Preface

An introductory course on Software Engineering remains one of the hardest subjects to teach largely because of the wide range of topics the area encompasses. I have believed for some time that we often tend to teach too many concepts and topics in an introductory course resulting in shallow knowledge and little insight on application of these concepts. And Software Engineering is finally about application of concepts to efficiently engineer good software solutions.

Goals

I believe that an introductory course on Software Engineering should focus on imparting to students the knowledge and skills that are needed to successfully execute a commercial project of a few person-months effort while employing proper practices and techniques. It is worth pointing out that a vast majority of the projects executed in the industry today fall in this scope—executed by a small team over a few months. I also believe that by carefully selecting the concepts and topics, we can, in the course of a semester, achieve this. This is the motivation of this book.

The goal of this book is to introduce to the students a limited number of concepts and practices which will achieve the following two objectives:

- Teach the student the skills needed to execute a smallish commercial project.
- Provide the students necessary conceptual background for undertaking advanced studies in software engineering, through courses or on their own.

Organization

I have included in this book only those concepts that I believe are foundational and through which the two objectives mentioned above can be met. Advanced topics have been consciously left out. As executing a software project requires skills in two dimensions—engineering and project management—this book focuses on key tasks in these two dimensions, and discusses concepts and techniques that can be applied to effectively execute these tasks.

The book is organized in a simple manner, with one chapter for each of the key tasks in a project. For engineering, these tasks are requirements analysis and specification, architecture design, module level design, coding and unit testing, and testing. For project management, the key tasks are project planning and project monitoring and control, but both are discussed together in one chapter on project planning as even monitoring has to be planned. In addition, the book contains one chapter that clearly defines the problem domain of Software Engineering, and another chapter that discusses the central concept of software process which integrates the different tasks executed in a project.

Each chapter opens with some introduction and then clearly lists the chapter goals, or what the reader can expect to learn from the chapter. For the task covered in the chapter, the important concepts are first discussed, followed by a discussion of the output of the task, the desired quality properties of the output, and some practical methods and notations for performing the task. The explanations are supported by examples, and the key learnings are summarized in the end for the reader. The chapter ends with some self-assessment exercises.

Target Audience

The book is primarily intented for an introductory course on Software Engineering in any undergraduate or graduate program. It is targeted for students who know programming but have not had a formal exposure to software engineering.

The book can also be used by professionals who are in a similar state—know some programming but want to be introduced to the systematic approach of software engineering.

Teaching Support and Supplemental Resources

Though the book is self-contained, some teaching support and supplemental resources are available through a website. The URL is:

http://www.cse.iitd.ac.in/ConciseIntroToSE

The resources available on the site include:

- The powerpoint presentations for each chapter in ppt format so instructors can change them to suit their style.
- Various templates for different outputs in a project, that can be used for the student project in the course.
- A case study with most of the major outputs of the project.
- Some practice exercises for unit testing and inspections.

Acknowledgments

I would like to express my gratitude to my editor, Wayne Wheeler, who conceived this idea of a concise introductory book and created this opportunity.

I would also like to express my thanks to my wife, Shikha, and my daughters Sumedha and Sunanda for once again bearing with my moods and odd hours.

Pankaj Jalote
New Delhi, May 2008

2
Software Processes

Now that we have a better understanding of the problem domain that software engineering deals with, let us orient our discussion to software engineering itself. *Software engineering* is defined as the systematic approach to the development, operation, maintenance, and retirement of software [52].

We have seen that besides delivering software, high quality, low cost, and low cycle time are also goals which software engineering must achieve. In other words, the systematic approach must help achieve a high quality and productivity (Q&P). In software, the three main factors that influence Q&P are people, processes, and technology. That is, the final quality delivered and productivity achieved depends on the skills of the people involved in the software project, the processes people use to perform the different tasks in the project, and the tools they use.

As it is people who ultimately develop and deliver (and productivity is measured with respect to people's effort as the basic input), the main job of processes is to help people achieve higher Q&P by specifying what tasks to do and how to do them. Tools are aids that help people perform some of the tasks more efficiently and with fewer errors. It should therefore be clear that to satisfy the objective of delivering software with high Q&P, processes form the core. Consequently, in software engineering, the focus is primarily on processes, which are referred to as the systematic approach in the definition given above. It is this focus on process that distinguishes software engineering from most other computing disciplines. Many other computing disciplines focus on some type of product—operating systems, databases, etc.—while software engineering focuses on the process for producing the products.

P. Jalote, *A Concise Introduction to Software Engineering*,
DOI: 10.1007/978-1-84800-302-6_2,

As processes form the heart of software engineering, with tools and technology providing support to efficiently execute the processes, this book focuses primarily on processes. In this chapter we will discuss:

- Role of a process and a process model in a project.
- Various component processes in the software process and the key role of the development process and the project management process.
- Various models for the development process—waterfall, prototyping, iterative, RUP, timeboxing, and XP.
- The overall structure of the project management process and its key phases.

2.1 Process and Project

A process is a sequence of steps performed for a given purpose [52]. As mentioned earlier, while developing (industrial strength) software, the purpose is to develop software to satisfy the needs of some users or clients, as shown in Figure 2.1. A *software project* is one instance of this problem, and the development process is what is used to achieve this purpose.

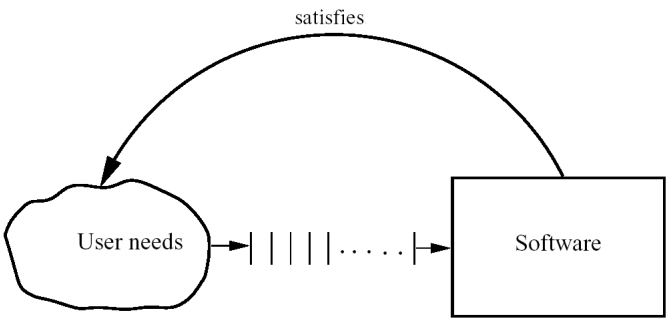

Figure 2.1: Basic problem.

So, for a project its development process plays a key role—it is by following the process the desired end goal of delivering the software is achieved. However, as discussed earlier, it is not sufficient to just reach the final goal of having the desired software, but we want that the project be done at low cost and in low cycle time, and deliver high-quality software. The role of process increases due to these additional goals, and though many processes can achieve the basic

goal of developing software in Figure 2.1, to achieve high Q&P we need some "optimum" process. It is this goal that makes designing a process a challenge.

We must distinguish process specification or description from the process itself. A process is a dynamic entity which captures the actions performed. Process specification, on the other hand, is a description of process which presumably can be followed in some project to achieve the goal for which the process is designed.

In a project, a process specification may be used as the process the project plans to follow. The actual process is what is actually done in the project. Note that the actual process can be different from the planned process, and ensuring that the specified process is being followed is a nontrivial problem. However, in this book, we will assume that the planned and actual processes are the same and will not distinguish between the two and will use the term *process* to refer to both.

A *process model* specifies a general process, which is "optimum" for a class of projects. That is, in the situations for which the model is applicable, using the process model as the project's process will lead to the goal of developing software with high Q&P. A process model is essentially a compilation of best practices into a "recipe" for success in the project. 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 a process structure that is well suited for a class of projects.

A process is often specified at a high level as a sequence of stages. The sequence of steps for a stage is the process for that stage, and is often referred to as a subprocess of the process.

2.2 Component Software Processes

As defined above, a process is the sequence of steps executed to achieve a goal. Since many different goals may have to be satisfied while developing software, multiple processes are needed. Many of these do not concern software engineering, though they do impact software development. These could be considered nonsoftware 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 processes that deal with the technical and management issues of software development are collectively called the *software process*. As a software project will have to engineer a solution and properly manage the project, there

are clearly two major components in a software process—a *development process* and a *project management process*. The development process specifies all the engineering 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. Effective development and project management processes are the key to achieving the objectives of delivering the desired software satisfying the user needs, while ensuring high productivity and quality.

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. 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.

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 2.2. 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).

In this book, we will focus primarily on processes relating to product

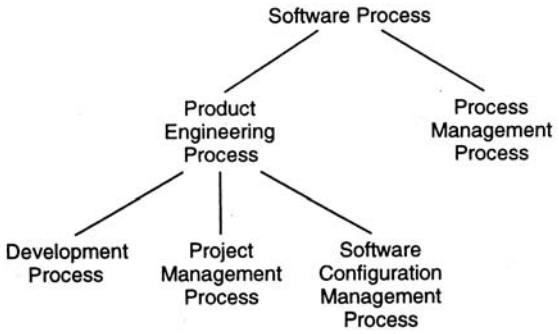

Figure 2.2: Software processes.

engineering, particularly the development and project management processes. Much of the book discusses the different phases of a development process and the subprocesses 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.

2.3 Software Development Process Models

For the software development process, the goal is to produce a high-quality software product. It therefore focuses on 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, it forms the core of the software process. The management process is often decided based on the development process.

A project's development process defines the tasks the project should perform, and the order in which they should be done. 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. The process drives a project and heavily influences the outcome.

As discussed earlier, 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 project's process

will lead to low cost, high quality, reduced cycle time, or provide other benefits. In other words, the process model provides generic guidelines for developing a suitable process for a project.

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 [74], 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 2.3.

The basic idea behind the phases is *separation of concerns*—each phase deals with a distinct and separate set of concerns. By doing this, the large and complex task of building the software is broken into smaller tasks (which, by themselves, are still quite complex) of specifying requirements, doing design, etc. Separating the concerns and focusing on a select few in a phase gives a better handle to the engineers and managers in dealing with the complexity of the problem.

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.

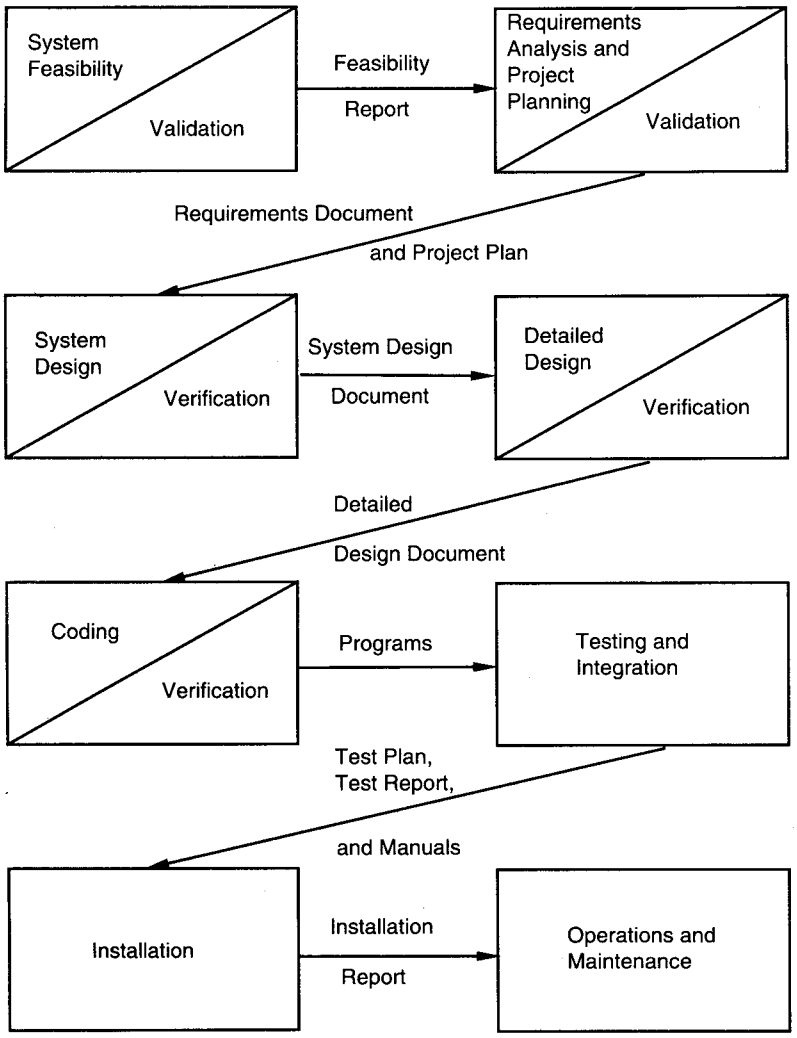

Figure 2.3: The waterfall model.

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.)

One of the main advantages of the waterfall 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 encourages "requirements bloating". Since all requirements must be specified at the start and only what is specified will be delivered, it encourages

the users and other stakeholders to add even those features which they think might be needed (which finally may not get used).

5. 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, and may be the most efficient process.

2.3.2 Prototyping

The goal of a prototyping-based development process is to counter the first limitation 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, which 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 2.4.

A development process using throwaway prototyping typically proceeds as follows [40]. 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 and explore the prototype. Based on their experience, they provide feedback to the

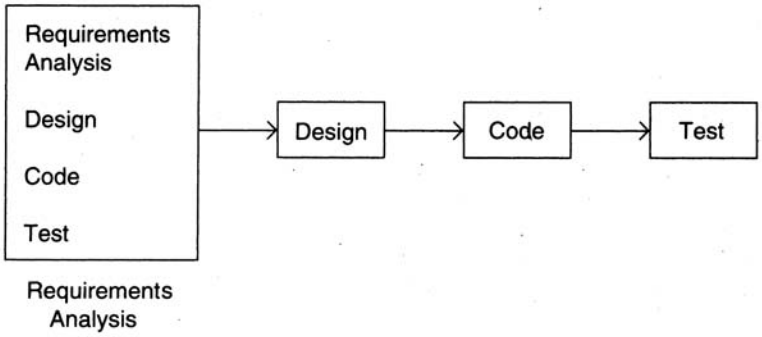

Figure 2.4: The prototyping model.

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 prototype developers and analysts, the benefit from further changing the system and 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 types of cost-cutting methods, it is possible to keep the cost of the prototype to less than a few percent of the total development cost.

And the returns from this extra cost can be substantial. First, the experience of developing the prototype will reduce the cost of the actual software development. Second, as requirements will be more stable now due to the feedback from the prototype, there will be fewer changes in the requirements. Consequently the costs incurred due to changes in the requirements will be substantially reduced. Third, the quality of final software is likely to be far superior, as the experience engineers have obtained while developing the prototype will enable them to create a better design, write better code, and do better testing. And finally, developing a prototype mitigates many risks that exist in a project where requirements are not well known.

Overall, prototyping is well suited for projects where requirements are hard to determine and the confidence in the stated requirements is low. In such 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 also an excellent technique for reducing some types of risks associated with a project.

2.3.3 Iterative Development

The iterative development process model counters the third and fourth limitations 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.

The iterative enhancement model [4] 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 2.5.

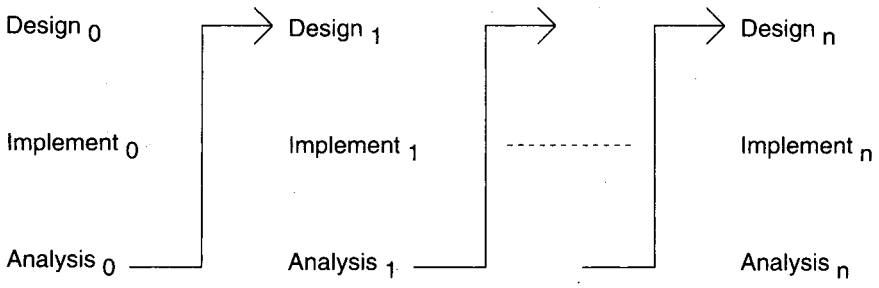

Figure 2.5: The iterative enhancement model.

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

Though there are clear benefits of iterative development, particularly in allowing changing requirements, not having the all-or-nothing risk, etc., there are some costs associated with iterative development also. For example, as the requirements for future iterations are not known, the design of a system may not be too robust. Also, changes may have to be made to the existing system to accommodate requirements of the future iterations, leading to extra rework and/or discarding of work done earlier. Overall, it may not offer the best technical solution, but the benefits may outweigh the costs in many projects.

Another common approach for iterative development is to do the requirements and the architecture design in a standard waterfall or prototyping approach, but deliver the software iteratively. That is, the building of the system, which is the most time and effort-consuming task, is done iteratively, though most of the requirements are specified upfront. We can view this approach as having one iteration delivering the requirements and the architecture plan, and then further iterations delivering the software in increments. At the start of each delivery iteration, which requirements will be implemented in this release are decided, and then the design is enhanced and code developed to implement the requirements. The iteration ends with delivery of a working software system

providing some value to the end user. Selecting of requirements for an iteration is done primarily based on the value the requirement provides to the end users and how critical they are for supporting other requirements. This approach is shown in Figure 2.6.

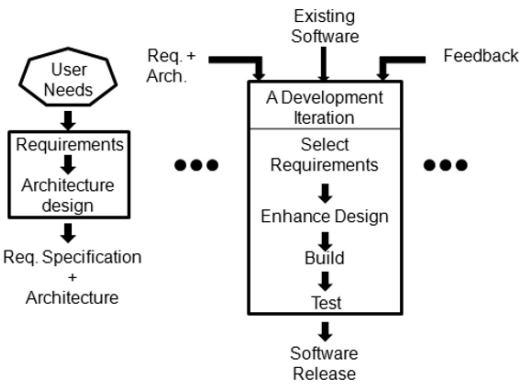

Figure 2.6: Iterative delivery approach.

The advantage of this approach is that as the requirements are mostly known upfront, an overall view of the system is available and a proper architecture can be designed which can remain relatively stable. With this, hopefully rework in development iterations will diminish. At the same time, the value to the end customer is delivered iteratively so it does not have the all-or-nothing risk. Also, since the delivery is being done incrementally, and planning and execution of each iteration is done separately, feedback from an iteration can be incorporated in the next iteration. Even new requirements that may get uncovered can also be incorporated. Hence, this model of iterative development also provides some of the benefits of the model discussed above.

The iterative approach is becoming extremely popular, despite some difficulties in using it in this context. There are a few key reasons for its increasing popularity. First and foremost, in today's world clients do not want to invest too much 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, and the risk to the client is therefore limited. Second, as businesses are changing 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. Iterative process allows this. Third, each iteration provides a working system for feedback, which helps in developing stable requirements for the next iteration. Below we will describe some other

process models, all of them using some iterative approach.

2.3.4 Rational Unified Process

Rational Unified Process (RUP) [51, 63] is another iterative process model that was designed by Rational, now part of IBM. Though it is a general process model, it was designed for object-oriented development using the Unified Modeling Language (UML). (We will discuss these topics in a later chapter).

RUP proposes that development of software be divided into *cycles*, each cycle delivering a fully working system. Generally, each cycle is executed as a separate project whose goal is to deliver some additional capability to an existing system (built by the previous cycle). Hence, for a project, the process for a cycle forms the overall process. Each cycle itself is broken into four consecutive phases:

- Inception phase
- Elaboration phase
- Construction phase
- Transition phase

Each phase has a distinct purpose, and completion of each phase is a well-defined milestone in the project with some clearly defined outputs. The purpose of the inception phase is to establish the goals and scope of the project, and completion of this phase is the *lifecycle objectives* milestone. This milestone should specify the vision and high-level capability of the eventual system, what business benefits it is expected to provide, some key illustrative use cases of the system, key risks of the project, and a basic plan of the project regarding the cost and schedule. Based on the output of this phase, a go/no-go decision may be taken. And if the project is to proceed, then this milestone represents that there is a shared vision among the stakeholders and they agree to the project, its vision, benefits, cost, usage, etc.

In the elaboration phase, the architecture of the system is designed, based on the detailed requirements analysis. The completion of this phase is the *lifecycle architecture* milestone. At the end of this phase, it is expected that most of the requirements have been identified and specified, and the architecture of the system has been designed (and specified) in a manner that it addresses the technical risks identified in the earlier phase. In addition, a high-level project plan for the project has been prepared showing the remaining phases and iterations in those, and the current perception of risks. By the end of this phase,

the critical engineering decisions regarding the choice of technologies, architecture, etc. have been taken, and a detailed understanding of the project exists. Outputs of this milestone allow technical evaluation of the proposed solution, as well as a better informed decision about cost-benefit analysis of the project.

In the construction phase, the software is built and tested. This phase results in the software product to be delivered, along with associated user and other manuals, and successfully completing this phase results in the *initial operational capability* milestone being achieved.

The purpose of the transition phase is to move the software from the development environment to the client's environment, where it is to be hosted. This is a complex task which can require additional testing, conversion of old data for this software to work, training of personnel, etc. The successful execution of this phase results in achieving the milestone *product release*. The different phases and milestones in RUP are shown in Figure 2.7.

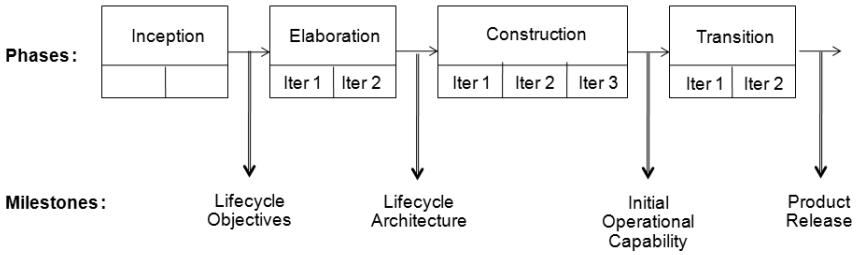

Figure 2.7: The RUP model.

Though these phases are consecutive, each phase itself may have multiple iterations, with each iteration delivering to an internal or external customer some well-defined output which is often a part of the final deliverable of that phase's milestone. Generally, it is expected that the construction phase will be broken into multiple iterations, each iteration producing a working system which can be used for feedback, evaluation, beta-testing, etc. Though iterations in construction are done often and it is clear what an iteration in this phase delivers, iterations may be done meaningfully in other phases as well. For example, in the elaboration phase, the first iteration may just specify the overall architecture and high-level requirements, while the second iteration may be done to thrash out the details. As another example, there may be multiple iterations to transition the developed software, with each iteration "making live" some part or some feature of the developed software.

RUP has carefully chosen the phase names so as not to confuse them with the engineering tasks that are to be done in the project, as in RUP the engineering tasks and phases are separate. Different engineering activities may

be performed in a phase to achieve its milestones. RUP groups the activities into different subprocesses which it calls *core process workflows*. These subprocesses correspond to the tasks of performing requirements analysis, doing design, implementing the design, testing, project management, etc. Some of the subprocesses are shown in Table 2.1.

One key difference of RUP from other models is that it has separated the phases from the tasks and allows multiple of these subprocesses to function within a phase. In waterfall (or waterfall-based iterative model), a phase within a process was linked to a particular task performed by some process like requirements, design, etc. In RUP these tasks are separated from the stages, and it allows, for example, during construction, execution of the requirements process. That is, it allows some part of the requirement activity be done even in construction, something the waterfall did not allow. So, a project, if it so wishes, may do detailed requirements only for some features during the elaboration phase, and may do detailing of other requirements while the construction is going on (maybe the first iteration of it). This not only allows a project a greater degree of flexibility in planning when the different tasks should be done, it also captures the reality of the situation—it is often not possible to specify all requirements at the start and it is best to start the project with some requirements and work out the details later.

Though a subprocess may be active in many phases, as can be expected, the volume of work or the effort being spent on the subprocess will vary with phases. For example, it is expected that a lot more effort will be spent in the requirement subprocess during elaboration, and less will be spent in construction, and still less, if any, will be spent in transition. Similarly, the model has the development process active in elaboration, which allows a project to build a prototype during the elaboration phase to help its requirements activity, if needed. However, most of the implementation does happen in the construction phase. The effort spent in a subprocess in different phases will, of course, depend on the project. However, a general pattern is indicated in Table 2.1 by specifying if the level of effort for the phase is high, medium, low, etc.

Table 2.1: Activity level of subprocesses in different phases of RUP.

	Inception	**Elaboration**	**Construction**	**Transition**
Requirements	High	High	Low	Nil
Anal. and Design	Low	High	Medium	Nil
Implementation	Nil	Low	High	Low
Test	Nil	Low	High	Medium
Deployment	Nil	Nil	Medium	High
Proj. Mgmt.	Medium	Medium	Medium	Medium
Config. Mgmt	Low	Low	High	High

Overall, RUP provides a flexible process model, which follows an iterative approach not only at a top level (through cycles), but also encourages iterative approach during each of the phases in a cycle. And in phases, it allows the different tasks to be done as per the needs of the project.

2.3.5 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 [60, 59].

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 nonnegotiable 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.)

To illustrate the use of this model, 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 this iteration along with a high-level design. The build team 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 requirements team has finished requirements for timebox-1, the requirements are given to the build team for building the software. The requirements team then goes on and starts preparing the requirements for timebox-2. When the build for 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 2.8 [59].

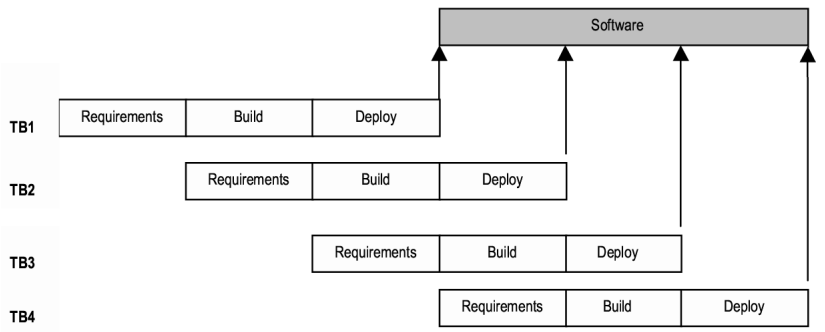

Figure 2.8: Executing the timeboxing process model.

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 after 18 weeks, the third after 27 weeks, and so on.

There are three teams working on the project—the requirements team, the build team, and the deployment team. The teamwise activity for the 3-stage pipeline discussed above is shown in Figure 2.9 [59].

It should be clear that the duration of each iteration has not been reduced.

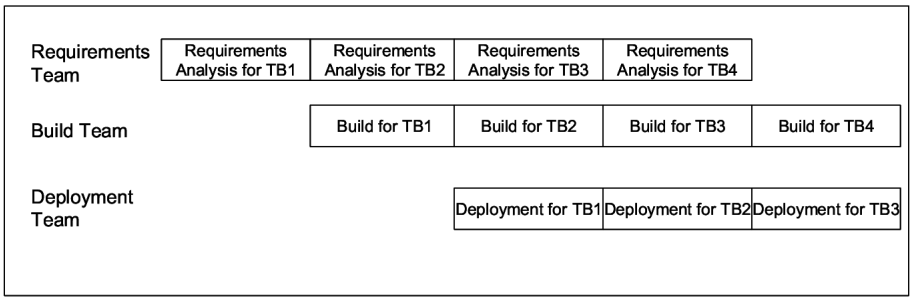

Figure 2.9: Tasks of different teams.

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 effort and time 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 the 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.

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 main cost of this model is the increased complexity of project management (and managing the products being developed) as multiple developments are concurrently active. Also, the impact of unusual situations in an iteration can be quite disruptive. Further details about the model, as well as a detailed example of applying the model on a real commercial project, are given in [60, 59].

2.3.6 Extreme Programming and Agile Processes

Agile development approaches evolved in the 1990s as a reaction to documentation and bureaucracy-based processes, particularly the waterfall approach. Agile approaches are based on some common principles, some of which are [www.extremeprogramming.org]:

- Working software is the key measure of progress in a project.
- For progress in a project, therefore, software should be developed and delivered rapidly in small increments.
- Even late changes in the requirements should be entertained (small-increment model of development helps in accommodating them).
- Face-to-face communication is preferred over documentation.
- Continuous feedback and involvement of customer is necessary for developing good-quality software.
- Simple design which evolves and improves with time is a better approach than doing an elaborate design up front for handling all possible scenarios.
- The delivery dates are decided by empowered teams of talented individuals (and are not dictated).

Many detailed agile methodologies have been proposed, some of which are widely used now. Extreme programming (XP) is one of the most popular and well-known approaches in the family of agile methods. Like all agile approaches, it believes that changes are inevitable and rather than treating changes as undesirable, development should embrace change. And to accommodate change, the development process has to be lightweight and quick to respond. For this, it develops software iteratively, and avoids reliance on detailed and multiple documents which are hard to maintain. Instead it relies on face-to-face communication, simplicity, and feedback to ensure that the desired changes are quickly and correctly reflected in the programs. Here we briefly discuss the development process of XP, as a representative of an agile process.

An extreme programming project starts with *user stories* which are short (a few sentences) descriptions of what scenarios the customers and users would like the system to support. They are different from traditional requirements specification primarily in details—user stories do not contain detailed requirements which are to be uncovered only when the story is to be implemented, therefore allowing the details to be decided as late as possible. Each story is written on a separate card, so they can be flexibly grouped.

The empowered development team estimates how long it will take to implement a user story. The estimates are rough, generally stated in weeks. Using

these estimates and the stories, *release planning* is done which defines which stories are to be built in which system release, and the dates of these releases. Frequent and small releases are encouraged, and for a release, iterations are employed. Acceptance tests are also built from the stories, which are used to test the software before the release. Bugs found during the acceptance testing for an iteration can form work items for the next iteration. This overall process is shown in Figure 2.10.

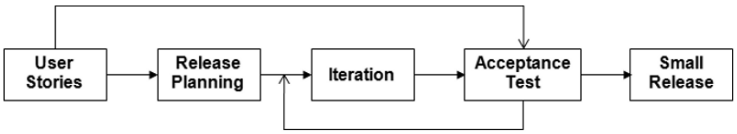

Figure 2.10: Overall process in XP.

Development is done in iterations, each iteration lasting no more than a few weeks. An iteration starts with *iteration planning* in which the stories to be implemented in this iteration are selected—high-value and high-risk stories are considered as higher priority and implemented in early iterations. Failed acceptance tests in previous iteration also have to be handled. Details of the stories are obtained in the iteration for doing the development.

The development approach used in an iteration has some unique practices. First, it envisages that development is done by pairs of programmers (called pair programming and which we will discuss further in Chapter 7), instead of individual programmers. Second, it suggests that for building a code unit, automated unit tests be written first before the actual code is written, and then the code should be written to pass the tests. This approach is referred to as test-driven development, in contrast to regular code-first development in which programmers first write code and then think of how to test it. (We will discuss test-driven development further in Chapter 7.) As functionality of the unit increases, the unit tests are enhanced first, and then the code is enhanced to pass the new set of unit tests. Third, as it encourages simple solutions as well as change, it is expected that the design of the solution devised earlier may at some point become unsuitable for further development. To handle this situation, it suggests that *refactoring* be done to improve the design, and then use the refactored code for further development. During refactoring, no new functionality is added, only the design of the existing programs is improved. (Refactoring will be discussed further in Chapter 7.) Fourth, it encourages frequent integration of different units. To avoid too many changes in the base code happening together, only one pair at a time can release their changes and integrate into the common code base. The process within an iteration is shown

in Figure 2.11.

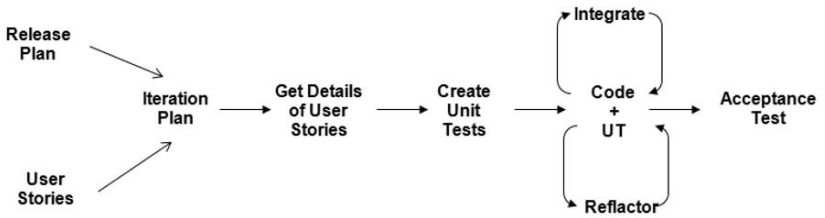

Figure 2.11: An iteration in XP.

This is a very simplified description of XP. There are many other rules in XP relating to issues like rights of programmers and customers, communication between the team members and use of metaphors, trust and visibility to all stakeholders, collective ownership of code in which any pair can change any code, team management, building quick *spike solutions* to resolve difficult technical and architectural issues or to explore some approach, how bugs are to be handled, how what can be done within an iteration is to be estimated from the progress made in the previous iteration, how meetings are to be conducted, how a day in the development should start, etc. The website www.extremeprogramming.org is a good source on these, as well as other aspects of XP.

XP, and other agile methods, are suitable for situations where the volume and pace of requirements change is high, and where requirement risks are considerable. Because of its reliance on strong communication between all the team members, it is effective when teams are collocated and of modest size, of up to about 20 members. And as it envisages strong involvement of the customer in the development, as well as in planning the delivery dates, it works well when the customer is willing to be heavily involved during the entire development, working as a team member.

2.3.7 Using Process Models in a Project

We have seen many different development process models. What is the need for the different models? As mentioned earlier, while developing (industrial strength) software, the purpose is not only to develop software to satisfy the needs of some users or clients, but we want that the project be done in low cost and cycle time, and deliver high-quality software. In addition, there could be other constraints in a project that the project may need to satisfy. Hence, given the constraints of the project, we would like to employ the process model

that is likely to maximize the chances of delivering the software, and achieve the highest Q&P. Hence, selecting a suitable development process model for a project is a key decision that a project manager has to take. Let us illustrate this by a few examples.

Suppose a small team of developers has been entrusted with the task of building a small auction site for a local university. The university administration is willing to spend some time at the start to help develop the requirements, but it is expected that their availability will be limited later. The team has been given 4 months to finish the project, and an extension of the deadline seems very improbable. It also seems that the auction site will have some features that are essential, but will also have some features that are desirable but without which the system can function reasonably well.

With these constraints, it is clear that a waterfall model is not suitable for this project, as the "all or nothing" risk that it entails is unacceptable due to the inflexible deadline. The iterative enhancement model where each iteration does a complete waterfall is also not right as it requires requirements analysis for each iteration, and the users and clients are not available later. However, the iterative delivery approach in which the complete requirements are done in the first iteration but delivery is done in iterations seems well suited, with delivery being done in two (or three) iterations (as time is short). From the requirements, the project team can decide what functionality is essential to have in a working system and include it in the first iteration. The other desirable features can be planned for the second iteration. With this approach, the chances of completing the first iteration before the final deadline increase. That is, with this model, the chances of delivering a working system increase. RUP, as it allows iterations in each phase, is also a suitable model.

Consider another example where the customers are in a highly competitive environment where requirements depend on what the competition is doing, and delivering functionality regularly is highly desirable. Furthermore, to reduce cost, the customer wants to outsource as much project work as possible to another team in another country.

For this project, clearly waterfall is not suitable as requirements are not even known at the start. Iterative enhancement also may not work as it may not be able to deliver rapidly. XP will be hard to apply as it requires that the entire team, including the customer, be collocated. For this project, the timeboxing model seems to fit the best. The whole project can employ three teams—one of analysts who will work with the customer to determine the requirements, one to do the development (which could be in some low-cost destination), and the third to do the deployment, which will be where the site is hosted. By suitably staffing the teams, the duration of each of the three phases—analysis and design, build, and deployment—can be made approximately equal. Then

the timeboxing model can be applied.

Consider another project, where a university wants to automate the registration process. It already has a database of courses and pre-requisites, and a database of student records. In this project, as the requirements are well understood (since registrations have been happening manually), the waterfall model seems to be the optimum.

2.4 Project Management Process

While the selection of the development process decides the phases and tasks to be done, it does not specify things like how long each phase should last, or how many resources should be assigned to a phase, or how a phase should be monitored. And quality and productivity in the project will also depend critically on these decisions. To meet the cost, quality, and schedule objectives, resources have to be properly allocated to each activity for the project, and progress of different activities has to be monitored and corrective actions taken when needed. All these activities are part of the project management process. Hence, a project management process is necessary to ensure that the engineering process ends up meeting the real-world objectives of cost, schedule, and quality.

The project management process specifies all activities that need to be done by the project management to ensure that cost and quality objectives are met. Its basic task is to ensure that, once a development process is chosen, it is implemented optimally. That is, the basic task is to plan the detailed implementation of the process for the particular project and then ensure that the plan is properly executed. For a large project, a proper management process is essential for success.

The activities in the management process for a project can be grouped broadly into three phases: planning, monitoring and control, and termination analysis. Project management begins with planning, which is perhaps the most critical project management activity. The goal of this phase is to develop a *plan* for software development following which the objectives of the project can be met successfully and efficiently. A software plan is usually produced before the development activity begins and is updated as development proceeds and data about progress of the project becomes available. During planning, the major activities are cost estimation, schedule and milestone determination, project staffing, quality control plans, and controlling and monitoring plans. Project planning is undoubtedly the single most important management activity, and it forms the basis for monitoring and control. We will devote one full chapter

later in the book to project planning.

Project monitoring and control phase of the management process is the longest in terms of duration; it encompasses most of the development process. It includes all activities the project management has to perform while the development is going on to ensure that project objectives are met and the development proceeds according to the developed plan (and update the plan, if needed). As cost, schedule, and quality are the major driving forces, most of the activity of this phase revolves around monitoring factors that affect these. Monitoring potential risks for the project, which might prevent the project from meeting its objectives, is another important activity during this phase. And if the information obtained by monitoring suggests that objectives may not be met, necessary actions are taken in this phase by exerting suitable control on the development activities.

Monitoring a development process requires proper information about the project. Such information is typically obtained by the management process from the development process. Consequently, the implementation of a development process model should ensure that each step in the development process produces information that the management process needs for that step. That is, the development process provides the information the management process needs. However, interpretation of the information is part of monitoring and control.

Whereas monitoring and control last the entire duration of the project, the last phase of the management process—termination analysis—is performed when the development process is over. The basic reason for performing termination analysis is to provide information about the development process and learn from the project in order to improve the process. This phase is also often called *postmortem analysis*. In iterative development, this analysis can be done after each iteration to provide feedback to improve the execution of further iterations. We will not discuss it further in the book; for an example of a postmortem report the reader is referred to [57].

The temporal relationship between the management process and the development process is shown in Figure 2.12. This is an idealized relationship showing that planning is done before development begins, and termination analysis is done after development is over. As the figure shows, during the development, from the various phases of the development process, quantitative information flows to the monitoring and control phase of the management process, which uses the information to exert control on the development process.

We will in a later chapter discuss in detail the project planning phase. As a plan also includes planning for monitoring, we will not discuss the monitoring separately but discuss it as part of the planning activity.

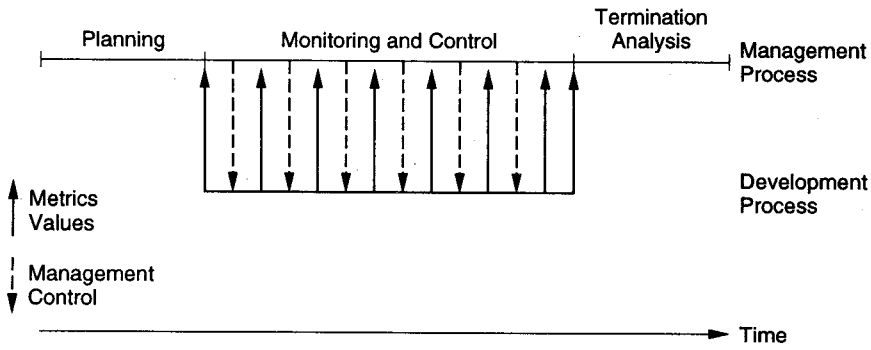

Figure 2.12: Temporal relationship between development and management process.

2.5 Summary

- The quality and productivity achieved in a software project depends on the process used for executing the project. Due to this, processes form the heart of software engineering.
- A process is the set of activities that are performed in some order so that the desired results will be achieved. A process model is a general process specification which has been found to be best suited for some situations.
- A software process consists of many different component processes, most important being the development process and the project management process.
- Development process focuses on how the software is to be engineered. There are many different process models, each being well suited for some type of problems.
  - The waterfall model is conceptually the simplest model of software development, where the requirement, design, coding, and testing phases are performed in linear progression. It has been very widely used, and is suitable for well-understood problems.
  - In the prototyping model, a prototype is built before building the final system, which is used to further develop the requirements leading to more stable requirements. This is useful for projects where requirements are not clear.
  - In the iterative development model, software is developed in iterations, each iteration resulting in a working software system. This model does not

require all requirements to be known at the start, allows feedback from earlier iterations for next ones, and reduces risk as it delivers value as the project proceeds.

- In RUP, a project is executed in a sequence of four phases—inception, elaboration, construction, and transition, each ending in a defined milestone. A phase may itself be done iteratively. The subprocesses of requirements, design, coding, testing, etc. are considered as active throughout the project, though their intensity varies from phase to phase. RUP is a flexible framework which can allow a project to follow a traditional waterfall if it wants to, or allow prototyping, if it so wishes.
- In the timeboxing model, the different iterations are of equal time duration, and are divided into equal length stages. There is a committed team for each stage of an iteration. The different iterations are then executed in a pipelined manner, with each dedicated team working on its stage but for different iterations. As multiple iterations are concurrently active, this model reduces the average completion time of each iteration and hence is useful in situations where short cycle time is highly desirable.
- Agile approaches to development are based on some key principles like developing software in small iterations, working system as the measure of progress, and allowing change at any time. In extreme programming (XP) approach, a project starts with short user stories, details of which are obtained in the iteration in which they are implemented. In an iteration, development is done by programmer-pairs, following the practices of test-driven development, frequent integration, and having simple designs which are refactored when needed.

– The project management process focuses on planning and controlling the development process and consists of three major phases—planning, monitoring and control, and termination analysis. Much of project management revolves around the project plan, which is produced during the planning phase.

Self-Assessment Exercises

1. What is the relationship between a process model, process specification, and process for a project?
2. What are the key outputs during an iteration in a project following an iterative development model?
3. Which of the development process models discussed in this chapter would you employ for the following projects?

a) A simple data processing project.
b) A data entry system for office staff who have never used computers before. The user interface and user-friendliness are extremely important.
c) A spreadsheet system that has some basic features and many other desirable features that use these basic features.
d) A web-based system for a new business where requirements are changing fast and where an in-house development team is available for all aspects of the project.
e) A Web-site for an on-line store which has a long list of desired features it wants to add, and it wants a new release with new features to be done very frequently.

4. A project uses the timeboxing process model with three stages in each time box (as discussed in the chapter), but with unequal length. Suppose the requirement specification stage takes 2 weeks with a team of 2 people, the build stage takes 3 weeks with a team of 4 people, and deployment takes 1 week with a team of 2 people. Design the process for this project that maximizes resource utilization. Assume that each resource can do any task. (Hint: Exploit the fact that the sum of durations of the first and the third stage is equal to the duration of the second stage.)
5. What effect is the project monitoring activity likely to have on the development process?