The Timeboxing Process Model for Iterative Software Development

Pankaj Jalote
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur – 208016; India

Aveejeet Palit, Priya Kurien
Infosys Technologies Limited
Electronics City
Bangalore – 561 229; India

Contact: *jalote@iitk.ac.in*

ABSTRACT

In today's business where speed is of essence, an iterative development approach that allows the functionality to be delivered in parts has become a necessity and an effective way to manage risks. In an iterative process, the development of a software system is done in increments, each increment forming of an iteration and resulting in a working system. A common iterative approach is to decide what should be developed in an iteration and then plan the iteration accordingly. A somewhat different iterative is approach is to time box different iterations. In this approach, the length of an iteration is fixed and what should be developed in an iteration is adjusted to fit the time box. Generally, the time boxed iterations are executed in sequence, with some overlap where feasible. In this paper we propose the *timeboxing* process model that takes the concept of time boxed iterations further by adding pipelining concepts to it for permitting overlapped execution of different iterations. In the timeboxing process model, each time boxed iteration is divided into equal length stages, each stage having a defined function and resulting in a clear work product that is handed over to the next stage. With this division into stages, pipelining concepts are employed to have multiple time boxes executing concurrently, leading to a reduction in the delivery time for product releases. We illustrate the use of this process model through an example of a commercial project that was successfully executed using the proposed model.

Keywords: Software process, life cycle process, process models, iterative development, timeboxing, pipelining.

1 INTRODUCTION

The main objective of a software project can be stated as follows –deliver a high quality software product within schedule and within budget. A successful project is the one that satisfies the constraints on all the three fronts of cost, schedule, and quality (we are including functionality or features as part of quality, though they could be treated as another driver.) 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.

A software project has to execute a number of engineering and management tasks for delivering a software product that satisfies the user requirements. Software projects utilize a process to organize the execution of the tasks to achieve the goals on the cost, schedule, and quality fronts. A process typically specifies the tasks that should be performed and the order in which they should be performed. Processes so utilized frequently conform to a process model – a general process structure for the lifecycle of software development. A process model generally specifies the 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 any process model is that, in the situations for which the model is applicable, using the process model for a project 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.

Software development is a large and complex task. As with any complex problem, the solution approach relies on the "divide and conquer" strategy. For software it means that this complex problem of developing software should be divided into parts that can be solved separately. At the top level, this division is typically done by breaking the overall project into key phases, with each phase focusing on a separate task. In other words, phases help in "separation of concerns". This partitioning of the whole problem into a set of phases is typically what a process model does. A process model specifies the phases such that this set of phases executed in the specified order leads to a successful execution of the project.

It should be pointed out that typically within each phase also methodologies or mini-processes are used to further apply the divide-and-conquer approach. However, process models usually focus only on the top level, phase-wise organization. Frequently, the major phases are requirements analysis and specification, design, build, and test. Process models specify how these tasks are partitioned and organized, keeping in view the project constraints.

The most influential process model is the waterfall model, in which the different phases of requirements specification, design, coding, and testing are performed in sequence. In this process model, the overall task of developing software is broken into a few phases, with a phase getting initiated when the previous phase ended. The linear organization of phases in this model requires that the requirements be frozen early in the requirements phase. Due to the current scenario of changing requirements and need for shortening the cycle time, iterative process models have now become more common. In an iterative model, the software is developed in a series of iterations, with each iteration acting like a "mini-waterfall" and delivering some software. In a typical iterative development project, the first iteration builds some core system and subsequent iterations add new features and functionality on the existing software. A different shade of iterative development is to have time boxed iterations in which each iteration is fixed in time and the functionality to be developed in an iteration is adjusted to fit the time box.

In an iterative development, generally the different iterations are executed in sequence. This form of iterative development does not directly help reduce the cycle time. However, iterative development also opens the possibility of executing different iterations in parallel and thereby reducing the average cycle time of an iteration. To exploit this potential of parallel execution of iterations, suitable process models are needed to structure the execution of different tasks in different iterations. In this chapter we describe the timeboxing process model that enhances the time boxed iterations by concepts of pipelining, thereby allowing parallel execution of iterations in a structured manner, resulting in a reduction in the cycle time for deliveries.

The chapter is organized as follows. In the next section we discuss the iterative development approaches in general and see how the timeboxing model relates to them. In Section 3, we describe the timeboxing process model in more detail, execution of a project using this process model, and issues like team size and impact of unequal stages or exceptions on the execution. In section 4, we discuss some aspects of applying the process model on projects – the nature of projects for which this is suitable, how changes are handled, project management issues, etc. In section 5 we discuss a real commercial project in which we applied this model, and discuss how we dealt with some of the constraints that the project presented. The chapter ends with conclusions.

2. ITERATIVE DEVELOPMENT MODELS

One cannot discuss the iterative models without first discussing the waterfall model, as it is the shortcomings of this model that lead to the development of the iterative models. The waterfall model for software development was first proposed by Royce [Royce, 1970] to suggest that there should be many distinct stages in a project execution. Though the waterfall model suggests a linear execution of stages, Royce had in fact suggested that, in practice, there is a need for feedback from testing to design and from design to early stages of requirements. In any case, waterfall model as a linear sequence of stages became the most influential process model – it was conceptually simple and was contractually somewhat easier to administer (e.g. each stages can be defined as a milestone at which some output is obtained and some payment is made.)

Waterfall model has some well known limitations [Boehm, 1981]. The biggest drawback with the waterfall model is that it assumes that requirements are stable and known at the start of the project. Unchanging requirements, unfortunately, do not exist in reality, and requirements do change and evolve. In order to accommodate requirement changes while executing the project in the waterfall model, organizations typically define a change management process which handles the change requests. Another key limitation is that it follows the "big bang" approach – the entire software is delivered in one shot at the end. This entails heavy risks, as the users do not know till the very end what they are getting. These two key limitations can be alleviated through the use of an iterative development model.

In an iterative development, software is built and delivered (either for production use or for feedback) in iterations – each iteration delivering a working software system that is generally an increment to the previous delivery. Iterative enhancement [Basili and Turner, 1975] and spiral [Boehm, 1988] are two well-known process models that support iterative development. More recently, agile methods [Cockburn, 2001] and XP [Beck, 2000] also promote iterative development – iterative development is a part of the agile manifesto and small iterations is a key practice in the XP methodology. Iterative development is also a foundation for methodologies like RUP [Kruchten, 2000] and DSDN [Stapleton, 2003]. The concept of iteratively developing software has been around for a long time and for a history of iterative development, the reader is referred to the paper by Larman and Basili [Larman and Basili, 2003].

With iterative development, the release cycle becomes shorter, which reduces some of the risks associated with the "big bang" approach. Requirements need not be completely understood and specified at the start of the project – they can evolve over time and can be incorporated in the system in any iteration. Incorporating change requests is also easy as any new requirements or change requests can be simply passed on to a future iteration. Overall, iterative development is able to handle some of the key shortcomings of the waterfall model, and is well suited for the rapidly changing business world, despite having some of its own drawbacks. (E.g. it is hard to preserve the simplicity and integrity of the architecture and the design.)

The commonly used iterative development approach is organized as a sequence of iterations, with each of the iterations delivering parts of the functionality. Features to be built in an iteration are decided in the start and then the iteration is planned for delivering them (an approach called *feature boxing* in [Malotaux].) Though the overall delivered functionality is delivered in parts, the total development time is not reduced. In fact, it can be argued that if the requirements are known then for the same amount of functionality, iterative development might take more time than a waterfall model-based development. Furthermore, for each iteration, the risk of over-committing and not being able to deliver in time is still there, though it is reduced as the scope of each iteration is smaller.

One approach to alleviate the schedule risk is to time box the iterations. With time boxing of each iteration, the duration of each iteration, and hence the delivery time, is fixed. The functionality that can be delivered in a time box is what is negotiated for an iteration while keeping the delivery time fixed. In contrast, in feature boxing, the functionality is selected and then the time to deliver is determined. Time boxing changes the perspective of development and makes the schedule as a non-negotiable and a high priority commitment. As the delivery date is sacrosanct, this approach also helps sharpen the focus on important requirements since only limited requirements can be accommodated and there is no flexibility to increase them. Time boxed development is a natural extension of an iterative development approach and has been proposed for use in RUP [Larman, 2002], in DSDN [Stapleton, 2003], and is a key strategy for rapid application development [Kerr and Hunter 1994, Martin 1991].

Even with time boxed development, the total time of development remains the same, if different iterations are executed in sequence. Time boxing helps in reducing development time by better managing the schedule risk and the risk of “gold plating”, which is a major cause of cost and schedule overruns.

To reduce the total development time, one approach is to use components and employ reuse – a technique that is employed frequently. With components and reuse, time to build an application is reduced as less software is being developed to deliver the desired functionality by leveraging existing software. However, components and reuse can be employed to reduce the delivery time even if the underlying development process model is waterfall-like and is not iterative.

Another natural approach to speed up development that is applicable only when iterative development process is used, is to employ parallelism between the different iterations. 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 delivery time for successive iterations (after the first iteration.) The Rational Unified Process (RUP) uses this approach by suggesting that the final stages of an iteration may overlap with the initial stages of the next [Kruchten, 2000]. In practice, many products evolve this way – the development of the next version starts well before the development of the earlier version has completed. However, this type of overlapping of iterations is unstructured and is not systematic.

Figure 1: Waterfall, Iterative, Timeboxing Models

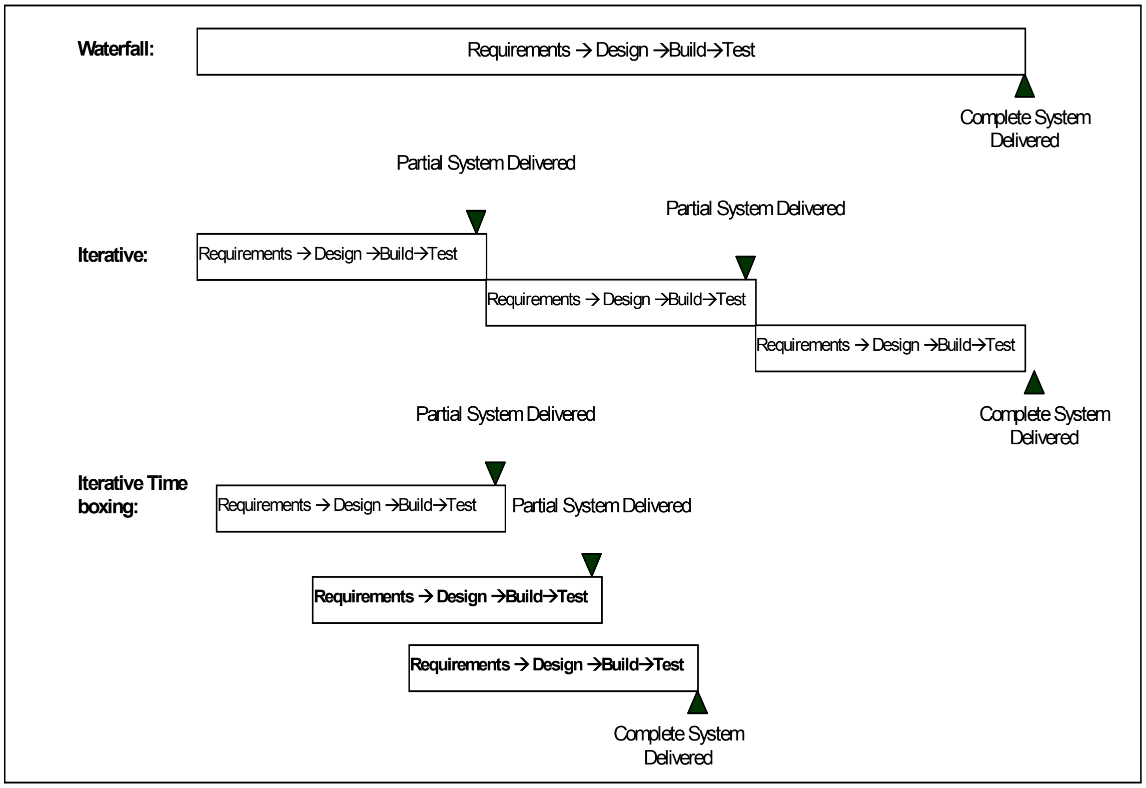
Waterfall:
Requirements → Design →Build→Test
Complete System Delivered
Partial System Delivered
Partial System Delivered
Iterative:
Requirements → Design →Build→Test
Requirements → Design →Build→Test
Requirements → Design →Build→Test
Complete System Delivered
Partial System Delivered
Iterative Time boxing:
Requirements → Design →Build→Test
Partial System Delivered
Requirements → Design →Build→Test
Requirements → Design →Build→Test
Complete System Delivered

Figure 1: Waterfall, Iterative, and timeboxing process models

In this chapter, we discus in detail the *timeboxing* process model that takes the concept of parallelism between different iterations further and structures it by using the pipelining concepts [Hennessy and Patterson 1998]. In this model, iterative development is done in a set of fixed duration time boxes. Each time box is divided into stages/phases of approximately equal duration, and the work of each stage is done by a dedicated team. Multiple iterations are executed concurrently by employing pipelining – as the first stage of the first time box completes, the team for that stage starts its activities for the next time box, while the team for the next stage carries on with the execution of the first time box. This model ensures that deliveries are made with a much greater frequency than once every time box, thereby substantially reducing the cycle time for each delivery. How execution of a project proceeds when using the waterfall, iterative, or the timeboxing process model proceed is shown in Figure 1.

As mentioned above, the concept of using time boxes for iterations has been around for quite some time, though mostly for predefining the delivery times and deriving the benefits that come from it. Overlapping of iterations also has been talked about and has been used in practice by many product vendors. The timeboxing process model formalizes this concept of overlapping iterations by structuring the iterations to facilitate the use of pipelining concepts to reduce software delivery time. It provides a conceptual framework that is grounded in the pipelining concepts developed to speed up execution of instructions in processors. The discussion of the timeboxing model is based on our earlier paper [Jalote et al 2004].

Note that this overlapping is different from the overlapping of different phases within an iteration, as is proposed in RUP[Kruchten 2000]. Overlapping the different phases means that an earlier phase in an iteration does not have to completely finish before the next phase of that iteration starts. This overlapping of phases avoids the hard "hand-over" from one phase to another and allows, for example, requirements to evolve even while design is being done. Though this approach for overlapping has clear practical benefits in handling evolving requirements it, however, does not provide any direct benefit in reducing the delivery time.

Note that the concept of pipelining the execution of different iterations in a software development is also quite different from the concept of software pipelines [Hennessy and Patterson 1998]. Software pipelines are used to apply some techniques to the source code of a program such that the transformed program is better suited for pipelined execution of the instructions in the hardware.

We believe that the timeboxing process model is a viable approach for executing projects when there is a strong business need to deliver working systems quickly. Due to the constraints the model imposes, this model is likely to work well for medium sized projects which have a stable architecture and have a lot of feature requirements that are not fully known and which evolve and change with time. Application of the model is eased if there is flexibility in grouping the requirements for the purpose of delivering meaningful systems that provide value to the users. The model is not likely to work well for projects where flexible grouping of requirements for the purpose of delivery is not possible. It is also not likely to work well where development within an iteration cannot be easily divided into clearly defined stages, each of which ending with some work product that form the main basis for the next stage.

3 THE TIMEBOXING PROCESS MODEL

In the timeboxing model, as in other iterative development approaches, some software is developed and a working system is delivered after each iteration. However, in timeboxing, each iteration is of equal duration, which is the length of the time box. In this section we discuss the various conceptual issues relating to this process model.

3.1 A Time box and Stages

In the timeboxing process model, the basic unit of development is a time box, which is of fixed duration. Within this time box all activities that need to be performed to successfully release the next version are executed. 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.

Each time box is divided into a sequence of *stages*, like in the waterfall model. Each stage performs some clearly defined task of the iteration and produces a clearly defined output. The output from one stage is the only input from this stage to the next stage, and it is assumed that this input is sufficient for performing the task of the next stage. When the output of one stage is given to the next stage, the development activity is *handed over* to the next stage, and for this time box, the activity shifts to the next stage. Note that handing over will require some artifacts to be created by a stage which provide all the information that is needed to perform the next stage.

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. (Impact of exceptions to this are discussed later.) Such a time box with equal stages in which the task is handed over from one stage to the next is shown in Figure 2. In the figure, the time box is the outer box, and stages are represented by boxes within the time box. In this figure, we have explicitly shown the handing over from one stage to another by showing an output of a stage which forms the input to the next. Later, we will dispense with this and just show a time box divided into stages.

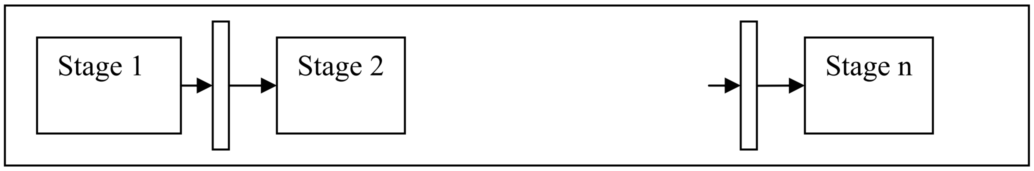

Figure 2: A time box with equal stages

There is 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. With dedicated teams, when an iteration is handed over from one stage to the next as shown in Figure 2, in effect, the responsibility for further development of that iteration is being handed over to another team. This is quite different from most other models where the implicit assumption is that the same team performs all the different tasks of the project or the iteration. In such resource models where the same team performs the entire task, the phases in the development process are logical tasks that are performed by the same team and completion of a task ending with a work product is primarily for review and control purposes. There is really no handing over to another resource group as the development moves from stage to stage.

As pipelining is to be employed, the stages must be carefully chosen. Each stage performs some logical activity which may be communication intensive – that is, the team performing the task of that stage needs to communicate and meet regularly. However, the stage should be such that its output is all that is needed from this stage by the team performing the task of the next stage. In other words, the output should be such that when the iteration is handed over from one stage to another, the team to which the iteration has been handed over needs to communicate minimally with the previous stage team for performing their task. Note that it does not mean that the team for a stage cannot seek clarifications with teams of earlier stages – all it means is that the communication needs between teams of different stages are so low that their communication has no significant effect on the work of any of the teams. However, this approach disallows parallelism between the different stages within a time box – it is assumed that when a stage finishes, its task for this iteration is completely done and only when a stage finishes its task, the activity of the next stage starts.

3.2 Pipelined Execution

Having time boxed iterations with stages of equal duration and having dedicated teams renders itself to pipelining of different iterations. Pipelining is one of the most powerful concepts for making faster CPUs and is now used in almost all processors [Hennessy and Patterson 1998]. Pipelining is like an assembly line in which different pipe-segments execute different parts of an instruction. The different segments execute in parallel, each working on a different instruction. The segments are connected to form a pipe – new instructions enter one end of the pipe and completed instructions exit from the other end. If each segment takes one clock cycle to complete its task, then in a steady state, one instruction will exit in each clock cycle from the pipeline, leading to the increased throughput and speedup of instruction execution. We refer the reader to [Hennessy and Patterson 1998] for further details on the concepts of pipelining in hardware.

In timeboxing, each iteration can be viewed like one instruction whose execution is divided into a sequence of fixed duration stages, a stage being executed after the completion of the previous stage. In general, let us consider a time box with duration T and consisting of *n* stages – S_1, S_2, …, S_n. As stated above, each stage S_i is executed by a dedicated team (similar to having dedicated segment for executing a stage in an instruction).

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.

Another way to view it is to consider the basic time unit as the duration of a stage. Suppose that a stage takes one stage-time-unit (STU) to complete. Then a n-stage time box will take n STUs to complete. However, after the completion of the first iteration, an iteration will complete after each STU. Once an STU is chosen, the time box will have to be divided into stages such that each stage takes only one STU to complete. With pipelining, if all the stages are properly balanced, in the steady state, on an average it will take one STU to complete an iteration.

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 this iteration. The requirements document is the main input for the build team, which designs and develops the code for implementing these requirements, and performs the testing. The tested code is then handed over to the deployment team, which performs pre-deployment tests, and then installs the system for production use.

These three stages are such that in a typical short-cycle development, they can be of equal duration (though the effort consumed is not the same, as the manpower deployed in the different stages is different.) Also, as the boundary between these stages is somewhat soft (e.g. high level design can be made a part of the first stage or the second), the duration of the different stages can be made approximately equal by suitably distributing the activities that lie at the boundary of two adjacent stages.

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. Meanwhile, the requirement team 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 3.

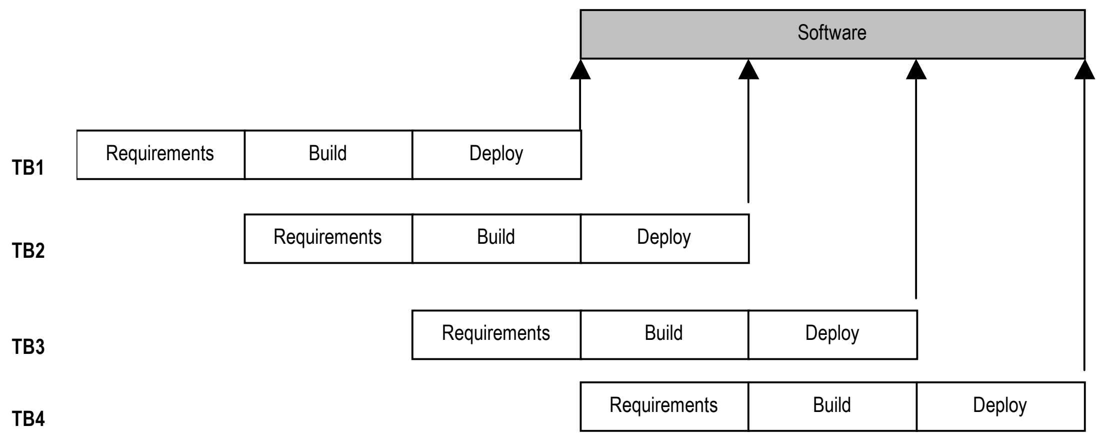

Figure 3: Executing the timeboxing process model

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 will be made after 18 weeks, the third after 27 weeks, and so on.

3.3 Time, Effort and Team Size

It should be clear that the duration of each iteration has not been reduced – in fact it may even increase slightly as the formality of handing over between stages may require extra overhead that may not be needed if the strict partitioning in stages was not there. The total work done in a time box 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. However, the delivery time to the end client (after the first iteration) reduces by a factor of n with an n-stage time box if pipelining is employed. As in hardware, let us define the *speedup* of this process model as the ratio of the number of stage-time-units it takes to deliver one time box output if development is done without overlapping with the number of stage-time-units it takes on an average to deliver one time box output when pipelining is used. It is clear that, in ideal conditions, with a n-stage time box, the speedup is n. In other words, the development is n times faster with this model as compared to the model where the iterations are executed serially.

We can also view it in terms of throughput. Let us define the throughput as the amount of software delivered per unit time. Note that throughput is different from productivity – in productivity we compute the output per unit effort while in throughput we are interested in the output per unit time. We can clearly see that in steady state, the throughput of a project using timeboxing is n times more than what is achieved if serial iterations were employed.

In other words, n times more functionality is being delivered per unit time. If in an iteration the team can develop S size units (in lines of code, function points, or some other unit) and the duration of the time box is T, then the throughput of a process in which the iterations are executed serially will be S/T size units per unit time. With the timeboxing process model, however, the throughput is n * S/T. That is, the throughput also increases by a factor of n.

If the size of the team executing the stage S_i is R_i, then the effort spent in the stage S_i is

$E(S_i) = R_i * T/n$.

Note that the model only requires that the duration of the stages be approximately the same, which is T/n in this case (or one stage-time-unit.) It does not imply that the amount of effort spent in a stage is same. The effort consumed in a stage S_i also depends on R_i, the size of the team for that stage. And there is no constraint from the model that the different R_is should be the same.

The total effort consumed in an iteration, i.e. in a time box, is

$$E(TB) = \sum_{i=1}^{n} E(S_i).$$

This effort is no different than if the iterations were executed serially – the total effort for an iteration is the sum of the effort for its stages. In other words, the total effort for an iteration remains the same in timeboxing as in serial execution of iterations. Note also that the productivity of the timeboxing process model (assuming no extra overhead) is same as without pipelining of iterations. In each of the cases, if S is the output in each iteration, the productivity is S/E(TB) size units per person-day (or whatever unit of effort we choose.)

If the same effort and time is spent in each iteration and the productivity also remains the same, then what is the cost of reducing the delivery time? The real cost of this increased throughput is in the resources used in this model. With timeboxing, there are dedicated teams for different stages. Each team is executing the task for that stage for a different iteration. 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. With timeboxing process model, different teams are working on different iterations and the total team size for the project is $\sum_{i=1}^{n}$ R_i. The team-wise activity for the 3-stage pipeline discussed above is shown in Figure 4.

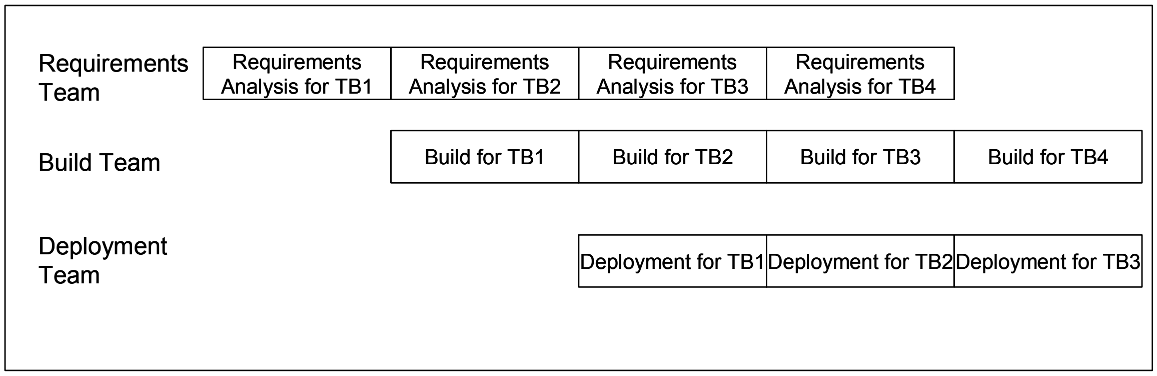

Figure 4: Tasks of different teams

Let us compare the team size of a project using timeboxing with another project that executes iterations serially. In a serial execution of iterations, it is implicitly assumed that the same team performs all the activities of the iteration, that is, they perform all the stages of the iteration. For sake of illustration, let us assume that the team size is fixed throughout the iteration, and that the team has R resources. So, the same R people perform the different stages – first they perform the tasks of stage 1, then of stage 2, and so on. Assuming that even with dedicated resources for a stage, the same number of resources are required for a stage as in the linear execution of stages, the team size for each stage will be R. Consequently, the total project team size when the time box has n stages is n*R. That is, the team size in timeboxing is n times the size of the team in serial execution of iterations.

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 four people will do the task of build, and then 3 people will do the deployment. That is, the team size is likely to be the peak team size, that is, four people. If the resources are allocated to the project as and when needed (that is, there is a resource ramp-up in the start and ramp-down in the end,) the average team size is (2+4+3)/3 = 3 persons.

If this project is executed using the timeboxing process model, there will be three 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 three times the average team size if iterations are executed serially. It is due to this increase in team size that the throughput is also 3 times the throughput in serial-iterations.

Hence, in a sense, 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 [8]. This principle holds here also within a time box – we cannot reduce the size of a time box 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.

The above example can be used to illustrate another usefulness of the timeboxing model. We know that in a project, typically the manpower requirement follows a Rayaleigh curve [ref], which means that in the start and end, the project typically requires less resources (even if there are more resources available, it can gainfully consume only the required number), and the peak is reached somewhere in the middle. This poses a challenge for staffing in a project – if the team size is fixed, then it means that in the start and the end, they may be underutilized (or in the middle, the progress is slower than it could be as not enough resources are available.) To avoid this underutilization, resources can be ramped-up on a need basis in the start and then ramped-down towards the end.

The same sort of resource requirement can be expected within an iteration, as an iteration is essentially a small project. However, though in a waterfall process model adding the resources slowly and then slowly removing them from the project can be done relatively easily, this ramp-up and ramp-down will not be always feasible in an iterative development as it will have to be done for each iteration, which can make resource management very difficult. Furthermore, as iterations may not be very large, the benefit of allocating resources when needed and freeing them when they are not needed might not be worth the effort as the slack time may be of short durations. Overall, optimum resource allocation in serial iterations is difficult and is likely to result in underutilization of resources.

The situation with the timeboxing process model is different. The team sizes for the different stages need not be same – in fact, they should be of the size needed to complete the task in the specified duration. Hence, we can allocate the "right" team size for the different phases, which generally will mean smaller team size for the initial and end phases and larger team size for the middle phases. These resources are "permanently" allocated to the project, which simplifies resource allocation and management. And the resources have, in the ideal case, a 100% utilization.

3.4 Unequal Stages

Clearly, the reality will rarely present itself in such a clean manner such that iterations can be fit in a time box and can be broken into stages of equal duration. There will be scenarios where these requirements will not hold. And the first possibility of a non-ideal situation is where the duration of the different stages is not the same – different stages take different time to complete.

As the pipelining concepts from hardware tell us [Hennessy and Patterson 1998], in such a situation the output is determined by the slowest stage, that is, the stage that takes the longest time. With unequal stages, each stage effectively becomes equal to the longest stage and therefore the frequency of output is once every time period of the slowest stage. In other words, the stage-time-unit is now equal to the duration of the longest stage, and the output is delivered once for each of this stage-time-unit. As each stage effectively becomes equal to the longest stage, the team for smaller stages will have slack times, resulting in resource under-utilization in these stages while the team with longest stage will be fully occupied.

As an example, let us consider a 3-stage pipeline of the type discussed above in which the different stages are 2 weeks, 3 weeks, and 4 weeks – that is, the duration of the time box is 9 weeks. For pipelined execution, such an iteration will first be transformed into an iteration of 12 weeks and 3 stages, each stage taking 4 weeks. Then, the iterations will be executed in parallel. The execution of different iterations from the perspective of different teams is shown in Figure 5 (W represents "work" and S represents "slack".)

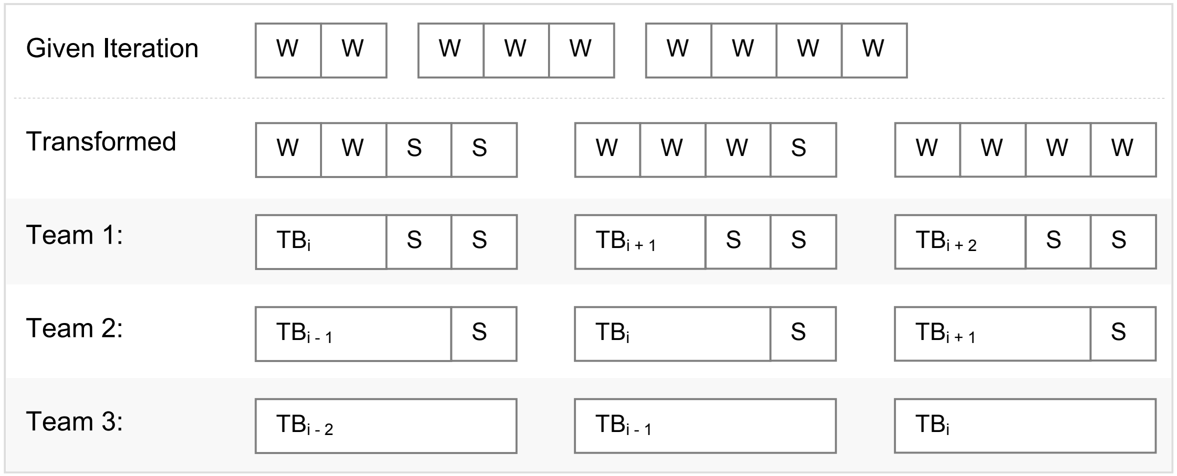

Figure 5: Execution with unequal stages

Note, however, that even with the output frequency being determined by the slowest stage, a considerable speedup is possible. In a serial iterative development, software will be delivered once every 9 weeks. With the timeboxing model, the slowest stage will determine the speed of execution, and hence the deliveries are done every 4 weeks. This delivery time is still less than half the delivery time of serial iterations!

However, there is a cost if the stages are unequal. As the longest stage determines the speed, each stage effectively becomes equal to the slowest stage. In this example, as shown in Figure 4, it means that the first and second stages will also get 4 weeks each, even though their work requires only 2 and 3 weeks. In other words, it will result in "slack time" for the teams for the first and the third stage, resulting in under utilization of resources. So, the resource utilization, which is 100% when all the stages are of equal duration, will reduce resulting in underutilization of resources. Of course, this wastage can easily be reduced by reducing the size of the teams for the slower stages to a level that they take approximately the same time as the slowest stage. Note that elongating the cycle time by reducing manpower is generally possible, even though the reverse is not possible.

3.5 Exceptions

What happens if an exceptional condition arises during the execution of a stage of some time box? The most common exception is likely to be that a stage in an iteration finishes either before or after its time. If a stage of an iteration completes before time, there is no problem – the team for the stage will have to wait for the remaining period (or till the previous stage of the next iteration finishes.) In other words, if the team for a stage I of an iteration K finishes ΔT time before its completion time, then in the worst case it will wait for this amount of time without any work (assuming that other stages do not slip.) Of course, if the stage I-1 of the iteration K+1 also completes before its time and is able to handover the work to the team for the stage I, then this team can start working and will have ΔT extra time to complete the task for the iteration K+1.

If the stage completes late, then there can be a cascade effect on other stages leading to each of the future stages getting delayed. Suppose that the stage I is late by ΔT. This means that the team for stage I+1 will have to wait for ΔT without any work, assuming that it finished its task for the earlier iteration in time. If the team for stage I+1 can complete the task in this reduced time, then there is no further effect on the execution. In other words, if one team is late, it can be compensated by the next team (or the next few teams) doing their tasks in lesser time.

If, however, each team continues to work for their fully allotted time, then there will be a cascading effect. The team for each stage, after completing its task for the previous iteration, will have to wait for ΔT to get the work for this iteration. This cascades to the last stage also, which starts and completes ΔT late. In other words, the net effect of the exception is that the delivery of this iteration is late by ΔT. After the delayed iteration, all future deliveries will come after every T/n time units (for a n-stage time box of T duration.) The effect on team utilization is that all the teams for stages subsequent to where the exception occurred have to wait without work for ΔT time, leading to reduced utilization of resources.

Of course there is the possibility that a stage is late not because of some exception in that stage but because the work itself is more than what can be done in allotted time and resources. If this is the situation, it implies that the estimation has not been proper for this iteration and more work has been committed than is feasible. The easiest solution to this situation is to detect is in the early stages and reduce the commitment for this stage.

Such an approach requires that project management carefully evaluate and do a causal analysis of slippage of any stage, as only a result of analysis will reveal whether it is a localized exceptional condition that will effect only this iteration or it is a situation that is in the design of the time box. If, however, the stage is inherently longer (or shorter), then it is not an exception and we have the situation of a longer (or shorter) iteration. The effect of this has been discussed earlier.

The timeboxing process model assumes that the resources for the different stages are separate, so there is no resource contention. In fact, as discussed above, this is one of the key features of this model – most other models share the resources in different stages. Situations can arise in which two concurrently executing iterations, each in a different stage, require some common resources. For example, for the requirement analysis of one iteration and deployment of another iteration, the same resources may be required. If timeboxing is to be used, such a situation should be avoided. However, in some exceptional conditions, this situation might arise.

If the shared resource that has caused this "conflict" is only a part of one team, then the net result of this sharing will be that the stage will be left with fewer resources and hence may take a longer time to complete. The effect of this we have discussed above. However, if the conflicting resources form the complete team of a stage, then we have a peculiar situation – there is no resource to execute the stage for the iteration. This case will finally translate to a "null" iteration being executed – this iteration consumes the time but no resources and produces no output. The effect of such a null iteration is that there will be no output when this null iteration completes. Also, the null iteration will cause the other teams which were not being shared or which had no conflicts to waste their time for this null iteration.

Other exceptional conditions must be possible. In each such case, one has to examine and see how to deal with it within the timeboxing framework. Note, however, all such exceptional situations will lead to some loss in terms of output or benefits. Consequently, if the exception is likely to occur very frequently – a situation implying that the process model was probably not suitable for the problem at hand – it may be worthwhile considering a different process model for the problem.

4 APPLYING THE MODEL

Effective use of the timeboxing model, as in any other process model, will require many practical issues to be addressed. In this section we discuss some other issues relating to deploying the model on projects.

4.1 Scope of Applicability

Like any other process model, the timeboxing model will be suitable only for some types of projects and some business contexts. The first clear requirement is that the business context should be such that there is a strong need for delivering a large number of features within a relatively short span of time. In other words, time to deliver is very important and is sought even if it implies that the team size will be large and there may be some wastage of manpower (due to slack times that may come in different situations.)

As for any iterative development approach, the model should be applied for projects where the requirements are such that some initial wish list is known to the users/clients, but there are many aspects that are dynamic and change with the competitive and business landscape. (If all requirements are clearly known in the start then the waterfall model will be most suitable and economic.)

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 that provides value to the users. Such a situation is frequently present in many commercial projects where a system already exists and the purpose of the project is to augment the existing system with new features for it. Another example of projects that satisfy this are many web-site development projects – generally some architecture is fixed early, and then the set of features to be added iteratively is decided depending on what the competition is providing and the perceived needs of the customer (which change with time).

To apply timeboxing, there should be a good feature-based estimation method, such as the bottom-up estimation technique described in [Jalote 2000]. With a good feature-based estimation model, the set of features that can be built in an iteration can be decided. As technology churn and unstable architecture make it harder to do feature-based estimation, projects where technology and architecture changes are frequent and likely are not suitable candidates for this process model.

The team size and composition is another critical parameter for a project using timeboxing. Clearly the overall project team should be large enough that it can be divided into sub-teams that are dedicated for performing the tasks of the different stages. However, to keep the management complexity under control, it is desirable that the team be not so large that coordination between different sub-teams and different time boxes become too hard to manage.

There is a side benefit of having separate teams for different stages. As the teams are separate, they need not be collocated. This means that the different stages of an iteration need not be done in one location – something that must happen if the same team is to work on all stages. Having different stages being executed by different teams permits outsourcing of different stages to different teams in different locations, thereby allowing the possibility of leveraging the global delivery models of modern multinational enterprises.

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. Such a situation may arise, for example, if only a few features are to be built, one (or a couple) in each iteration, each tied to some business need. In this case, as there is only one feature to be built in an iteration, that feature will determine the duration of the iteration.

4.2 Project Planning

One of the key activities in planning each time box will be selecting requirements that should be built in a time box. Remember, one dimension of planning does not exist anymore – the project manager does not have to do scheduling of the iteration or its stages as they are given to him. Also, in general, the effort and team sizes are also fixed. The quality requirement also remain the same. This means, that out of the four variables in a project, namely time, cost, quality, and scope [Beck 2000], the only variable that is free for a time box is scope. Hence, the key planning activity for a time box is what requirements to build.

As discussed above, this can be achieved if there is a good feature-based estimation model, and there is some flexibility in grouping the features to be delivered. This assumes that the processes being used for each iteration are stable and repeatable and that the effort distribution of each iteration among the different stages is approximately the same. With a stable and repeatable process, if we have a feature based estimation, we can estimate the impact of each feature on the effort needed and the schedule. If we have some flexibility in grouping the features to be built in an iteration, we can select the set that can together be built in a time box, as we know the total effort available in a time box.

In the project commencement stage, the key planning activity with this model is the design of the time box to be used. That is, the duration of the time box, the number and definition of the stages, and the teams for the different stages. Having a large duration of the time box will minimize the benefits of iterative development. And having too small a time box may imply too little functionality getting developed to the customer. Hence, the size of the time box is likely to be of the order of a few weeks to a few months. Frequently, the exact duration will be determined by business considerations.

The next obvious issue is how many stages should be there in a time box. The answer to this will clearly depend on the nature of the project. However, as too many parallel executions can make it difficult to manage the project, it is most likely that the model will be used with a few stages, perhaps between two and four. It may be pointed out that substantial benefit accrues even with two stages – the delivery time (after the first delivery) is reduced by half. At Infosys, we suggest a 3-stage time box, as discussed in the example above.

The team sizes for the different stages need to be carefully selected so that the resource utilization is high. We know that effort distribution among different stages is not uniform and that number of resources that can be utilized effectively is also not uniform [Basili 1980, Basili and Rombach 1994]. Generally, in a project, few resources are required in the start and the end and maximum resources are required in the middle. Within a time box, the same pattern should be expected. This means, that the team size for the first and the last stage should be small, while the size of the team for the middle stages should be the largest. The actual number, of course, will depend on the nature of the project and the delivery commitments. However, the team sizes should be such that the effort distribution among the different stages is reasonable.

A heuristic that can be used for selecting the team sizes for the different stages, once the time box duration and the stages are fixed, is given below. This heuristic tries to keep the resource utilization high, while preserving the timeboxing process model property of having approximately equal stages. It assumes that in the start, the project manger has a good idea of the effort estimate of each of the stages for a typical iteration.

1. Select the stage that is likely to take the longest in an iteration. Often, this may be the build phase. Estimate the effort needed to complete the work for this stage in a typical iteration.
2. Fix a stage duration from the time box duration and the number of iterations.
3. Using the effort estimate and the stage duration, determine the team size needed to complete this stage. Check that the team size and the duration are feasible. If not, adjust the time box duration and/or the team size.
4. From the finally selected time box duration and the finally selected team size for the stage, determine the total effort for that stage. Using the expected distribution of effort among different stages, determine the effort estimates for other stages.

5. For the remaining stages, use the effort estimate of a stage and the stage duration to estimate the size of the team needed for that stage.

This heuristic is essentially trying to ensure that the different stages are of equal length and that the team size and duration of the key-stage is manageable. Note also, as we first fix the critical stage which potentially takes the longest time, it means that for the other stages, we are basically elongating the stage length to make it equal to the longest stage, and then suitably allocating the resources so the resource utilization is high. Note also that increasing the duration of a task by reducing the number of resources is possible even though the reverse (that is, reducing the time by increasing the resources) is not always possible. With this heuristic, the duration of the stage is determined by the "slowest" stage, which becomes the "rate determining step". Resources are then adjusted accordingly.

A challenge that project managers could face is determining the relationship between the effort estimates as determined from the estimation model and the cost model of the project. This problem is more acute in timeboxing as the different teams may not be collocated and hence may have different cost-basis. Such an issue can slow down negotiation and may result in features for an iteration not being finalized in time. One possible solution to this is to ensure that there is agreement between the customer and the supplier before-hand about the estimation model and how a feature is considered to be of a particular cost. Tying the feature estimate to cost also provides the cost visibility of different requirements groupings.

4.3 Project Monitoring and Control

Monitoring and controlling a project which employs timeboxing is clearly going to be more complex than a serial iterative development. There are a few clear reasons for it. First, as discussed above, the team size has become larger and the division of resources stricter. This makes resource management within the project harder. There are other issues also relating to project resources – for example the HR impact of having one team performing the same type of activity continuously.

Second, monitoring is now more intense as multiple iterations are concurrently active, each having some internal milestones (at the very least, completion of each stage will be a milestone.) Generally, milestones are important points for monitoring the health of a project (for some example of analysis at milestones, the user is referred to [Jalote 2000, Jalote 2002].) In a timeboxing project, the frequency of milestones is more and hence considerably more effort needs to be spent in these analysis. Furthermore, project management also requires more regular monitoring and making local corrections to plans depending on the situation in the project. Due to the tight synchronization among stages of different iterations, making these local corrections is much more challenging as it can have an impact that will go beyond this iteration to other time boxes as well.

The project will require tight configuration management as many teams are working concurrently. A basic CM challenge is to ensure that a common consistent source code is made available to the teams for the different stages. Daily Build and Smoke Test [McConnel 1996] is a technique that can be used for ensuring a common consistent source code. With this technique, a software product is completely built very frequently (every day or every few days) and then put through a series of tests. These tests typically include compiling all files, linking all files, ensuring that there are no showstopper bugs, and that the code passes the smoke test. The smoke test are typically test scripts that exercise the key functionality of the entire system. The test should be thorough enough that if the build passes the smoke test, it should mean that it is sufficiently stable and that changes made did not affect the functionality of any other portion of the system.

Despite frequent synchronization, there will be situations where multiple simultaneous changes done to the same file. This is quite likely to happen between the build and deployment stages as the bugs found during deployment are typically fixed by the deployment team (though in consultation with the build team.) To handle this situation, the reconciliation procedures (i.e. procedures that are used to reconcile two changes if they are done concurrently [Jalote 2000]) need to be solid and applied regularly.

Another CM requirement is the need to have multiple areas whose access is strictly controlled. Typically these areas can be:

- Test – The software that has been put through various tests and is ready for integration and system testing.
- Preproduction – a copy of the production system to be used for testing.
- Production – The actual area where the software is running when in use.

Typically only the executable version and not the source code is available in either Preproduction or production environments. With multiple time boxes occurring at the same point in time, there is a necessity to ensure that the test, preproduction, and production environments are "clean" and strictly controlled. A lack of discipline in access control, which may be acceptable in non-parallel development, would result in confusion and rework and because multiple timeboxes are involved.

Overall, with timeboxing, the project management needs to be very proactive and tight. This can sometimes lead to decisions being taken that can have adverse impact. For example, a project manager, in order to complete in a time box might compromise on the quality of the delivery. This also implies that a project using timeboxing requires an experienced project manager – an inexperienced project manager can throw the synchronization out of gear leading to loss in productivity and quality, and delayed deliveries.

4.4 Handling Changes

We know that requirements change and that such changes can be quite disruptive [Boehm 1987, Jones 1996]. Changes can vary from being very minor, such as a change to an error message being displayed in the system, to a significant requirement, such as to change the architecture or design. Major change requests could have a detrimental impact on the project by making work that was done in the past redundant and add new work that requires significant effort. Thus changes tend to be one of the major risks for any project on a tight schedule. Hence, most development processes add a change management process for accepting a change request, analyzing it, and then implementing it, if approved. Such a process is necessary if the process is not intrinsically capable of allowing change, as is the case with the waterfall model. A change management process is needed to ensure that changes do not have a very detrimental impact on the project. One such process works as follows [Jalote 2000]. When a change request arrives, it is logged into a system. It is then analyzed for its impact, that is, what will be the impact on the project if the change is implemented. Such an impact analysis will typically detail out the scope of the change, what are the risks, and what is the impact on the cost and schedule for implementing the change. Based on the impact analysis, a decision is taken whether to incorporate the change or not.

With timeboxing, requirement change requests can be handled in a somewhat different manner – unless a request is extremely critical, the change request is passed on to the next possible time box. That is, the change request comes as a new requirement in a future time box. Since the time boxes are likely to be relatively short, deferring the requirement change request to the next time box does not cause inordinate delay.

The same can be done for the defects that are found after deployment. Such defects are viewed as change requests. If the defect is such that it is hurting the business of the organization and whose repair cannot be delayed, then it is fixed as soon as possible. Otherwise, its fixing is treated like a change request and is fixed in the next possible iteration.

There is thus a significant benefit that a timebox can be executed without having to invest time and effort in incorporating modifications that arise due to change requests.

However, with timeboxing ensuring that related documents like design and test cases are updated and maintained under the face of changes becomes even more critical than in other process models because with overlaps and multiple teams, things can get outdated very quickly resulting in incorrect specifications, design or test cases that will directly impact the quality. The rework because of the lack of maintenance of documentation can also have a much harder impact on the timelines of a stage or time box. It is therefore necessary to have tighter management and control of requirements, design and test documentation. Frequent baselining, supported by a traceability matrix that ensures that a requirement can be traced from the requirements document to a test document and vice versa, a change matrix in all documentation that clearly identifies the changes that were made and ties it to the specific change request are steps that can help.

4.5 Localized Adjustments in Time Boxes

We have been assuming that the team for each stage is fixed. However, the basic requirement for the model to operate smoothly is that each stage should finish in its allotted time. If the number of resources in the sub-team of a stage changes across time boxes, there is no problem as far as the model is concerned (though the resource management may become harder.) Clearly then, for some time boxes, additional resources can be added in some stages, if desired. This type of adjustment might be considered when, for example, the system "core" is to be developed, or some large feature is to be developed that requires more work than what a time box can provide, etc. This adjustment can also be used to first start the project with fewer resources and then when the project picks up and there is experience with the model, the resources for the different stages are increased while preserving the time box structure.

Similarly, if some time box has lesser work to be done, some resources can be freed for that time box. However, in practice, if the work is lesser, the chances are that the team composition will not be changed for temporary changes in work (it is very hard to find temporary work in other projects), and the adjustment will be made by putting in more or less hours.

Local adjustment of stages is also possible. For example, in some time box, the team of a stage can finish its work in lesser time and “contribute” the remaining time towards the next stage, effectively shortening one stage and correspondingly lengthening the other one. This local adjustment also has no impact on the functioning of the model and may be used to handle special iterations where the work in some stages is lesser.

4.6 Refactoring

Any iterative development, due to the fact that design develops incrementally, can eventually lead to systems whose architecture and designs are more complex than necessary. This problem can be handled by techniques used for refactoring, wherein code from the previous iterations are redesigned without changing functionality, but paves the way for extensibility and better design for the future iterations. Martin Fowler defines refactoring as “Refactoring is a process of changing a software system in such a way that it does not alter the external behavior of the code yet alters its internal structure” [Fowler et. al. 1999]. The goal of refactoring is therefore to pave the way for software to be extended and modified more easily without adding any new functionality. An iteratively developed system should undergo some refactoring otherwise it may become too complex to easily enhance in future. Some methodologies, like the XP, explicitly plan for refactoring. In XP, for example, refactoring can be done at any time.

Refactoring provides a catalog of techniques, such as using “wrapper classes”, to reuse code previously written while still improving the design of the system. The refactoring process can be viewed like a development process except that the activity in the different stages is different – the analysis is largely a “discovery” of redesign opportunities in the existing code, the design is actually “re-design” of the code for the purpose of improved design to reduce complexity and enhance extensibility, the coding is mostly “code transformation”, and the testing is really “re-testing” for the same functionality that was originally intended.

Though refactoring suggests that it is a process of evolution of the quality of the code and therefore a continuous process, in the timeboxing model the most natural way to perform refactoring will be to consider it as the goal of one of the time boxes. That is, in some time box, the basic objective is to do refactoring and not to add new features (or minimal features). In this time box, refactoring undergoes the same process as developing new features – i.e. through its stages, except, as discussed above, the activity within each stage will be performed differently. So, if the pipeline has the three stages given earlier, then in the time box in which refactoring is done, first the requirements for refactoring will be decided. The team for requirements will analyze the system to decide what part of the system can and should be refactored, their priorities, and what parts of the system will be refactored in this time box, etc. In the build stage, the refactoring will actually be done and tested, and in the last stage, the new system will be deployed. So, for all practical purposes, refactoring is just another iteration being done in a time box, except that at the end of this iteration no new (or very little) functionality is delivered – the system that is delivered has the same functionality but is simpler (and perhaps smaller.)

5 AN EXAMPLE

In this section we discuss a real commercial project on which this process model was successfully applied. We firs give the background about the project and the commercial and other constraints on it and then describe how these constraints were accommodated within this process model. We also discuss the metrics data collected.

5.1 Background

A US-based e-store (the customer), specializing in selling sporting good, approached Infosys, for further developing its site. They already had a desired list of about 100 features that they wanted their site to have in the next 6 months, and had another set of features that they felt that they would need about a year down the road. This list was constantly evolving with new features getting added and some getting dropped depending on what the competitors were doing and how the trends were changing.

As many of the features were small additions, and as the list of features was very dynamic, Infosys proposed to use the timeboxing model, with a 3-stage time box (of the type discussed above). To keep the costs low, it was decided that the offshore model for development will be used. In this model, the team in India will do the development, while analysis and deployment will be done by teams at the customer's site. Furthermore, to reduce costs further, it was decided that that total effort of the first and the third stages would be minimized, while passing most of the work on to the build stage.

After studying the nature of the project and detailed processes for the three stages, the actual duration chosen for the stages was as follows: 2 weeks for requirements, 3 weeks for build, and 1 week for deployment. The initial team size for the three stages was selected as 2 persons, 6 persons, and 2 persons respectively. It was felt that these times and team sizes are sufficient for the tasks of the different stages. (With these durations and team sizes, the effort distribution between requirements, build, and deployment is 4: 18: 2, which is consistent with what Infosys has seen in many similar offshore based development projects.)

The work was also classified using the matrix shown in Figure 6. As shown in the figure, features were classified on the dimensions of their complexity and business payoff, and for each dimension they were classified as "high" or "low". Features in the top left quadrant, which represent those that have the maximum business payoff with the least technology complexity were taken up for development in the earlier iterations. In subsequent iterations, the features in the other quadrants were chosen.

Technology Complexity

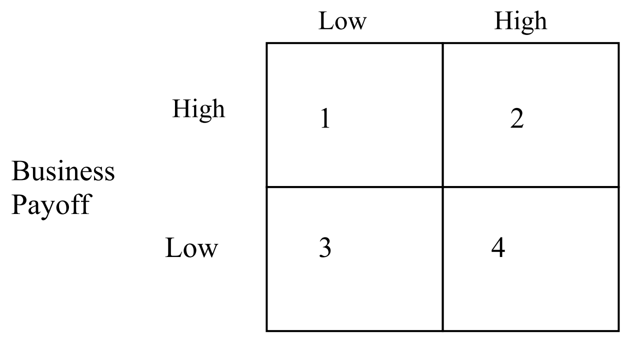

Figure 6: Prioritizing the features

On at least one occasion, one feature was too big to be completed within one iteration. This feature was split into two sub-features and executed across two consecutive iterations. However, the formal deployment of the two sub-features was done together.

5.2 Process Design

In this project, the size of the different stages is not equal. As discussed above, with unequal stages, the delivery frequency is determined by the longest stage. In other words, with these durations, by using the timeboxing model in this project, delivery will be done (except for the first one) after every 3 weeks, as this is the duration of the build stage which is the slowest stage in the time box. We have also seen that unequal stages result in slack times for the dedicated teams of the shorter stages. In this project, the slack times for the requirements team will be 1 week and the slack time for the deployment team will be 2 weeks. Obviously, this resource wastage has to be minimized.

So, we have a 3-stage pipeline, each stage effectively of 3-week duration. The execution of the different time boxes is shown in Figure 7. The resource planning was done such that the requirements team executed its task in the 2nd and the 3rd weeks of its stage (with the 1st one as the slack), and the deployment team executed its task in the 1st week (with the 2nd and 3rd as the slack.)

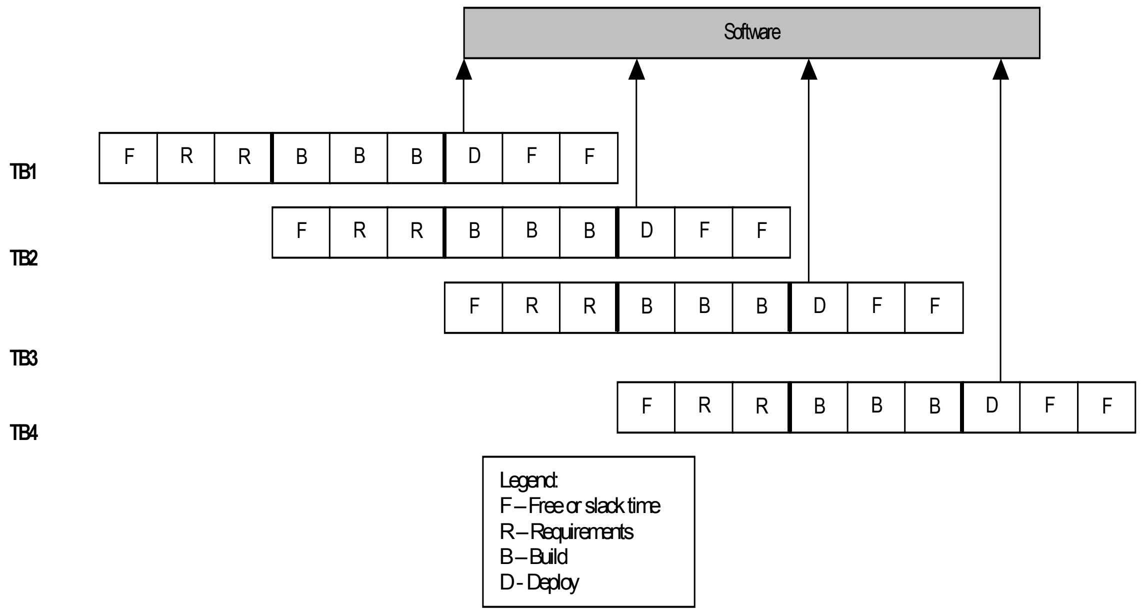

Figure 7: Time boxed execution of the project

Figure 3 shows four different time boxes – R refers to the requirements activity, B to the build activity, and D to the deployment activity. The boundaries of the different stages are shown, with each stage being of 3 weeks. The activity of each week is shown in the diagram. F is used to represent that the team is free – i.e. to show the slack time. The first delivery takes place 6 weeks after actually starting the iteration. (Note that this made possible by organizing the slack times of the first stage in the start and the last stage towards the end.) All subsequent deliveries take 3 weeks, that is, deliveries are made after 6 weeks, after 9 weeks, after 12 weeks, and so on.

As mentioned above, this execution will lead to a slack time of 1 week in the first stage and 2 weeks in the third stage. This resource wastage was reduced in the project by properly organizing the resources in the teams. First we notice that the first and the last stage are both done on-site, that is, the same location (while the second stage is done in a different location.) In this project as the slack time of the first stage is equal to the duration of the third stage, and the team size requirement of both stages is the same, a natural way to reduce waste is to have the same team perform both the stages. By doing this, the slack time is eliminated. It is towards this end that the slack time of the first stage was kept in the start and the slack time of the 3rd stage was kept at the end. With this organization, the slack time of the 1st stage matches exactly with the activity time of the third stage. Hence, it is possible to have the same team perform the activities of the two stages in a dedicated manner – for 1 week the team is dedicated for deployment and for 2 weeks it is dedicated for requirements.

With this, we now have two teams – on-site team that performs the requirements stage and the deployment stage, and the off-shore team that performs the build stage. The process worked as follows. The offshore team, in a 3-week period, would build the software for which the requirements would be given by the on-site team. In the same period, the on-site team would deploy the software built by the offshore team in an earlier time box in the 1st week, and then do the requirements analysis for this time box for the remaining 2 weeks. The activity of the two teams is shown in Figure 8. As is shown, after the initial time boxes, there is no slack time.

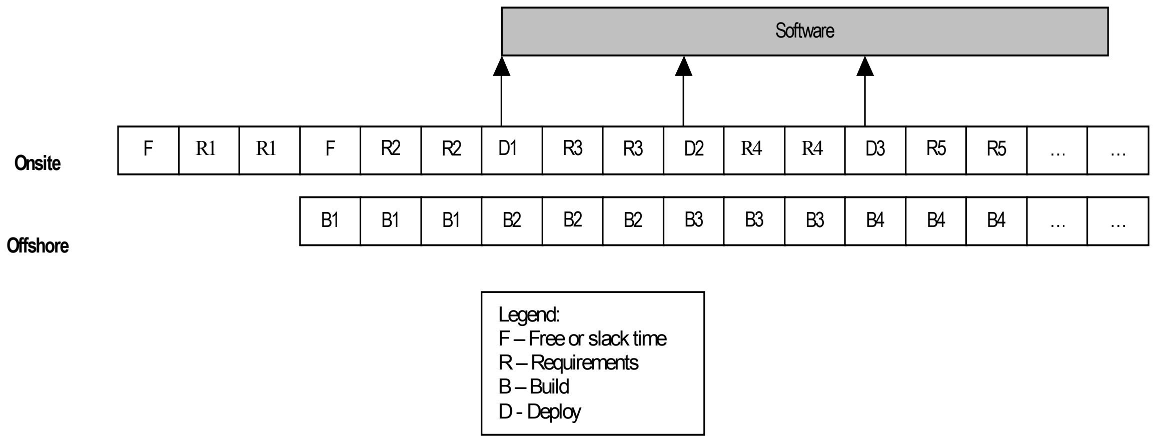

Figure 8: Tasks of the on-site and offshore teams

We can also view this in another manner. We can say that the number of resources in the sub-team for the requirements stage is 4/3 and the number of resources in the sub-team for the deployment stage is 2/3 (or that two resources working $2/3^{rd}$ of their time for requirements and $1/3^{rd}$ for deployment.) With these team sizes, to perform the work of the stages which has been specified as 2 x 2 = 4 person-weeks for requirements and 2 x 1 = 2 person-weeks for deployment, full 3 weeks are required for these stages. In other words, with team sizes of 4/3, 6, and 2/3, respectively for the three stages, we now have a time box with three stages of 3 weeks each. That is, by dividing the on-site team into "dedicated" resources for the two stages, we have the ideal time box with each stage requiring 3 weeks.

It should be added that the team size was gradually ramped up while preserving the basic structure of the time boxes. Starting with a team size of 6 (in offshore) in Iteration 1, the team was ramped up to 8 persons in Iteration II and to 12 persons by Iteration V. Consequently the number of features handled per Iteration also increased from 3 to 12 between Iteration I and Iteration V.

5.3 Project Management

As discussed earlier, once the basic structure of the timeboxing process model is fixed, one of the key issues is to have a proper estimation procedure that can permit feature-to-cost mapping. In this project, the standard bottom-up effort estimation process of Infosys was used [cmm, swpm]. In this approach, estimation starts with identifying the main features to be built and classifying them as simple, medium, or complex based on some defined criteria. When the number of different types of features is known, using some baseline cost which is determined from data from past projects, the total effort for coding and unit testing is determined. Then from this effort, the effort for other phases is determined as a percentage of coding cost.

In this project there are only three major phases, unlike the standard process used at Infosys. Hence, the percentages were suitably combined to determine the distribution among these three stages. This distribution was then used to determine the effort for the other two stages from the build stage (the build stage is approximately the same as the coding and unit testing stage of the standard process.)

For applying the estimation model, every feature was broken down in terms of the pages and components that needed to be created or changed. The simple, medium, and complex classification was clearly specified, using experience with similar projects in the past. The first few iterations were consciously underestimated to account for the learning-curve and to reduce the risks. However, a systematic adjustment was done for the subsequent iterations which corrected this bias. At the end of every iteration, effort consumed is compared with the estimates on the project, and the Adjustment Factor (AF) for an iteration computed as the ratio of the actual effort to the estimated effort. We arrive at the weighted adjustment factor for estimation for the N^{th} iteration by using the formula 0.5*AF[N-1] + 0.30 AF[N-2] + 0.20 AF[N-3]. This adjustment factor was applied in the N^{th} iteration estimate. This ensured that errors in the estimation methodology were addressed in the subsequent Iterations. Interestingly, the value of Adjustment Factors for the later iterations was pretty close to 1.

The number of people offshore on the project varied between 8 to 15 for different iterations. This change was made on the basis of the estimate done for every iteration. However, the time box schedule remained the same. This ability to ramp up/down the amount of resources without compromising the delivery dates was very critical to the success of the project. As resource allocation cannot be arbitrarily expanded or shrunk without affecting the schedule, it was agreed with the client that a minimum offshore-team size of 6 would be maintained, and continuity of the key players will also be maintained across iterations. While increasing the team size so as to build more features, a sanity check was made whether the features that are being planned can be built within the build effort available for the stage.

For the project to proceed smoothly, since two different sites are involved, an on-site coordinator for the project was assigned. The main tasks for the coordinator were to coordinate with the users and other teams in the user organization and be the contact point for the project and the main communication channel between the teams in the two locations. He was also the point-person for reporting and escalation management. This position considerably helped in keeping the coordination smooth and keep the pipeline going.

Risk management and its planning is another key activity in a project. It is quite possible that a project using the timeboxing approach will experience different types of risks. The risk management approach followed in the project was also the standard organization approach (more details on this are given in [Jalote 2000, Jalote 2002].) In this approach, the risks to the project are first enumerated. They are then ranked for their probability of occurrence and the impact on a scale. The project generally focuses on high probability and high to medium impact risks for mitigation. The top few risks and their mitigation approach is shown in Table 1.

Table 1: Top risks and their risk mitigation plan

Risk	**Prob.**	**Impact**	**Mitigation**
Changes in requirements	High	High	Impact analysis will be done based on impact on schedule. A decision will be taken to accommodate in the current or the next iteration
New technology	Med	High	Training of team members in the technologies used. Identified integration issues as part of the

			requirements stage and work with product vendors to find a solution. Additionally specify the integration pattern early.
To complete requirements in time for build	Med	Med	Freeze requirements up-front; Have tight escalation rules for requirements clarifications; Keep a close interaction with the customer's representative; If answers not provided within specified timeslots, shift features to subsequent iterations.
Code reconciliation	Med	Med	Daily build and smoke test; strictly follow the CM procedures; frequent reconciliations.

As we can see, the top risk is changes in requirements as it can derail the process easily. The mitigation approach is to do an impact analysis for each change request with a special focus on its impact on the schedule. Then based on the impact analysis, a decision will be taken whether to include it in this iteration or include it in the next one(s). Notice that completing requirements within the time allocated for that stage is also a risk to the project. Interestingly, one of the risk mitigation strategy for this also leverages the iterative development approach with short delivery times – if some clarifications were holding up specification of some requirement, that requirement was shifted to the next time box. Code reconciliation is also a risk – handling of this we have discussed earlier in the chapter.

5.4 Configuration Management

The primary purpose of configuration management (CM) in a project is to mange the evolving configuration of the various parts of the system like source files, documentation, plans, etc. A proper CM in a project will keep track of latest versions and state of different documents, allow for updates to be undone, prevent unauthorized changes, collect all sources related to a release, etc. Traditionally, the main mechanisms used to support CM are naming conventions, version control, and access control. In the project, as multiple iterations are simultaneously active, it was realized that the CM procedures will have to be enhanced and that versions and baselines will have to be handled carefully.

To allow the timeboxing model to work smoothly, at any given time the project supported a minimum of three baselines – production support baseline, acceptance test baseline, and the development baseline. The development baseline represents the state of the current development – it goes through a series of versions till it is ready for release, i.e. when the development team (i.e. the offshore team) is ready to release it for acceptance testing and deployment. The released version is taken as the starting baseline for acceptance testing. The acceptance test team (i.e. the onsite team) runs the acceptance tests and makes changes to the acceptance test baseline locally, creating a series of versions of this baseline. When all the defects are fixed and the software is ready for production release, the acceptance test baseline becomes the starting baseline for production. The deployment team (i.e. the onsite team) works with the production baseline and may make local changes to the production baseline, if needed, creating newer versions.

Besides the migration of baselines from development to acceptance testing to production, there is a reverse movement also – the starting baseline for any iteration is the latest production baseline. Due to the relationship between the three baselines, it is clear that the changes being done to a baseline will affect other baselines. To ensure that changes are not lost, changes made to the production baseline and the acceptance test baseline is regularly propagated to the development baseline. This process is shown in Figure 9.

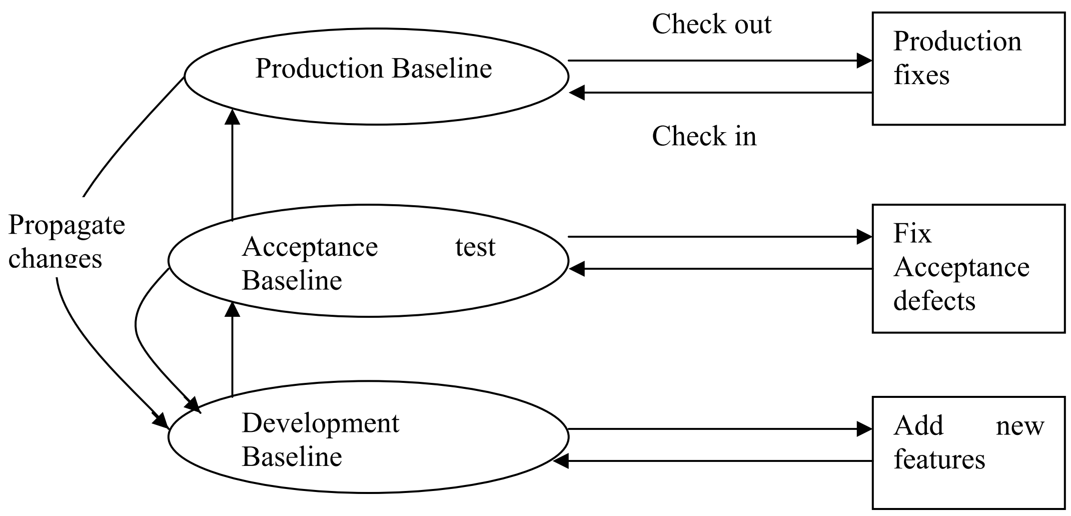

Figure 9: Baselining process

In the CM plan of the project, a clearly defined directory structure for managing the program assets was also planned. A baseline audit was done regularly, and before any release. A quality control check about the configuration procedures was performed once a month. At the start of every iteration, a synchronization of the complete development baseline was done with the production baseline. This synchronization is done to ensure that any changes made in other applications and modules (which may have been developed by a different team or a different vendor) on which this project depends are reflected in the sources used by the development team.

Requirement changes were handled as per the model – unless urgent, they were pushed to the next available time box. As the time boxes are small, there were no problems in doing this. For bug fixes also this was done. Unless the bug required immediate attention (in which case, it was corrected within 24 hours), the bug report was logged and scheduled as a part of the requirements for the next time box.

5.5 Data Collection and Analysis

As mentioned above, with the timeboxing process model, a project has to be very tightly monitored and controlled. For having an objective assessment of the health of a project, measurements are necessary. The basic measurements needed for monitoring a project are size, effort, schedule, and defects [refs]. At Infosys, there is are defined standards for collecting this data on projects. For effort, there is a weekly activity report system in which effort is logged in a project against different activities. For defects, there is a defect control system in which defects are logged along with their origin, severity, status and other properties. The reader is referred to [Jalote 2000, Jalote 2002] for further details on how these measurements are recorded in Infosys.

Even thought this project used a different process model than was generally used, it followed the same measurements and analysis approach, and the standard tools were used. One main difference was that effort and defects were also tied to an iteration, not just to a project.

The data was analyzed at milestones using the standard milestone analysis template of the organization. In this analysis, an actual vs estimated analysis is done for effort and schedule, and customer complaints and change requests are examined. In this project, for an iteration there are only two milestones. In the project, the purpose of the milestone analysis is to properly control the execution of the iteration under progress. As the weekly status reporting mechanism was providing sufficient visibility and since the duration of an iteration was short, later on the milestone analysis was stopped.

At the end of each time box, an analysis of the time box was done, much in the same way a postmortem analysis is done for a project [Basili and Rombach 1994, Chikofsky 1990, Collier 1996]. In general, a postmortem analysis is done to collect data and lessons learned that will benefit future projects and enhance the knowledge base of an organization. However, in an iterative development, the postmortem of one iteration can benefit the remaining iterations of the project itself. In this project, the standard postmortem analysis of the organization was followed [Jalote 2000]. The results were reviewed at the start of the next iteration. In this project, the analysis showed that the estimation approach being used was quite accurate and the process design was stable and the project was able to meet the commitments in each of the iterations.

6 CONCLUSIONS

Iterative software development is now a necessity, given the velocity of business and the need for more effective ways to manage risks. The commonly used approach for iterative development is to decide the functionality of each iteration in the start of the iteration and then plan the effort and schedule for delivering the functionality in that iteration.

In the timeboxing model, the development is done in a series of fixed duration time boxes – the functionality to be developed in an iteration is selected in a manner that it can “fit” into the time box. Furthermore, each time box is divided into a sequence of approximately equal duration stages. There is a dedicated team for each stage, and as a team completes its task for a time box, it hands over the development of the iteration to the team for the next stage, and starts working on its task for the next time box (which is handed over to it by the team of the previous stage.) The teams for the different stages all work concurrently, though each team works on a different iteration. And in a n-stage time box, n different iterations are simultaneously executing. The timeboxing model essentially emulates the pipelined execution of instructions in hardware which is now almost universally used.

Due to parallelism in the execution of different iterations, the turnaround time for each release is reduced substantially – for a n-stage time box, the average turnaround time for a time box of T time units duration will be T/n. That is, instead of each iteration completing after T time units, which is the case if the iterations are executed serially, with timeboxing, iterations complete, on an average, after every T/n time units. The execution speeds up by a factor of n, under ideal circumstances. The throughput, that is the amount of software delivered per unit time, also increases by a factor of n.

The total effort for each iteration, however, remains the same, and so does the productivity (total output per unit effort.) This compression of average delivery time without any increase in productivity comes as the total team size in the timeboxing model is larger. If the size of the team for each stage is the same, then the team size with a n-stage time box will be n times the team size needed if the same team is to execute all stages of an iteration, as is generally the case when iterations are performed serially. Hence, the timeboxing provides a structured approach of utilizing additional manpower in a project for reducing the software delivery times – something that is not generally possible.

We have discussed the impact of unequal stages and exceptions on the execution of this model, and have shown that even if the situation is not ideal with all stages fully balanced, a considerable speedup is still possible, but with a reduction in resource utilization. The model allows easy handling of change requests and in some ways simplifies the resource allocation issues by eliminating the need of resource ramp-up and ramp-down. Refactoring, which is needed in an iterative development, is treated as an iteration in its own right.

An example of applying the process model to a commercial project was also discussed. In the project, the duration of the three stages was set at 2 weeks, 3 weeks, and 1 week respectively. The initial size of the teams for these stages was 2 persons, 6 persons, and 2 persons. The first and the third stage were done at the customer's site while the second stage was done off-shore. With this time box, the first delivery was done after 6 weeks; subsequent deliveries were done after every 3 weeks. To minimize the slack time in the first and third stage, the same team performed the two stages – the resource requirements of these stages are such that it eliminated the slack times.

The timeboxing process model is applicable in situations where it is possible to construct approximately equal sized iterations, and where the nature of work in each iteration is quite repeatable. The development process for each iteration should also be such that it can be partitioned into clean stages with work of a stage requiring little interaction with people who perform the other stages. However, this process model makes the task of project monitoring and control more difficult and challenging. Consequently, it may not be well suited for projects with large teams.

So far the experience in using the model is very positive. However, further experience is needed to develop better guidelines for applying this model on real projects. There are many issues relating to project planning and project monitoring that need to be further studied. Experience is also needed to better understand the scope of applicability as well as the nature of exceptions that might occur, how to handle them, and the impact of such exceptions on the performance of this process model.

ACKWNOWLEGEMENTS. The authors greatly acknowledge the help provided by V.T. Peetamber, who was the project manager for the case study, and … for providing valuable inputs for improving this chapter.

REFERENCES

[Basili and Turner, 1975] V. R. Basili and A. Turner, Iterative enhancement, a practical technique for software development, *IEEE Transactions on Software Engg.,* 1(4), Dec 1975.

[Basili, 1980] V. R. Basili, Ed., *Tutorial on Models and Metrics for Software Management and Engineering,* IEEE Press, 1980.

[Basili and Rombach, 1994] V. R. Basili and H. D. Rombach, The experience factory*, The Encyclopedia of Software Engineering*, John-Wiley and Sons, 1994.

[Beck, 2000] K. Beck, *Extreme Programming Explained*, Addison Wesley, 2000.

[Boehm, 1987] B. W. Boehm. Improving software productivity, *IEEE Computer,* Sept. 1987, 43-57.

[Boehm, 1988] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61--72, May 1988.

[Boehm, 1981] B. W. Boehm. *Software engineering economics*. Prentice Hall, Englewood Cliffs NJ, 1981.

[Brooks, 1975] F. P. Brooks, *The Mythical Man Month,* Addison Wesley, Reading, MA, 1975.

[Chikofsky, 1990] E. J. Chikofsky, Changing your endgame strategy, *IEEE Software*, Nov. 1990, pp. 87, 112.

[Cockburn, 2001] A. Cockburn, *Agile Software Development*, Addison Wesley, 2001.

[Collier, 1996] B. Collier, T. DeMarco, and P. Fearey, *A* defined process for project postmortem review, *IEEE Software*, pp. 65-72, July 96.

[Fowler et.al., 1999] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Reading, MA: Addison-Wesley, 1999.

[Hennessy and Patterson, 1996] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach,* Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[Jalote, 2000] P. Jalote, *CMM in Practice – Processes for Executing Software Projects at Infosys*, SEI Series on Software Engineering, Addison Wesley, 2000.

[Jalote, 2002] P. Jalote, *Software Project Management in Practice,* Addison Wesley, 2002.

[Jalote et. al.] P. Jalote, Timeboxing: A process model for iterative software development, *Journal of Systems and Software,* To appear.

[Jones, 1996] C. Jones, Strategies for managing requirements creep, *IEEE Computer,* 29 (7): 92-94. 1996.

[Kerr and Hunter, 1994] J. Kerr and R. Hunter, *Inside RAD – How to Build Fully Functional Computer Systems in 90 Days or Less*, McGraw Hill, 1994.

[Kruchten, 2000] P. Kruchten, *The Rational Unified Process – An Introduction*, Addison Wesley, 2000.

[Larman, 2002] C. Larman, *Applying UML and Patterns,* 2nd Edition, Pearson Education, 2002.

[Larman and Basili, 2003] C. Larman and V. R. Basili, "Iterative and Incremental Development: A Brief History", June 2003, IEEE Computer.

[Malotaux] N. Malotaux, "Evolutionary Project Management Methods", www.malotaux.nl/nrm/pdf/MxEvo.pdf.

[Martin, 1991] J. Martin, *Rapid Application Development,* Macmillan Publishing Co., 1991.

[McConnel, 1996] S. McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996.

[Royce, 1970] W. W. Royce, Managing the development of large software systems, *IEEE Wescon,* Aug. 1970, reprinted in *Proc. 9th Int. Conf. on Software Engineering (ICSE-9)*, 1987, IEEE/ACM, pp. 328-338.

[SEI, 1995] Software Engineering Institute, *The Capability Maturity Model for Software: Guidelines for Improving the Software Process*, Addison Wesley, 1995.

[Stapleton, 2003] J. Stapleton, editor, *DSDM – Business Focused Development,* Addison Wesley, 2003.