective of the system—how it fits into the larger system, and an overview of all the requirements of this system. Detailed requirements are not mentioned. Product perspective is essentially the relationship of the product to other products; defining if the product is independent or is a part of a larger product, and what the principal interfaces of the product are. A general abstract description of the functions to be performed by the product is given. Schematic diagrams showing a general view of different functions and their relationships with each other can often be useful. Similarly, typical characteristics of the eventual end user and general constraints are also specified.

1. Introduction
- **1.1 Purpose**
- **1.2 Scope**
- **1.3 Definitions, Acronyms, and Abbreviations**
- **1.4 References**
- **1.5 Overview**

2. Overall Description
- **2.1 Product Perspective**
- **2.2 Product Functions**
- **2.3 User Characteristics**
- **2.4 General Constraints**
- **2.5 Assumptions and Dependencies**

3. Detailed Requirements
- **3.1 External Interface Requirements**
  - **3.1.1 User Interfaces**
  - **3.1.2 Hardware Interfaces**
  - **3.1.3 Software Interfaces**
  - **3.1.4 Communication Interfaces**
- **3.2. Functional Requirements**
  - **3.2.1 Mode 1**
    - **3.2.1.1 Functional Requirement 1.1**
    - **:**
    - **3.2.1.n Functional Requirement 1.n**
  - **:**
  - **3.2.m Mode m**

3.2.m.1 Functional Requirement m.1

:

3.2.m.n Functional Requirement m.n

3.3 Performance Requirements

3.4 Design Constraints

3.5 Attributes

3.6 Other Requirements

Figure 3.2: General structure and detailed structure of an SRS

The performance section should specify both static and dynamic perfor- mance requirements. All factors that constrain the system design are described in the performance constraints section. The attributes section specifies some of the overall attributes that the system should have. Any requirement not covered under these is listed under other requirements. Design constraints specify all the constraints imposed on design (e.g., security, fault tolerance, and standards compliance).

Functional Specification with Use Cases

Functional requirements often form the core of a requirements document. The traditional approach for specifying functionality is to specify each function that the system should provide. Use cases specify the functionality of a system by specifying the behavior of the system, captured as interactions of the users with the system. Use cases can be used to describe the business processes of the larger business or organization that deploys the software, or it could just describe the behavior of the software system. We will focus on describing the behavior of software systems that are to be built.

Though use cases are primarily for specifying behavior, they can also be used effectively for analysis. Later when we discuss how to develop use cases, we will discuss how they can help in eliciting requirements also.Use cases drew attention after they were used as part of the object-oriented modeling approach proposed by Jacobson

Basics

A software system (in our case whose requirements are being uncovered) may be used by many users, or by other systems. In use case terminology, an actor is a person or a system which uses the system for achieving some goal. Note that as an actor interacts for achieving some goal, it is a logical entity that represents a group of users (people or system) who behave in a similar manner. Different actors represent groups with different goals. So, it is better to have a “receiver” and a “sender” actor rather than having a generic “user” actor for a system in which some messages are sent by users and received by some other users.

A primary actor is the main actor that initiates a use case (UC) for achieving a goal, and whose goal satisfaction is the main objective of the use case. The primary actor is a logical concept and though we assume that the primary actor executes the use case, some agent may actually execute it on behalf of the primary actor. For example, a VP may be the primary actor for get sales growth report by region use case, though it may actually be executed by an assistant.

Table 3.1: Use case terms.

Term	Definition

Actor	A person or a system which uses the system being built for achieving some goal.
Primary actor	The main actor for whom a use case is initiated and whose goal satisfaction is the main objective of the use case.
Scenario	A set of actions that are performed to achieve a goal under some specified conditions.
Main success scenario	Describes the interaction if nothing fails and all steps in the scenario succeed.
Extension scenario	Describes the system behavior if some of the steps in the main scenario do not complete successfully.

Developing Use Cases

UCs not only document requirements, as their form is like storytelling and uses text, both of which are easy and natural with different stakeholders, they also are a good medium for discussion and brainstorming. Hence, UCs can also be used for requirements elicitation and problem analysis. While developing use cases, informal or formal models may also be built, though they are not required.

UCs can be evolved in a stepwise refinement manner with each step adding more details. This approach allows UCs to be presented at different levels of ab- straction. Though any number of levels of abstraction are possible, four natural levels emerge:

– Actors and goals. The actor-goal list enumerates the use cases and specifies the actors for each goal. (The name of the use case is generally the goal.) This table may be extended by giving a brief description of each of the use cases. At this level, the use cases together specify the scope of the system and give an overall view of what it does. Completeness of functionality can be assessed fairly well by reviewing these.

– Main success scenarios. For each of the use cases, the main success scenar- ios are provided at this level. With the main scenarios, the system behavior for each use case is specified. This description can be reviewed to ensure that interests of all the stakeholders are met and that the use case is delivering the desired behavior.

– Failure conditions. Once the success scenario is listed, all the possible failure conditions can be identified. At this level, for each step in the main success scenario, the different ways in which a step can fail form the failure conditions. Before deciding what should be done in these failure conditions (which is done at the next level), it is better to enumerate the failure condi- tions and review for completeness.

– Failure handling. This is perhaps the most tricky and difficult part of writing a use case. Often the focus is so much on the main functionality that people do not pay attention to how failures should be handled. Determining what should be the behavior under different failure conditions will often identify new business rules or new actors.

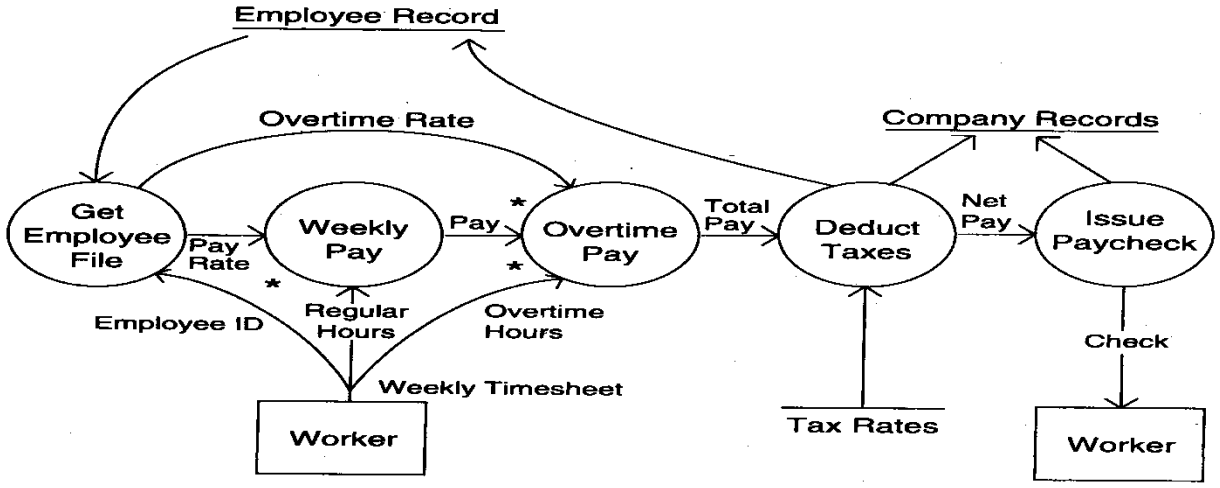
The different levels can be used for different purposes. For discussion on overall functionality or capabilities of the system, actors and goal-level descrip- tion is very useful. Failure conditions, on the

other hand, are very useful for understanding and extracting detailed requirements and business rules under special

Other Approaches for Analysis

Data Flow Diagrams

Data flow diagrams (also called data flow graphs) are commonly used during problem analysis. Data flow diagrams (DFDs) are quite general and are not limited to problem analysis for software requirements specification. They were in use long before the software engineering discipline began. DFDs are very useful in understanding a system and can be effectively used during analysis.



In a DFD, data flows are identified by unique names. These names are chosen so that they convey some meaning about what the data is. However, for specifying the precise structure of data flows, a data dictionary is often used. The associated data dictionary states precisely the structure of each data flow in the DFD. To define the data structure, a regular expression type notation is used. While specifying the structure of a data item, sequence or composition is represented by "+", selection by vertical bar "|" (means one OR the other), and repetition by "*".

3.5.2 ER Diagrams

Entity relationship diagrams (ERDs) have been used for years for modeling the data aspects of a system. An ERD can be used to model the data in the system and how the data items relate to each other, but does not cover how the data is to be processed or how the data is actually manipulated and changed in the system. It is used often by database designers to represent the structure of the database and is a useful tool for analyzing software systems which employ databases. ER models form the logical database design and can be easily converted into initial table structure for a relational database.

ER diagrams have two main concepts and notations to representing them. These are entities and relationships. Entities are the main information holders or concepts in a system. Entities can be viewed as types which describe all elements of some type which have common properties. Entities are represented as boxes in an ER diagram, with a box representing all instances of the concept or type the entity is representing. An entity is essentially equivalent to a table in a database or a sheet in a spreadsheet, with each row representing an instance of this entity. Entities may have attributes, which are properties of the concept being represented. Attributes can be viewed as the columns of the database table and are represented as ellipses attached to the entity. To avoid cluttering, attributes are sometimes not shown in an ER diagram.

Let us draw the ER diagram for the university auction system, some use cases of which were discussed earlier. From the use cases described, we can easily identify some entities—users, categories, items, and bids. The relationships between them are also clear. A user can sell many items, but each item has only one seller, so there is a one-to-many relationship "Sell" between the user and items. Similarly, there is a one-to-many relationship between

items and bids, between users and bids, and between categories and items. The ER diagram of this is shown in Figure 3.7.

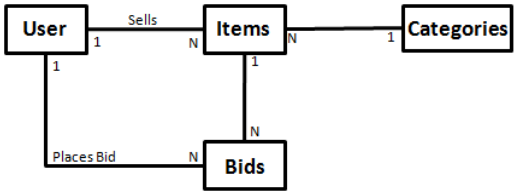


Figure 3.7: ER diagram for an auction system.

From the ER diagram, it is easy to determine the initial logical structure of the tables. Each entity represents a table, and relationships determine what fields a table must have to support the relationship, in addition to having fields for each of the attributes. For example, from the ER diagram for the auction system, we can say that there will be four tables for users, categories, items, and bids. As user is related to item by one-to-many, the item table should have a user-ID field to uniquely identify the user who is selling the item. Similarly, the bid table must have a user-ID to identify the user who placed the bid, and an item-ID to identify the item for which the bid has been made.

Validation

The development of software starts with a requirements document, which is also used to determine eventually whether or not the delivered software system is acceptable. It is therefore important that the requirements specification contains no errors and specifies the client's requirements correctly.

Before we discuss validation, let us consider the type of errors that typically occur in an SRS. Many different types of errors are possible, but the most common errors that occur can be classified in four types: **omission, inconsistency, incorrect fact, and ambiguity**. Omission is a common error in requirements. In this type of error, some user requirement is simply not included in the SRS; the omitted requirement may be related to the behavior of the system, its performance, constraints, or any other factor. Omission directly affects the external completeness of the SRS. Another common form of error in requirements is inconsistency. Inconsistency can be due to contradictions within the requirements themselves or to incompatibility of the stated requirements with the actual requirements of the client or with the environment in which the system will operate. The third common requirement error is incorrect fact. Errors of this type occur when some fact recorded in the SRS is not correct. The fourth common error type is ambiguity. Errors of this type occur when there are some requirements that have multiple meanings, that is, their interpretation is not unique.

Requirements review is a review by a group of people to find errors and point out other matters of concern in the requirements specifications of a system. The review group should include the author of the requirements document, someone who understands the needs of the client, a person of the design team, and the person(s) responsible for maintaining the requirements document. It is also good practice to include some people not directly involved with product development, like a software quality engineer.

Although the primary goal of the review process is to reveal any errors in the requirements, such as those discussed earlier, the review process is also used to consider factors affecting quality, such as testability and readability. During the review, one of the jobs of the reviewers is to uncover the requirements that are too subjective and too difficult to define criteria for testing that requirement. During the review, the review team must go through each requirement and if any errors are there, then they discuss and agree on the nature of the error.

Planning a Software Project

Effort Estimation: effort and schedule estimates are essential prerequisites for planning the project. These estimates are needed before development is initiated, as they establish the cost and schedule goals of the project. Without these, even simple questions like "is the project late?" "are there cost overruns?" and "when is the project likely to complete?" cannot be answered. Effort and schedule estimates are also required for determining the staffing level for a project during different phases, for the detailed plan, and for project monitoring.

The accuracy with which effort can be estimated clearly depends on the level of information available about the project. The more detailed the infor- mation, the more accurate the estimation can be. Of course, even with all the information available, the accuracy of the estimates will depend on the effec- tiveness and accuracy of the estimation procedures or models employed and the process. If from the requirements specifications, the estimation approach can produce estimates that are within 20% of the actual effort about two-thirds of the time, then the approach can be considered good.

Top-Down Estimation Approach

Although the effort for a project is a function of many parameters, it is generally agreed that the primary factor that controls the effort is the size of the project. That is, the larger the project, the greater is the effort requirement. The top- down approach utilizes this and considers effort as a function of project size. Note that to use this approach, we need to first determine the nature of the function, and then to apply the function, we need to estimate the size of the project for which effort is to be estimated.

A more general function for determining effort from size that is commonly used is of the form:

$$EFFORT = a * SIZE^b,$$

where a and b are constants [2], and project size is generally in KLOC (size could also be in another size measure called function points which can be de- termined from requirements).

Though size is the primary factor affecting cost, other factors also have some effect. In the COCOMO model, after determining the initial estimate, some other factors are incorporated for obtaining the final estimate. To do this, COCOMO uses a set of 15 different attributes of a project called cost driver attributes. Examples of the attributes are required software reliability, product complexity, analyst capability, application experience, use of modern tools, and required development schedule. Each cost driver has a rating scale, and for each rating, a multiplying factor is provided. For example, for the reliability, the rating scale is very low, low, nominal, high, and very high; the multiplying factors for these ratings are .75, .88, 1.00, 1.15, and 1.40, respectively. So, if the reliability requirement for the project is judged to be low, then the multiplying factor is .75, while if it is judged to be very high, the factor is 1.40. The attributes and their multiplying factors for different ratings are shown in Table 4.1 [12, 13].

Table 4.1: Effort multipliers for different cost drivers.

Cost Drivers	Rating				
	Very Low	Low	Nom-inal	High	Very High
Product Attributes					
RELY, required reliability	.75	.88	1.00	1.15	1.40
DATA, database size		.94	1.00	1.08	1.16
CPLX, product complexity	.70	.85	1.00	1.15	1.30
Computer Attributes					
TIME, execution time constraint			1.00	1.11	1.30
STOR, main storage constraint			1.00	1.06	1.21
VITR, virtual machine volatility		.87	1.00	1.15	1.30
TURN, computer turnaround time		.87	1.00	1.07	1.15
Personnel Attributes					
ACAP, analyst capability	1.46	1.19	1.00	.86	.71
AEXP, application exp.	1.29	1.13	1.00	.91	.82
PCAP, programmer capability	1.42	1.17	1.00	.86	.70
VEXP, virtual machine exp.	1.21	1.10	1.00	.90	
LEXP, prog. language exp.	1.14	1.07	1.00	.95	
Project Attributes					
MODP, modern prog. practices	1.24	1.10	1.00	.91	.82
TOOL, use of SW tools	1.24	1.10	1.00	.91	.83
SCHED, development schedule	1.23	1.08	1.00	1.04	1.10

The multiplying factors for all 15 cost drivers are multiplied to get the effort adjustment factor (EAF). The final effort estimate, E, is obtained by multiplying the initial estimate by the EAF. In other words, adjustment is made to the size-based estimate using the rating for these 15 different factors.

we want to

use COCOMO for estimation, we should estimate the value of the different cost drivers. Suppose we expect that the complexity of the system is high, the programmer capability is low, and the application experience of the team is low. All other factors have a nominal rating. From these, the effort adjustment factor (EAF) is

$$EAF = 1.15 * 1.17 * 1.13 = 1.52.$$

The initial effort estimate for the project is obtained from the relevant equa- tions. We have

$$Ei = 3.9 * 2^{.91} = 7.3 \text{ PM}.$$

Using the EAF, the adjusted effort estimate is

$$E = 1.52 * 7.3 = 11.1 \text{ PM}.$$

From the overall estimate, estimates of the effort required for the different phases in the projects can also be determined. This is generally done by using an effort distribution among phases. The percentage of total effort spent in a phase varies with the type and size of the project, and can be obtained from data of similar projects in the past. A general distribution of effort among different phases was discussed in the previous chapter.

Bottom-Up Estimation Approach

A somewhat different approach for effort estimation is the bottom-up approach. In this approach, the project is first divided into tasks and then estimates for the different tasks of the project are obtained. From the estimates of the different tasks, the overall estimate is determined. That is, the overall estimate of the project is derived from the estimates of its parts. This type of approach is also called activity-based estimation. Essentially, in this approach the size and complexity of the project is captured in the set of tasks the project has to perform.

The bottom-up approach lends itself to direct estimation of effort; once the project is partitioned into smaller tasks, it is possible to directly estimate the effort required for them, especially if tasks are relatively small. One difficulty in this approach is that to get the overall estimate, all the tasks have to

be enumerated. A risk of bottom-up methods is that one may omit some activities. Also, directly estimating the effort for some overhead tasks, such as project management, that span the project can be difficult.

Project Schedule and Staffing

After establishing a goal on the effort front, we need to establish the goal for delivery schedule. With the effort estimate (in person-months), it may be tempting to pick any project duration based on convenience and then fix a suitable team size to ensure that the total effort matches the estimate. However, as is well known now, person and months are not fully interchangeable in a software project. Person and months can be interchanged arbitrarily only if all the tasks in the project can be done in parallel, and no communication is needed between people performing the tasks. a project with some estimated effort, multiple schedules (or project duration) are indeed possible. For example, for a project whose effort estimate is 56 person-months, a total schedule of 8 months is possible with 7 people. A schedule of 7 months with 8 people is also possible, as is a schedule of approximately 9 months with 6 people. (But a schedule of 1 month with 56 people is not possible. Similarly, no one would execute the project in 28 months with 2 people.) In other words, once the effort is fixed, there is some flexibility in setting the schedule by appropriately staffing the project, but this flexibility is not unlimited.
the staffing level is not changed continuously in a project and ap- proximations of the Rayleigh curve are used: assigning a few people at the start, having the peak team during the coding phase, and then leaving a few people for integration and system testing. If we consider design and analysis, build, and test as three major phases, the manpower ramp-up in projects typically resembles the function shown in Figure 4.1

For ease of scheduling, particularly for smaller projects, often the required people are assigned together around the start of the project. This approach can lead to some people being unoccupied at the start and toward the end. This slack time is often used for supporting project activities like training and documentation.

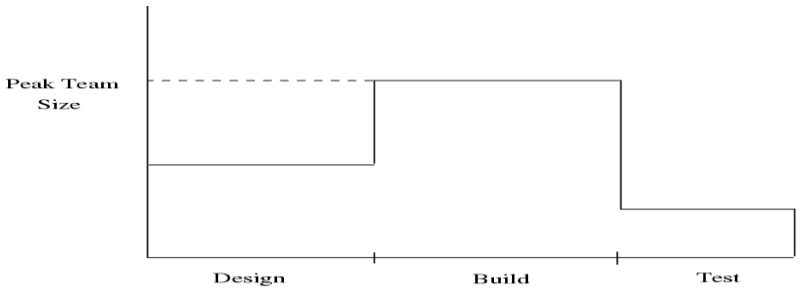


Figure 4.1: Manpower ramp-up in a typical project.

Quality Planning

Having set the goals for effort and schedule, the goal for the third key dimension of a project—quality—needs to be defined. However, unlike schedule and effort, quantified quality goal setting for a project and then planning to meet it is much harder. For effort and schedule goals, we can easily check if a detailed plan meets these goals (e.g., by seeing if the last task ends before the target date and if the sum total of effort of all tasks is less than the overall effort goal). For quality, even if we set the goal in terms of expected delivered defect density, it is not easy to plan for achieving this goal or for checking if a plan can meet these goals. Hence, often, quality goals are specified in terms of acceptance criteria— the delivered software should finally work for all the situations and test cases in the acceptance criteria. Having set the goals for effort and schedule, the goal for the third key dimension of a project—quality—needs to be defined. However, unlike schedule and effort, quantified quality goal setting for a project and then planning to meet it is much harder. For effort and schedule goals, we can easily check if a detailed plan meets these goals (e.g., by seeing if the last task ends before the target date and if the sum total of effort of all tasks is less than the overall effort goal). For quality, even if we set the goal in terms of expected delivered defect density, it is not easy

to plan for achieving this goal or for checking if a plan can meet these goals. Hence, often, quality goals are specified in terms of acceptance criteria— the delivered software should finally work for all the situations and test cases in the acceptance criteria.

Software development is a highly people-oriented activity and hence it is error-prone. In a software project, we start with no defects (there is no soft- ware to contain defects). Defects are injected into the software being built during the different phases in the project. That is, during the transformation from user needs to software to satisfy those needs, defects are injected in the transformation activities undertaken. These injection stages are primarily the requirements specification, the high-level design, the detailed design, and cod- ing. To ensure that high-quality software is delivered, these defects are removed through the quality control (QC) activities. The QC activities for defect re- moval include requirements reviews, design reviews, code reviews, unit testing, integration testing, system testing, acceptance testing, etc. Figure 4.2 shows the process of defect injection and removal.

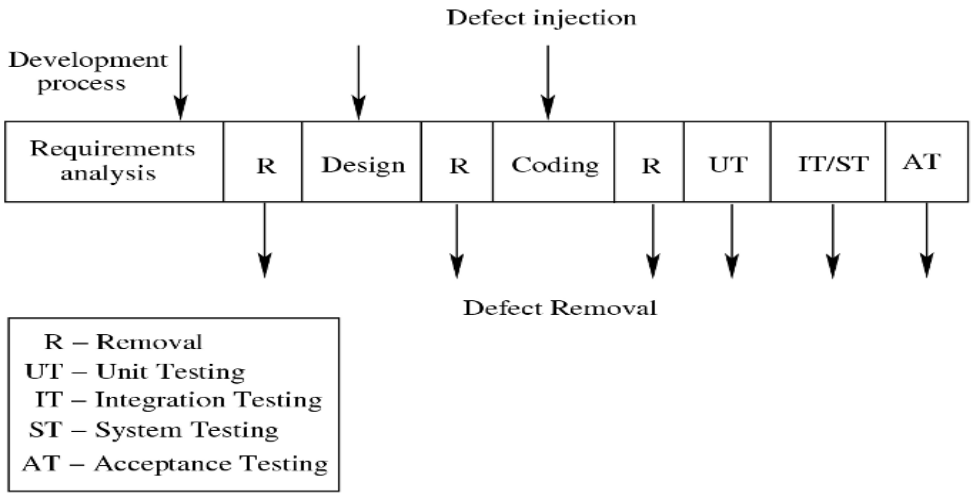


Figure 4.2: Defect injection and removal cycle.

As the final goal is to deliver software with low defect density, ensuring quality revolves around two main themes: reduce the defects being injected, and increase the defects being removed. The first is often done through standards, methodologies, following of good processes, etc., which help reduce the chances of errors by the project personnel. (There are specific techniques for defect prevention also.) The quality plan therefore focuses mostly on planning suitable quality control tasks for removing defects.

Risk Management Planning:

A software project is a complex undertaking. Unforeseen events may have an adverse impact on a project's ability to meet the cost, schedule, or quality goals. Risk management is an attempt to minimize the chances of failure caused by unplanned events. The aim of risk management is not to avoid getting into projects that have risks but to minimize the impact of risks in the projects that are undertaken.

A risk is a probabilistic event—it may or may not occur. For this reason, we frequently have an optimistic tendency to simply not see risks or to hope that they will not occur.

c

Table 4.3: Risk management plan for a project.

	Risk	Prob.	Impact	Exp.	Mitigation Plan
1	Failure to meet the high performance	High	High	High	Study white papers and guidelines on perf. Train team on perf. tuning. Update review checklist to look for perf. pitfalls. Test application for perf. during system testing.
2	Lack of people with right skills	Med	Med	Med	Train resources. Review prototype with customer. Develop coding practices.
3	Complexity of application	Med	Med	Med	Ensure ongoing knowledge transfer. Deploy persons with prior experience with the domain.
4	Manpower attrition	Med	Med	Med	Train a core group of four people. Rotate assignments among people. Identify backups for key roles.
5	Unclear requirements	Med	Med	Med	Review a prototype. Conduct a midstage review.

Risk Management Concepts

Risk is defined as an exposure to the chance of injury or loss. That is, risk implies that there is a possibility that something negative may happen Risk management can be considered as dealing with the possibility and actual occurrence of those events that are not "regular" or commonly expected, that is, they are probabilistic. The commonly expected events, such as people going on leave or some requirements changing, are handled by normal project management. So, in a sense, risk management begins where normal project management ends. It deals with events that are infrequent, somewhat out of the control of the project management, and which can have a major impact on the project.

Risk Assessment

The goal of risk assessment is to prioritize the risks so that attention and resources can be focused on the more risky items. Risk identification is the first step in risk assessment, which identifies all the different risks for a particular project. These risks are project-dependent and identifying them is an exercise in envisioning what can go wrong. Methods that can aid risk identification include checklists of possible risks, surveys, meetings and brainstorming, and reviews of plans, processes, and work products

	Risk Item	Risk Management Techniques
1	Personnel Shortfalls	Staffing with top talent; Job matching; Team building; Key personnel agreements; Training; Prescheduling key people
2	Unrealistic Schedules and Budgets	Detailed cost and schedule estimation; Design to cost; Incremental development; Software reuse; Requirements scrubbing
3	Developing the Wrong Software Functions	Organization analysis; Machine analysis; User surveys; Prototyping; Early user's manuals
4	Developing the Wrong User Interface	Prototyping; Scenarios; Task analysis; User characterization
5	Gold Plating	Requirements scrubbing; Prototyping; Cost benefit analysis; Design to cost

Figure 4.3: Top risk items and techniques for managing them.

Once the probabilities of risks materializing and losses due to materializa- tion of different risks have been analyzed, they can be prioritized. One approach for prioritization is through the concept of risk exposure (RE) [11], which is sometimes called risk impact. RE is defined by the relationship RE = P rob(U O) ∗ Loss(U O),

where P rob(U O) is the probability of the risk materializing (i.e., undesirable outcome) and Loss(U O) is the total loss incurred due to the unsatisfactory outcome. The loss is not only the direct financial loss that might be incurred but also any loss in terms of credibility, future business, and loss of property or life. The RE is the expected value of the loss due to a particular risk. For risk prioritization using RE is, the higher the RE, the higher the priority of the risk item.

Risk Control

The main objective of risk management is to identify the top few risk items and then focus on them. Once a project manager has identified and prioritized the risks, the top risks can be easily identified. The question then becomes what to do about them. Knowing the risks is of value only if you can prepare a plan so that their consequences are minimal—that is the basic goal of risk management.the strategy is to perform the actions that will either reduce the probability of the risk materializing or reduce the loss due to the risk materializing. These are called risk mitigation steps.

Risk prioritization and consequent planning are based on the risk perception at the time the risk analysis is performed. Because risks are probabilistic events that frequently depend on external factors, the threat due to risks may change with time as factors change. Clearly, then, the risk perception may also change with time.

Project Monitoring Plan

A project management plan is merely a document that can be used to guide the execution of a project. Even a good plan is useless unless it is properly executed. And execution cannot be properly driven by the plan unless it is monitored carefully and the actual performance is tracked against the plan.

Monitoring requires measurements to be made to assess the situation of a project. If measurements are to be taken during project execution, we must plan carefully regarding what to measure, when to measure, and how to measure. Hence, measurement planning is a key element in project planning. In addition, how the measurement data will be analyzed and reported must also be planned in advance to avoid the situation of collecting data but not knowing what to do with it. Without careful planning for data collection and

its analysis, neither is likely to happen. In this section we discuss the issues of measurements and project tracking.

Measurements

The basic purpose of measurements in a project is to provide data to project management about the project's current state, such that they can effectively monitor and control the project and ensure that the project goals are met. As project goals are established in terms of software to be delivered, cost, schedule, and quality, for monitoring the state of a project, size, effort, schedule, and defects are the basic measurements that are needed. Schedule is one of the most important metrics because most projects are driven by schedules and deadlines. Only by monitoring the actual schedule can we properly assess if the project is on time or if there is a delay.

Effort is the main resource consumed in a software project. Consequently, tracking of effort is a key activity during monitoring; it is essential for evaluating whether the project is executing within budget. For effort data some type of timesheet system is needed where each person working on the project enters the amount of time spent on the project. For better monitoring, the effort spent on various tasks should be logged separately.

Size is another fundamental metric because it represents progress toward delivering the desired functionality, and many data (for example, delivered de- fect density) are normalized with respect to size. The size of delivered software can be measured in terms of LOC (which can be determined through the use of regular editors and line counters) or function points. At a more gross level, just the number of modules or number of features might suffice.

For effective monitoring, a project must plan for collecting these measure- ments. Most often, organizations provide tools and policy support for recording this basic data, which is then available to project managers for tracking.

Project Monitoring and Tracking

The main goal of project managers for monitoring a project is to get visibility into the project execution so that they can determine whether any action needs to be taken to ensure that the project goals are met. As project goals are in terms of effort, schedule, and quality, the focus of monitoring is on these aspects. Different levels of monitoring might be done for a project. The three main levels of monitoring are activity level, status reporting, and milestone analysis. Measurements taken on the project are employed for monitoring.

Status reports are often prepared weekly to take stock of what has happened and what needs to be done. Status reports typically contain a summary of the activities successfully completed since the last status report, any activities that have been delayed, any issues in the project that need attention, and if everything is in place for the next week.

Detailed Scheduling

The activities discussed so far result in a project management plan document that establishes the project goals for effort, schedule, and quality; and defines the approach for risk management, ensuring quality, and project monitoring. Now this overall plan has to be translated into a detailed action plan which can then be followed in the project, and which, if followed, will lead to a successful project. That is, we need to develop a detailed plan or schedule of what to do when such that following this plan will lead to delivering the software with expected quality and within cost and schedule.

For the detailed schedule, the major phases identified during effort and schedule estimation, are broken into small schedulable activities in a hierarchi- cal manner. For example, the detailed design phase can be broken into tasks for developing the detailed design for each module, review of each detailed design, fixing of defects found, and so on. For each detailed task, the project manager estimates the time required

to complete it and assigns a suitable resource so that the overall schedule is met, and the overall effort also matches.

At each level of refinement, the project manager determines the effort for the overall task from the detailed schedule and checks it against the effort estimates. If this detailed schedule is not consistent with the overall schedule and effort estimates, the detailed schedule must be changed.

For detailed scheduling, tools like Microsoft Project or a spreadsheet can be very useful. For each lowest-level activity, the project manager specifies the effort, duration, start date, end date, and resources. Dependencies between activities, due either to an inherent dependency (for example, you can conduct a unit test plan for a program only after it has been coded) or to a resource- related dependency (the same resource is assigned two tasks), may also be specified. From these tools the overall effort and schedule of higher-level tasks can be determined.

A detailed project schedule is never static. Changes may be needed because the actual progress in the project may be different from what was planned, because newer tasks are added in response to change requests, or because of other unforeseen situations. Changes are done as and when the need arises.

The final schedule, frequently maintained using some suitable tool, is often the most "live" project plan document. During the project, if plans must be changed and additional activities must be done, after the decision is made, the changes must be reflected in the detailed schedule, as this reflects the tasks actually planned to be performed.

Software Architecture

Role of Software Architecture

What is architecture? Generally speaking, architecture of a system provides a very high level view of the parts of the system and how they are related to form the whole system. That is, architecture partitions the system in logical parts such that each part can be comprehended independently, and then describes the system in terms of these parts and the relationship between these parts.

Any complex system can be partitioned in many different ways, each pro- viding a useful view and each having different types of logical parts. The same holds true for a software system—there is no unique structure of the system that can be described by its architecture; there are many possible structures.

An architecture description of a system will therefore describe the different structures of the system. The next natural question is why should a team building a software system for some customer be interested in creating and documenting the structures of the proposed system. Some of the important uses that software architecture descriptions play are [6, 23, 54]:

1. Understanding and communication. An architecture description is primar- ily to communicate the architecture to its various stakeholders, which in- clude the users who will use the system, the clients who commissioned the system, the builders who will build the system, and, of course, the archi- tects. Through this description the stakeholders gain an understanding of some macro properties of the system and how the system intends to ful- fill the functional and quality requirements.
2. 2. Reuse. The software engineering world has, for a long time, been working toward a discipline where software can be assembled from parts that are developed by different people and are available for others to use. If one wants to build a software product in which existing components may be reused, then architecture becomes the key point at which reuse at the high- est level is decided.
3. Construction and Evolution. As architecture partitions the system into parts, some architecture-provided partitioning can naturally be used for constructing the system, which also requires that the system be broken into parts such that different teams (or individuals) can separately work on different parts. A suitable partitioning in the architecture can provide the project with the parts that need to be built to build the system