2

Software Processes

As we saw in the previous chapter, the concept of process is at the heart of the software engineering approach. According to Webster, the term *process* means "a particular method of doing something, generally involving a number of steps or operations." In software engineering, the phrase *software process* refers to the methods of developing software.

A software process is a set of activities, together with ordering constraints among them, such that if the activities are performed properly and in accordance with the ordering constraints, the desired result is produced. The basic desired result is, as stated earlier, high quality and productivity. In this chapter, we will discuss the concept of software processes further, the component processes of a software process, and some models that have been proposed.

2.1 Software Process

In an organization whose major business is software development, there are typically many processes executing simultaneously. Many of these do not concern software engineering, though they do impact software development. These could be considered nonsoftware engineering process. Business processes, social processes, and training processes, are all examples of processes that come under this. These processes also affect the software development activity but are beyond the purview of software engineering.

The process that deals with the technical and management issues of software development is called a *software process*. Clearly, many different types of activities need to be performed to develop software. All these activities

together comprise the software process. As different type of activities are being performed, which are frequently done by different people, it is better to view the software process as consisting of many component processes, each consisting of a certain type of activity. Each of these component processes typically has a different objective, though they obviously cooperate with each other to satisfy the overall software engineering objective. In this section, we will define the major component processes of a software process and what their objectives are.

2.1.1 Processes and Process Models

A successful project is the one that satisfies the expectations on all the three goals of cost, schedule, and quality (we are including functionality or features as part of quality.) Consequently, when planning and executing a software project, the decisions are mostly taken with a view to ultimately reduce the cost or the cycle time, or for improving the quality. Software projects utilize a process to organize the execution of tasks to achieve the goals on the cost, schedule, and quality fronts.

A project's process specification defines the tasks the project should perform, and the order in which they should be done. The actual process exists when the project is actually executed. Although process specification is distinct from the actual process, we will consider the process specification for a project and the actual process of the project as one and the same, and will use the term process to refer to both of them. It should, however, be mentioned that although we are assuming that there is no difficulty in a project following a specified process, in reality it is not as simple. Often the actual process being followed in the project may be very different from the project's process specification. Reasons for this divergence vary from laziness to lack of appreciation of importance of process to "old habits die hard." Ensuring that the project is following the process it planned for itself is an important issue for organizations in the business of executing projects, and there are different ways to deal with it—we will not discuss this issue in this book.

A process model specifies a general process, usually as a set of stages in which a project should be divided, the order in which the stages should be executed, and any other constraints and conditions on the execution of stages. The basic premise behind a process model is that, in the situations for which the model is applicable, using the process model as the projects process will lead to low cost, high quality, or reduced cycle time. In other words, a process is a means to reach the goals of high quality, low cost, and

low cycle time, and a process model provides generic guidelines for developing a suitable process for a project.

A project's process may utilize some process model. That is, the project's process has a general resemblance to the process model with the actual tasks being specific to the project. However, using a process model is not simply translating the tasks in the process model to tasks in the project. Typically, to achieve the project's objectives, a project will require a process that is somewhat different from the process model. That is, the project's process is generally a tailored version of a general process model. How the process model has to be tailored for a particular project, of course, depends on the project characteristics. What we need to understand is that a project's process may be obtained from a process model, by tailoring the process model to suit the project needs. For organizations that use standard processes, tailoring is an important issue. We will not discuss it further—the reader can find more about tailoring in [96].

When a process is executed on a project, software products are produced, one of them being the final software. That is, a process specifies the steps, the project executes these steps, and during the course of execution products are produced. A process limits the degrees of freedom for a project by specifying what types of activities must be undertaken and in what order, such that the "shortest" (or the most efficient) path is obtained from the user needs to the software satisfying these needs. It should be clear that it is the process that drives a project and heavily influences the expected outcomes of a project. Due to this, the focus of software engineering lies heavily on the process.

2.1.2 Component Software Processes

We have mentioned that the development process is the central process which specifies the tasks to be done in a project. Planning and scheduling the tasks and monitoring their execution fall in the domain of project management process. Hence, there are clearly two major components in a software process—a development process and a project management process—corresponding to the two axes in Figure 1.3. The development process specifies the development and quality assurance activities that need to be performed, whereas the management process specifies how to plan and control these activities so that cost, schedule, quality, and other objectives are met.

During the project many products are produced which are typically composed of many items (for example, the final source code may be composed

of many source files). These items keep evolving as the project proceeds, creating many versions on the way. To ensure that the software being produced uses the correct versions of these items requires suitable processes to control the evolution of these items. As development processes generally do not focus on evolution and changes, to handle them another process called *software configuration control process*, is often used. The objective of this component process is to primarily deal with managing change, so that the integrity of the products is not violated despite changes. Sometimes, changes in requirements may be handled separately by a *requirements change management process.*

These three constituent processes focus on the projects and the products and can be considered as comprising the *product engineering processes*, as their main objective is to produce the desired product. If the software process can be viewed as a static entity, then these three component processes will suffice. However, a software process itself is a dynamic entity, as it must change to adapt to our increased understanding about software development and availability of newer technologies and tools. Due to this, a process to manage the software process is needed.

The basic objective of the process management process is to improve the software process. By *improvement*, we mean that the capability of the process to produce quality goods at low cost is improved. For this, the current software process is studied, frequently by studying the projects that have been done using the process. The whole process of understanding the current process, analyzing its properties, determining how to improve, and then affecting the improvement is dealt with by the *process management process.*

The relationship between these major component processes is shown in Figure 2.1. These component processes are distinct not only in the type of activities performed in them, but typically also in the people who perform the activities specified by the process. In a typical project, development activities are performed by programmers, designers, testers, etc.; the project management process activities are performed by the project management; configuration control process activities are performed by a group generally called the *configuration controller*; and the process management process activities are performed by the *software engineering process group (SEPG).*

Later in the chapter we will briefly discuss each of these processes, as well as the inspection process which is used for quality control of various work products. In the rest of the book, however, we will focus primarily

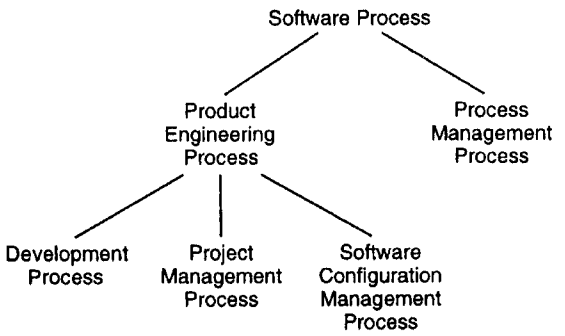

Figure 2.1: Software processes.

on processes relating to product engineering, as process management is an advanced topic beyond the scope of this book. Much of the book discusses the different phases of a development process and the processes or *methodologies* used for executing these phases. For the rest of the book, we will use the term *software process* to mean product engineering processes, unless specified otherwise.

2.1.3 ETVX Approach for Process Specification

A process has a set of phases (or steps), each phase performing a well-defined task which leads a project towards satisfaction of its goals. To reduce the cost, a process should aim to detect defects in the phase in which they are introduced. This requires that there be some verification at the end of each step, which in turn requires that there is a clearly defined output of a phase, which can be verified by some means. In other words, it is not acceptable to say that the output of a phase is an idea or a thought in the mind of someone; the output must be a formal and tangible entity. Such outputs of a development process, which are not the final output, are frequently called the *work products*. In software, a work product can be the requirements document, design document, code, prototype, and the like.

This restriction that the output of each step be some work product that can be verified suggests that the process should have a small number of steps. Having too many steps results in too many work products or documents. Due to this, at the top level, a process typically consists of a few steps, each satisfying a clear objective and producing a document which can be verified. How to perform the activity of the particular step or phase

discipline. In the context of software, as the productivity (and hence the cost of a project) and quality are determined largely by the process, to satisfy the objectives of quality improvement and cost reduction, the software process must be improved.

Having process improvement as a fundamental objective requires that the software process be a closed-loop process. That is, the process must be improved based on previous experiences, and each project done using the existing process must feed information back to facilitate this improvement. As stated earlier, this activity of analyzing and improving the process is largely done in the process management component of the software process. However, to support this activity, information from various other processes will have to flow to the process management process. In other words, to support this activity, other processes will also have to take an active part.

Process improvement is also an objective in a large project where feedback from the early parts of the project can be used to improve the execution of the rest of the project. This type of feedback is eminently suited when the iterative development process model is used—feedback from one iteration is used to improve the execution of later iterations.

2.3 Software Development Process Models

In the software development process we focus on the activities directly related to production of the software, for example, design, coding, and testing. As the development process specifies the major development and quality control activities that need to be performed in the project, the development process really forms the core of the software process. The management process is decided based on the development process. Due to the importance of the development process, various models have been proposed. In this section we will discuss some of the major models.

2.3.1 Waterfall Model

The simplest process model is the *waterfall model*, which states that the phases are organized in a linear order. The model was originally proposed by Royce [132], though variations of the model have evolved depending on the nature of activities and the flow of control between them. In this model, a project begins with feasibility analysis. Upon successfully demonstrating the feasibility of a project, the requirements analysis and project planning begins. The design starts after the requirements analysis is complete, and

coding begins after the design is complete. Once the programming is completed, the code is integrated and testing is done. Upon successful completion of testing, the system is installed. After this, the regular operation and maintenance of the system takes place. The model is shown in Figure 2.5.

The requirements analysis phase is mentioned as "analysis and planning." *Planning* is a critical activity in software development. A good plan is based on the requirements of the system and should be done before later phases begin. However, in practice, detailed requirements are not necessary for planning. Consequently, planning usually overlaps with the requirements analysis, and a plan is ready before the later phases begin. This plan is an additional input to all the later phases.

Linear ordering of activities has some important consequences. First, to clearly identify the end of a phase and the beginning of the next, some certification mechanism has to be employed at the end of each phase. This is usually done by some verification and validation means that will ensure that the output of a phase is consistent with its input (which is the output of the previous phase), and that the output of the phase is consistent with the overall requirements of the system.

The consequence of the need for certification is that each phase must have some defined output that can be evaluated and certified. That is, when the activities of a phase are completed, there should be some product that is produced by that phase. The outputs of the earlier phases are often called *work products* and are usually in the form of documents like the requirements document or design document. For the coding phase, the output is the code. Though the set of documents that should be produced in a project is dependent on how the process is implemented, the following documents generally form a reasonable set that should be produced in each project:

- Requirements document
- Project plan
- Design documents (architecture, system, detailed)
- Test plan and test reports
- Final code
- Software manuals (e.g., user, installation, etc.)

In addition to these work products, there are various other documents that are produced in a typical project. These include review reports, which are

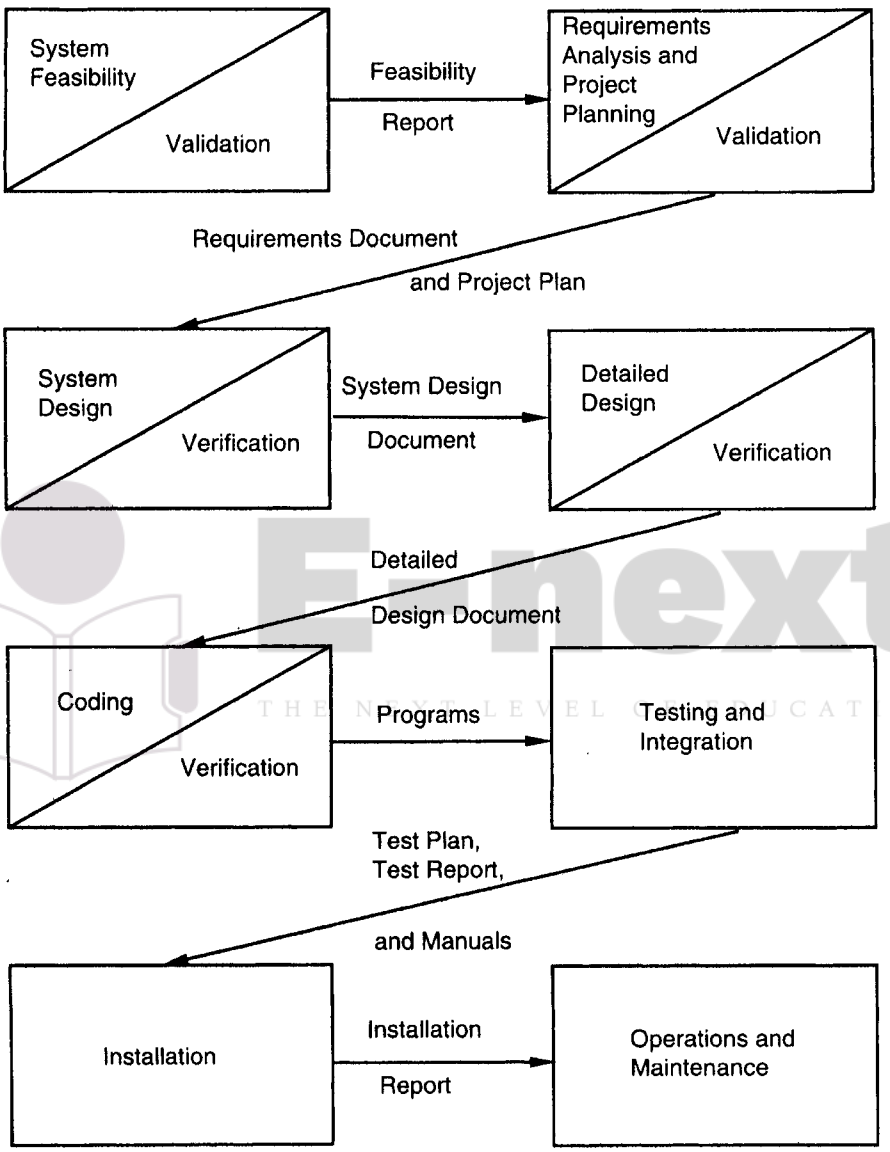

Figure 2.5: The waterfall model.

the outcome of reviews conducted for work products, as well as status reports that summarize the status of the project on a regular basis. Many other reports may be produced for improving the execution of the project or project reporting.

One of the main advantages of this model is its simplicity. It is conceptually straightforward and divides the large task of building a software system into a series of cleanly divided phases, each phase dealing with a separate logical concern. It is also easy to administer in a contractual setup—as each phase is completed and its work product produced, some amount of money is given by the customer to the developing organization.

The waterfall model, although widely used, has some strong limitations. Some of the key limitations are:

1. It assumes that the requirements of a system can be frozen (i.e., baselined) before the design begins. This is possible for systems designed to automate an existing manual system. But for new systems, determining the requirements is difficult as the user does not even know the requirements. Hence, having unchanging requirements is unrealistic for such projects.

2. Freezing the requirements usually requires choosing the hardware (because it forms a part of the requirements specification). A large project might take a few years to complete. If the hardware is selected early, then due to the speed at which hardware technology is changing, it is likely that the final software will use a hardware technology on the verge of becoming obsolete. This is clearly not desirable for such expensive software systems.

3. It follows the "big bang" approach—the entire software is delivered in one shot at the end. This entails heavy risks, as the user does not know until the very end what they are getting. Furthermore, if the project runs out of money in the middle, then there will be no software. That is, it has the "all or nothing" value proposition.

4. It is a document-driven process that requires formal documents at the end of each phase.

Despite these limitations, the waterfall model has been the most widely used process model. It is well suited for routine types of projects where the requirements are well understood. That is, if the developing organization is

quite familiar with the problem domain and the requirements for the software are quite clear, the waterfall model works well.

2.3.2 Prototyping

The goal of a prototyping-based development process is to counter the first two limitations of the waterfall model. The basic idea here is that instead of freezing the requirements before any design or coding can proceed, a throwaway prototype is built to help understand the requirements. This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding, and testing, but each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an actual feel of the system, because the interactions with the prototype can enable the client to better understand the requirements of the desired system. This results in more stable requirements that change less frequently.

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements. In such situations, letting the client "play" with the prototype provides invaluable and intangible inputs that help determine the requirements for the system. It is also an effective method of demonstrating the feasibility of a certain approach. This might be needed for novel systems, where it is not clear that constraints can be met or that algorithms can be developed to implement the requirements. In both situations, the risks associated with the projects are being reduced through the use of prototyping. The process model of the prototyping approach is shown in Figure 2.6.

A development process using throwaway prototyping typically proceeds as follows [72]. The development of the prototype typically starts when the preliminary version of the requirements specification document has been developed. At this stage, there is a reasonable understanding of the system and its needs and which needs are unclear or likely to change. After the prototype has been developed, the end users and clients are given an opportunity to use the prototype and play with it. Based on their experience, they provide feedback to the developers regarding the prototype: what is correct, what needs to be modified, what is missing, what is not needed, etc. Based on the feedback, the prototype is modified to incorporate some of the suggested changes that can be done easily, and then the users and the clients are again allowed to use the system. This cycle repeats until, in the judgment of the prototypers and analysts, the benefit from further changing the system and

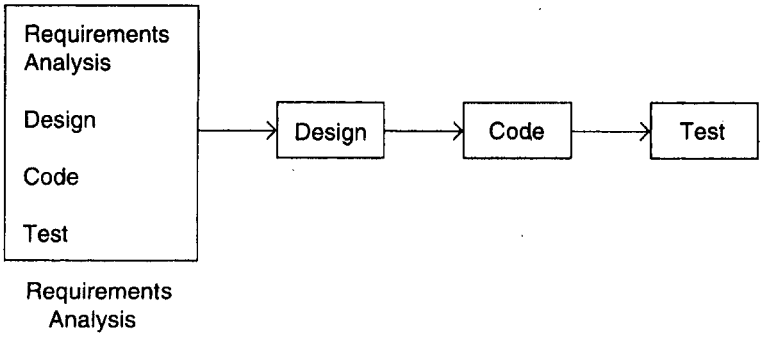

Figure 2.6: The prototyping model.

obtaining feedback is outweighed by the cost and time involved in making the changes and obtaining the feedback. Based on the feedback, the initial requirements are modified to produce the final requirements specification, which is then used to develop the production quality system.

For prototyping for the purposes of requirement analysis to be feasible, its cost must be kept low. Consequently, only those features are included in the prototype that will have a valuable return from the user experience. Exception handling, recovery, and conformance to some standards and formats are typically not included in prototypes. In prototyping, as the prototype is to be discarded, there is no point in implementing those parts of the requirements that are already well understood. Hence, the focus of the development is to include those features that are not properly understood. And the development approach is "quick and dirty" with the focus on quick development rather than quality. Because the prototype is to be thrown away, only minimal documentation needs to be produced during prototyping. For example, design documents, a test plan, and a test case specification are not needed during the development of the prototype. Another important cost-cutting measure is to reduce testing. Because testing consumes a major part of development expenditure during regular software development, this has a considerable impact in reducing costs. By using these type of cost-cutting methods, it is possible to keep the cost of the prototype less than a few percent of the total development cost.

Prototyping is often not used, as it is feared that development costs may become large. However, in some situations, the cost of software development without prototyping may be more than with prototyping. There are two ma-

jor reasons for this. First, the experience of developing the prototype might reduce the cost of the later phases when the actual software development is done. Secondly, in many projects the requirements are constantly changing, particularly when development takes a long time. We saw earlier that changes in requirements at a late stage of development substantially increase the cost of the project. By elongating the requirements analysis phase (prototype development does take time), the requirements are "frozen" at a later time, by which time they are likely to be more developed and, consequently, more stable. In addition, because the client and users get experience with the system, it is more likely that the requirements specified after the prototype will be closer to the actual requirements. This again will lead to fewer changes in the requirements at a later time. Hence, the costs incurred due to changes in the requirements may be substantially reduced by prototyping. Hence, the cost of the development after the prototype can be substantially less than the cost without prototyping; we have already seen how the cost of developing the prototype itself can be reduced.

Prototyping is well suited for projects where requirements are hard to determine and the confidence in the stated requirements is low. In such projects, a waterfall model will have to freeze the requirements in order for the development to continue, even when the requirements are not stable. This leads to requirement changes and associated rework while the development is going on. Requirements frozen after experience with the prototype are likely to be more stable. Overall, in projects where requirements are not properly understood in the beginning, using the prototyping process model can be the most effective method for developing the software. It is an excellent technique for reducing some types of risks associated with a project. We will further discuss prototyping when we discuss requirements specification and risk management.

2.3.3 Iterative Development

The iterative development process model counters the third limitation of the waterfall model and tries to combine the benefits of both prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system until the full system is implemented. At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier than testing the entire system as in the waterfall model. Furthermore, as in

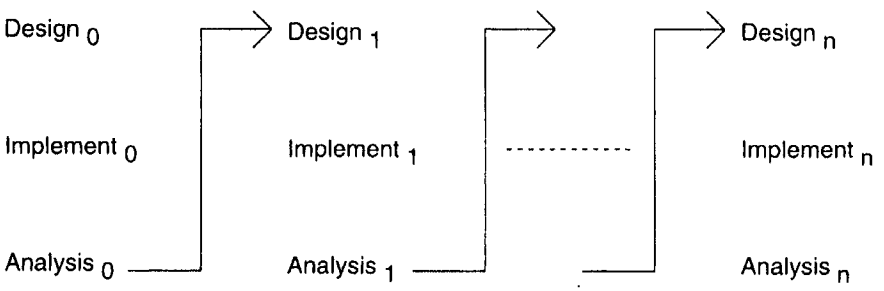

Figure 2.7: The iterative enhancement model.

prototyping, the increments provide feedback to the client that is useful for determining the final requirements of the system.

The iterative enhancement model [7] is an example of this approach. In the first step of this model, a simple initial implementation is done for a subset of the overall problem. This subset is one that contains some of the key aspects of the problem that are easy to understand and implement and which form a useful and usable system. A *project control list* is created that contains, in order, all the tasks that must be performed to obtain the final implementation. This project control list gives an idea of how far along the project is at any given step from the final system.

Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. These three phases are called *the design phase*, *implementation phase*, and *analysis phase*. The process is iterated until the project control list is empty, at which time the final implementation of the system will be available. The iterative enhancement model is shown in Figure 2.7.

The project control list guides the iteration steps and keeps track of all tasks that must be done. Based on the analysis, one of the tasks in the list can include redesign of defective components or redesign of the entire system. However, redesign of the system will generally occur only in the initial steps. In the later steps, the design would have stabilized and there is less chance of redesign. Each entry in the list is a task that should be performed in one step of the iterative enhancement process and should be simple enough to be completely understood. Selecting tasks in this manner will minimize the chances of error and reduce the redesign work. The design

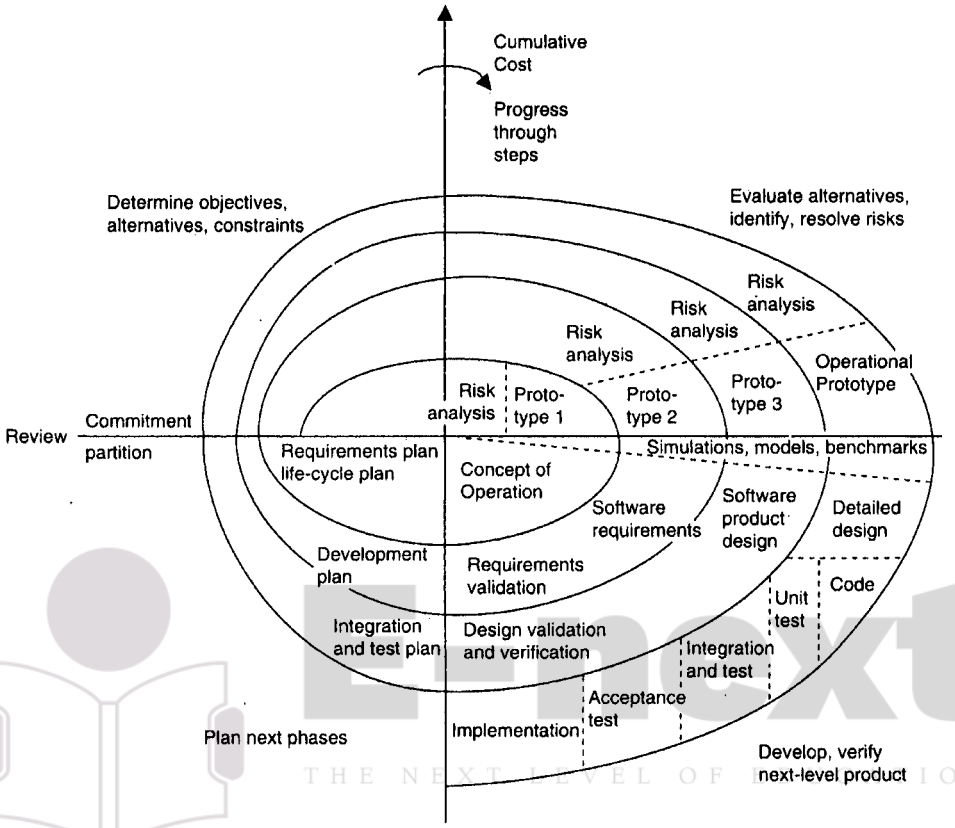

Figure 2.8: The spiral model.

and implementation phases of each step can be performed in a top-down manner or by using some other technique.

The spiral model is another iterative model that has been proposed [18]. As the name suggests, the activities in this model can be organized like a spiral that has many cycles as shown in Figure 2.8 [18].

Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the objectives, and the constraints that exist. The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints. The focus of evaluation in this step is based on the risk perception for the project. The next step is to develop strategies that resolve the uncertainties

and risks. This step may involve activities such as benchmarking, simulation, and prototyping. Next, the software is developed, keeping in mind the risks. Finally the next stage is planned.

One effective use of the iterative model is often seen in product development, in which the developers themselves provide the specifications and therefore have a lot of control on which specifications go in the system and which stay out. Generally, a version of the product is released that contains some capability. Based on the feedback from users and experience with this version, technology changes, business changes, etc., a list of additional desirable features and capabilities is generated. These features form the basis of enhancement of the software, and are included in the next version. In other words, the first version contains some core capability. And then more features are added to later versions.

In a customized software development, where the client has to provide and approve the specifications, this process model is becoming extremely popular, despite some difficulties in using it in this context. The main reason is the same—as businesses are changing very rapidly today, they never really know the "complete" requirements for the software, and there is a need to constantly add new capabilities to the software to adapt the business to changing situations. Furthermore, customers do not want to invest too much for a long time without seeing returns. In the current business scenario, it is preferable to see returns continuously of the investment made. The iterative model permits this—after each iteration some working software is delivered.

The iterative approach to software development is now widely used. Many contemporary development approaches like extreme programming [10] and Agile approaches [38] consider iterative development as a basic strategy for developing software for current times. Rational Unified Process (RUP) [108] also employs an iterative process.

2.3.4 Timeboxing Model

To speed up development, parallelism between the different iterations can be employed. That is, a new iteration commences before the system produced by the current iteration is released, and hence development of a new release happens in parallel with the development of the current release. By starting an iteration before the previous iteration has completed, it is possible to reduce the average delivery time for iterations. However, to support parallel execution, each iteration has to be structured properly and teams have to be organized suitably. The timeboxing model proposes an approach for these

[100, 99].

In the timeboxing model, the basic unit of development is a time box, which is of fixed duration. Since the duration is fixed, a key factor in selecting the requirements or features to be built in a time box is what can be fit into the time box. This is in contrast to regular iterative approaches where the functionality is selected and then the time to deliver is determined. Timeboxing changes the perspective of development and makes the schedule a non-negotiable and a high priority commitment.

Each time box is divided into a sequence of stages, like in the waterfall model. Each stage performs some clearly defined task for the iteration and produces a clearly defined output. The model also requires that the duration of each stage, that is, the time it takes to complete the task of that stage, is approximately the same. Furthermore, the model requires that there be a dedicated team for each stage. That is, the team for a stage performs only tasks of that stage—tasks for other stages are performed by their respective teams. This is quite different from other iterative models where the implicit assumption is that the same team performs all the different tasks of the project or the iteration.

Having time boxed iterations with stages of equal duration and having dedicated teams renders itself to pipelining of different iterations. (Pipelining is a concept from hardware in which different instructions are executed in parallel, with the execution of a new instruction starting once the first stage of the previous instruction is finished.) Let us consider a time box with duration T and consisting of n stages—S_1, S_2, ..., S_n, each stage S_i being executed by a dedicated team. The team of each stage has T/n time available to finish their task for a time box, that is, the duration of each stage is T/n. When the team of a stage i completes the tasks for that stage for a time box k, it then passes the output of the time box to the team executing the stage $i + 1$, and then starts executing its stage for the next time box $k + 1$. Using the output given by the team for S_i, the team for S_{i+1} starts its activity for this time box. By the time the first time box is nearing completion, there are $n - 1$ different time boxes in different stages of execution. And though the first output comes after time T, each subsequent delivery happens after T/n time interval, delivering software that has been developed in time T.

As an example, consider a time box consisting of three stages: requirement specification, build, and deployment. The requirement stage is executed by its team of analysts and ends with a prioritized list of requirements to be built in in this iteration along with a high level design. The build team

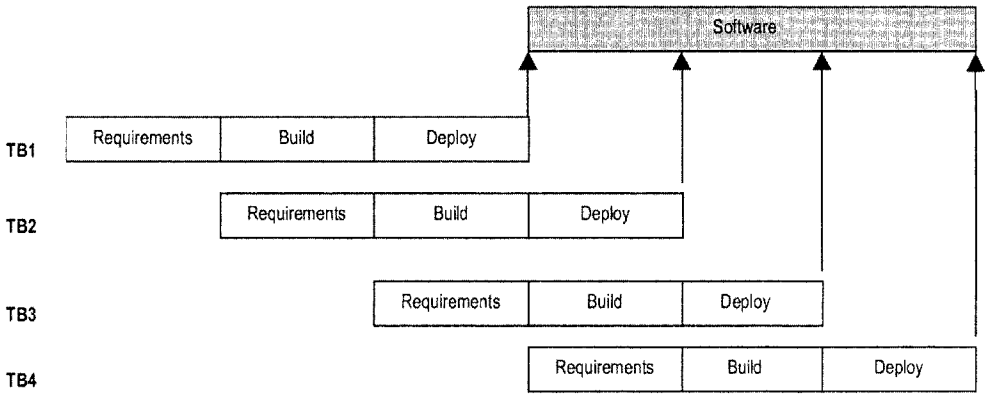

Figure 2.9: Executing the timeboxing process model.

develops the code for implementing the requirements, and performs the testing. The tested code is then handed over to the deployment team, which performs predeployment tests, and then installs the system for production use. These three stages are such that they can be done in approximately equal time in an iteration.

With a time box of three stages, the project proceeds as follows. When the requirement team has finished requirements for timebox-1, the requirements are given to the build team for building the software. The requirement team then goes on and starts preparing the requirements for timebox-2. When the build for the timebox-1 is completed, the code is handed over to the deployment team, and the build team moves on to build code for requirements for timebox-2, and the requirements team moves on to doing requirements for timebox-3. This pipelined execution of the timeboxing process is shown in Figure 2.9 [99].

With a three-stage time box, at most three iterations can be concurrently in progress. If the time box is of size T days, then the first software delivery will occur after T days. The subsequent deliveries, however, will take place after every T/3 days. For example, if the time box duration T is 9 weeks (and each stage duration is 3 weeks), the first delivery is made 9 weeks after the start of the project. The second delivery is made after 12 weeks, the third after 15 weeks, and so on. Contrast this with a linear execution of iterations, in which the first delivery will be made after 9 weeks, the second will be made after 18 weeks, the third after 27 weeks, and so on.

There are three teams working on the project—the requirements team,

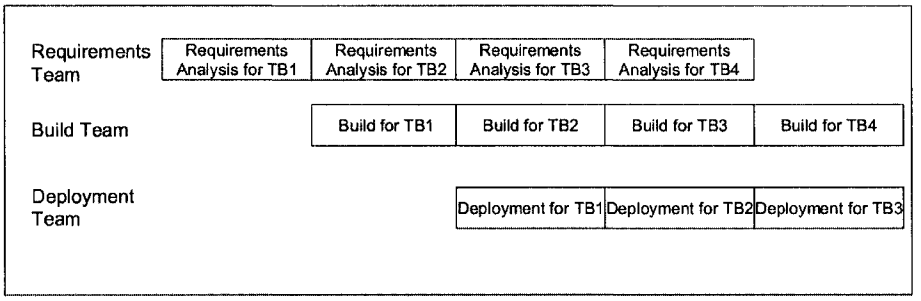

Figure 2.10: Tasks of different teams.

the build team, and the deployment team. The team-wise activity for the 3-stage pipeline discussed above is shown in Figure 2.10 [99].

It should be clear that the duration of each iteration has not been reduced. The total work done in a time box and the effort spent in it also remains the same—the same amount of software is delivered at the end of each iteration as the time box undergoes the same stages. If the same effort and time is spent in each iteration also remains the same, then what is the cost of reducing the delivery time? The real cost of this reduced time is in the resources used in this model. With timeboxing, there are dedicated teams for different stages and the total team size for the project is sum of teams of different stages. This is the main difference from the situation where there is a single team which performs all the stages and the entire team works on the same iteration.

For example, consider an iterative development with three stages, as discussed above. Suppose that it takes 2 people 2 weeks to do the requirements for an iteration, it takes 4 people 2 weeks to do the build for the iteration, and it takes 3 people 2 weeks to test and deploy. If the iterations are serially executed, then the team for the project will be 4 people (the maximum size needed for a stage)—in the first 2 weeks two people will primarily do the requirements, then all the 4 people will do the task of build, and then 3 people will do the deployment.

If this project is executed using the timeboxing process model, there will be 3 separate teams—the requirements team of size 2, the build team of size 4, and the deployment team of size 3. So, the total team size for the project is (2+4+3) = 9 persons. This is more than twice the peak team size if iterations are executed serially. It is due to this increase in team size that the throughput increases and the average delivery time decreases.

Hence, the timeboxing provides an approach for utilizing additional manpower to reduce the delivery time. It is well known that with standard methods of executing projects, we cannot compress the cycle time of a project substantially by adding more manpower. However, through the timeboxing model, we can use more manpower in a manner such that by parallel execution of different stages we are able to deliver software quicker. In other words, it provides a way of shortening delivery times through the use of additional manpower.

Timeboxing is well suited for projects that require a large number of features to be developed in a short time around a stable architecture using stable technologies. These features should be such that there is some flexibility in grouping them for building a meaningful system in an iteration that provides value to the users.

The model is not suitable for projects where it is difficult to partition the overall development into multiple iterations of approximately equal duration. It is also not suitable for projects where different iterations may require different stages, and for projects whose features are such that there is no flexibility to combine them into meaningful deliveries. We have only discussed the basic process model and have not discussed the impact of unequal stages, exceptions on the execution of this model, project management issues, etc. For further details about the model, as well as a detailed example of applying the model on a real commercial project, the reader is referred to [100, 99].

2.3.5 Comparision of Models

As discussed earlier, each process model is suitable for some context, and the main reason for studying different models is to develop the ability to choose the proper model for a given project. Using a model as the basis, the actual process for the project can be decided, which hopefully is the optimal process for the project. To help select a model, we summarize the strengths and weaknesses of the different models, along with the types of projects for which they are suitable, in Figure 2.11.

2.4 Other Software Processes

Though the development process is the central process in software processes, other processes are needed to properly execute the development process and