Timeboxing: A Process Model for Iterative Software Development

Pankaj Jalote[†], Aveejeet Palit
Priya Kurien, V. T. Peethamber
Infosys Technologies Limited
Electronics City
Bangalore - 561 229; India
Fax: +91-512-590725/590413
Contact Address: 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 this paper we propose the *timeboxing* model for iterative software development in which each iteration is done in a time box of fixed duration, and the functionality to be built is adjusted to fit the time box. By dividing the time box 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

Software projects utilize a process to enable execution of the engineering tasks to achieve the goal of delivering a software product that satisfies the user requirements. 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 most influential and commonly used process model is the waterfall model, in which the different phases of requirements specification, design, coding, and testing are performed in sequence. This model was first proposed by Royce [16] 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

[†] P. Jalote's current address: Dept of CSE; Indian Institute of Technology; Kanpur – 208016; India.

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 [7]. The biggest drawback with the waterfall model was 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 (an example of a change management process in an organization can be found in [13].) Another key limitation is that it follows the "big bang" approach – the entire software is delivered in one shot at the end. And till the end, no working system is delivered. This entails heavy risks, as the users do not know till the very end what they are getting.

To alleviate these two key limitations, an iterative development model can be employed. In an iterative development, software is built and delivered to the customer in iterations – each iteration delivering a working software system that is generally an increment to the previous delivery. Iterative enhancement [1] and spiral [6] are two well-known process models that support iterative development. More recently, agile methods [10] and XP [4] also promote iterative development – in fact small iterations is a key practice in the XP methodology.

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. 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 takes more time than a waterfall model.) If we wish to reduce the total development time, a natural approach will be 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. 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 [15].

In this paper, we propose the *timeboxing* process model that takes the concept of parallelism between different iterations further and employs the pipelining concepts [12] to reduce cycle time. In this model, iterative development is done in a set of fixed duration time boxes. That is, in each iteration, functionality developed is what can be "fit" into the time box. Each time box is divided into stages 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.

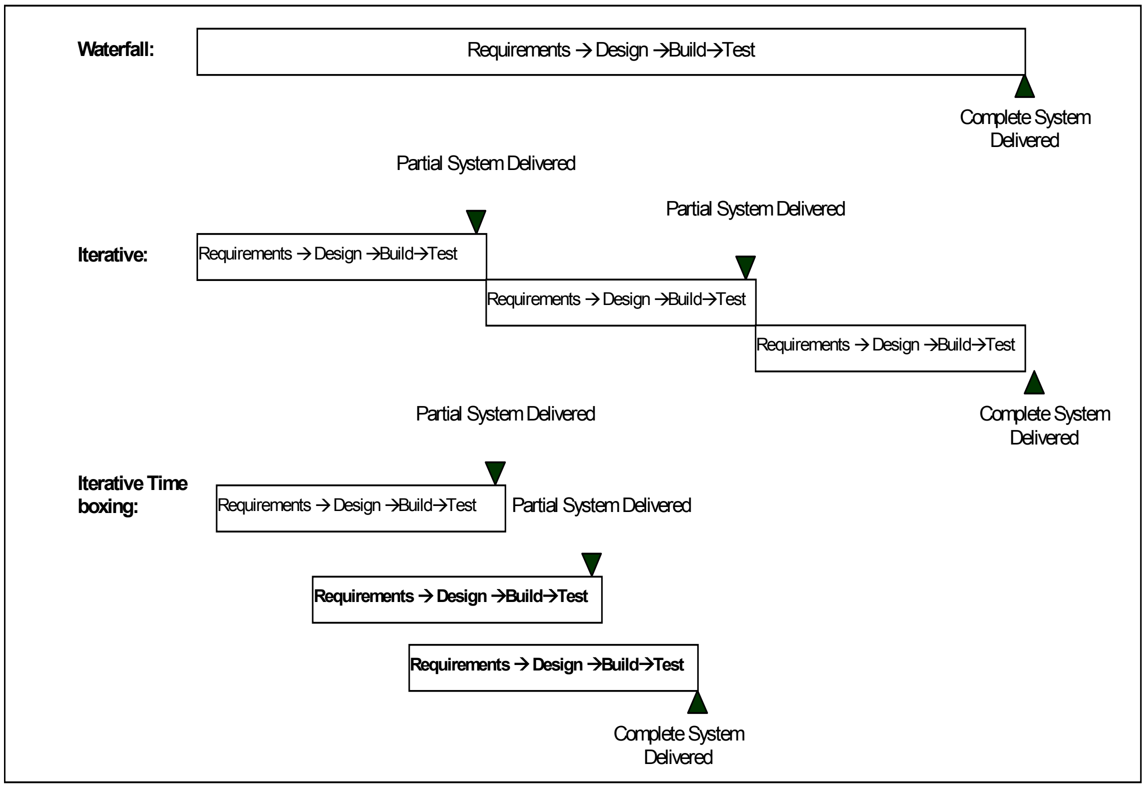
Waterfall:
Requirements → Design →Build→Test
Complete System Delivered
Partial System Delivered
Partial System Delivered
Iterative:
Requirements → Design →Build→Test
Requirements → Design →Build→Test
Requirements → Design →Build→Test
Complete System Delivered
Partial System Delivered
Iterative Time boxing: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

Figure 1: Waterfall, Iterative, Timeboxing Models

The concept of using time boxes for iterations has been discussed informally among developers for quite some time (for example, one of the reviewers of the paper indicated that he has been using it for years.) A key contribution of the paper is that we have formalized this concept and provided a conceptual framework that is grounded in the pipelining concepts developed to speed up execution of instructions in processors. We have also employed this process model successfully for executing real-life, commercial projects.

We believe that the 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.

The paper is organized as follows. In the next section we discuss the basic timeboxing model, 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 3, 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 4 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 paper ends with conclusions.

2 THE TIMEBOXING MODEL

In timeboxing, as in other iterative development approaches, some software is developed and a working system is delivered after each iteration. 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.

2.1 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. Furthermore, the model 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.)

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. This is quite different from many other models where the implicit assumption is that the same team (by and large) performs all the different tasks of the project or the iteration.

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 need 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 it can be passed to the team for next stage, and the team 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.

2.2 Pipelined Execution

With fixed duration for execution of an iteration, the model renders itself to pipelining (the reader is referred to [12] for concepts on pipelining in hardware.) 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 hardware for executing a stage in an instruction). Let the size of the team dedicated for stage S_i be R_i, representing the number of resources assigned to this stage.

The team of each stage has T/n time available to finish their task for a time box, that is, the duration of each stage is T/n. When the team of a stage *i* completes the tasks for that stage for a time box *k*, it then passes the output of the time box to the team executing the stage *i+1*, and then starts executing its stage for the next time box *k+1*. Using the output given by the team for S_i, the team for S_{i+1} starts its activity for this time box. By the time the first time box is nearing completion, there are *n-1* different time boxes in different stages of execution. And though the first output comes after time T, each subsequent delivery happens after T/n time interval, delivering software that has been developed in time T.

As an example, consider a time box consisting of three stages: requirement specification, build, and deployment. The requirement stage is executed by its team of analysts and ends with a prioritized list of requirements to be built in this iteration. The requirements document is the main input for the build team, which 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 2.

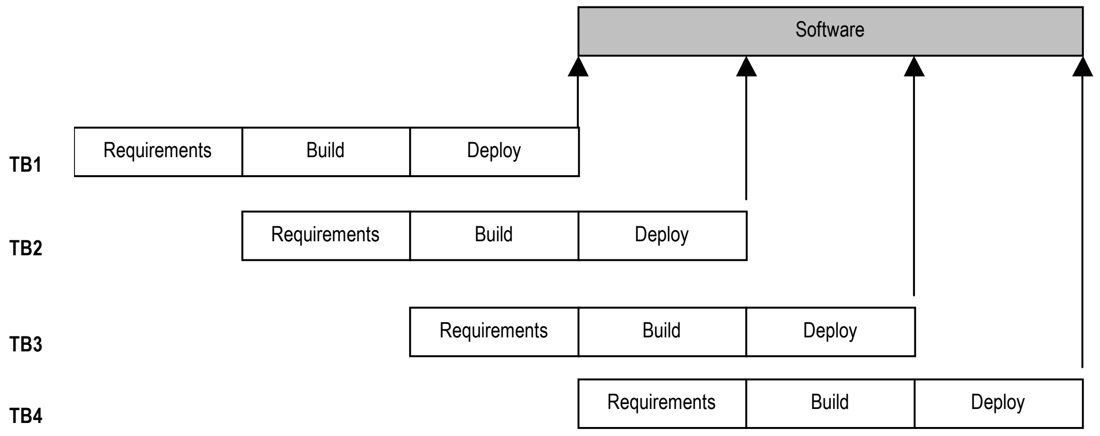

Figure 2: Executing the timeboxing process model

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.

2.3 Time, Effort and Team Size

It should be clear that though the duration of each iteration has not been reduced, the delivery time to the end client (after the first iteration) reduces by a factor of n with a n-stage time box. That is, the speedup (the reduction in the average completing time of an iteration) is n times. We can also view it in terms of throughput – the amount of output per unit time. We can clearly see that in steady state, the throughput of a project using timeboxing is n times that of if serial iterations were employed. In other words, n times more functionality is being delivered per unit time.

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. 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.

If the same effort is spent in each iteration, what is the cost of reducing the delivery time? The real cost of this increased throughput is in the resources used in this model. The total team size for the project, in which multiple time boxes may be running in parallel, is

$$\text{Project Team Size} = \sum_{i=1}^{n} R_i.$$

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.

With timeboxing, there are different teams for different stages. 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.

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 more manpower. However, through the timeboxing model, we have been able to use more manpower in a manner such that by parallel execution of different stages we are able to deliver software quicker.

2.4 Unequal Stages and Exceptions

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. What happens when such exceptions present themselves?

First situation which is likely to occur is that the stages are of unequal duration. As the pipelining concepts from hardware tell us [12], 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. Note that even with this, a considerable speedup is possible. For example, let us consider a 3-stage pipeline of the type discussed above in which the different stages are 2 weeks, 4 weeks, and 3 weeks – that is, the duration of the time box is 9 weeks. In a serial iterative development, software will be delivered every 9 weeks. With timeboxing, the slowest stage will determine the speed of execution, and hence the deliveries will be done every 4 weeks. This delivery time is 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 the example given above, it means that the first and third 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 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 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.)

Another special situation can easily arise – an exceptional condition arises during the execution of a stage of some time box, due to which the stage is not able to finish in its allotted time. We do not need to worry about the nature of the exception – except that the net effect of the exception is that it elongates that stage by ΔT. Clearly, if such an exception occurs, the execution of the later stages will be delayed resulting in the output being delivered late by ΔT. Similarly, due to this delay, the output of earlier stages in later time boxes cannot be consumed in time, resulting in the teams of these stages "waiting" for their output to be consumed. The net result of this is that, one delivery gets delayed by ΔT, with a corresponding slack time for each team for one time box. After that, all future deliveries will come after every T/n time units (for a n-stage time box of T duration.)

3 APPLYING THE MODEL

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

3.1 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 [13]. With a good estimation model, which set of features can be built in an iteration can be decided. (Technology churn and unstable architecture make it harder to do feature-based estimation as technology and architecture changes impact effort requirement.)

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.

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.

3.2 Handling Changes

We know that requirements change and that such changes can be quite disruptive [5, 14]. Most development processes add a change management process for accepting a change request, analyzing it, and then implementing it, if approved (an example of such a process can be found in [13].) 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.

3.3 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.

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.

3.4 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 refactoring, during which the design of the system is simplified and redundancies removed [4]. 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.

In the timeboxing model, 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 refactor 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. 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.)

3.5 Project Management

Managing 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 [13].) In a timeboxing project, the frequency of milestones is more and hence a 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.

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.

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 [4], 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.

The project will require tight configuration management as many teams are working concurrently. The reconciliation procedures (i.e. procedures that are used to reconcile two changes if they are done concurrently [13]) need to be solid and applied regularly as it is likely that changes will be made by the team for a stage to the output produced by the previous team. And when this is done, as the previous team is already working on the next iteration, there will be a need for reconciliation. 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.)

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.

As discussed above, to keep the project manageable, the number of stages should be few. We expect the stages to be two to four. 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 [2, 7]. 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.

4 AN EXAMPLE

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) of 6 week duration. 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 – 2 weeks for requirements, 3 weeks for build, and 1 week for deployment. The size of the teams 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.)

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 3. 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.)

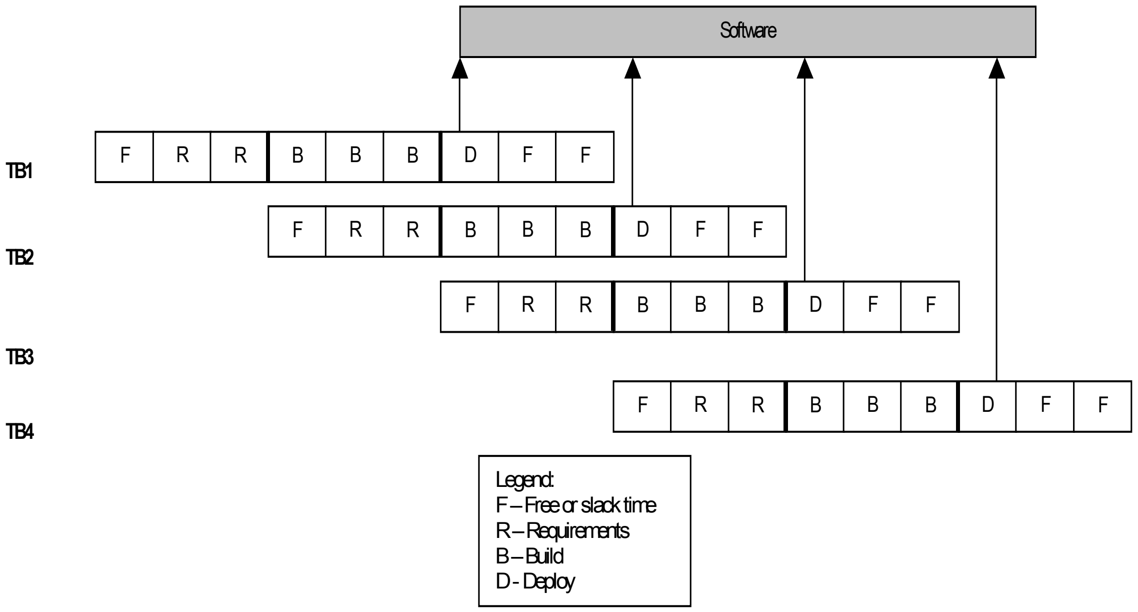

Figure 3: Time boxed execution of the project

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 4. As is shown, after the initial time boxes, there is no slack time.

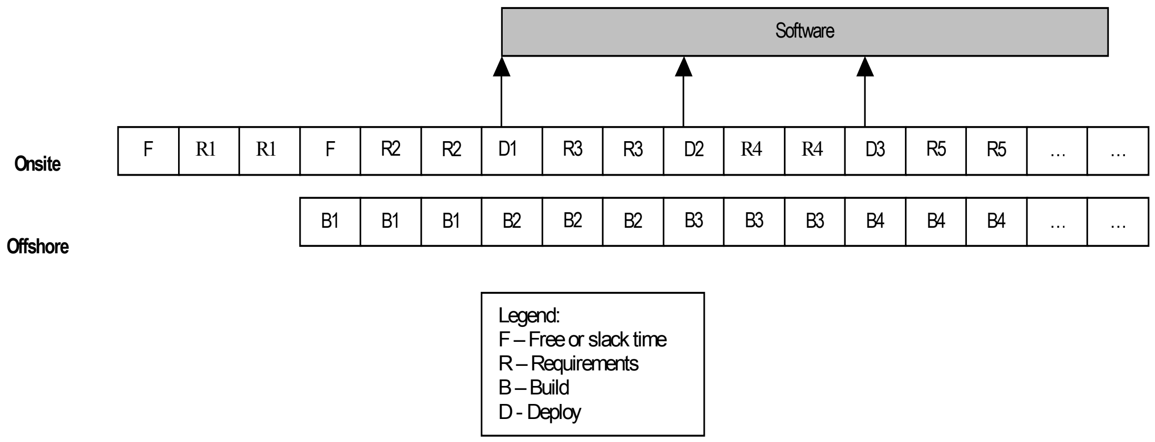

Figure 4: Tasks of the on-site and offshore teams

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 2/3rd of their time for requirements and 1/3rd 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.

Some other interesting features of implementing the timeboxing model on this project are:

- Though the features to be built in a time box should be decided on priority, the learning curve that is needed by any team to take over an existing system was kept in mind. Consequently, for the first time box, a few lightweight features were chosen – they enabled both the teams to become familiar with the business domain and the existing system.
- 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 [3, 9, 11, 13]. However, unlike a postmortem that is supposed to benefit the next projects, the focus of this analysis was to learn from one time box to improve the execution of the future time boxes.

- 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 CONCLUSIONS

Iterative software development is now a necessity, given the velocity of business and the need for more effective ways to manage risks. One 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. Each time box is divided into a sequence of fixed duration stages, with a dedicated team for each stage. As a team completes its task for a time box, it passes the outputs to the team for the next stage, and starts working on its task for the next time box. Due to pipelining, the turnaround time for each release is reduced substantially, without increasing the effort requirement.

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 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.

So far the experience in using the model is very positive. To make the process applicable for a wider range of projects, we are currently developing tailoring guidelines. The impact of one time box on another, and the learning and other feedback from a time box to later time boxes and its impact on quality and productivity are some other areas that we are currently studying.

REFERENCES

1. V. R. Basili and A. Turner, Iterative enhancement, a practical technique for software development, *IEEE Transactions on Software Engg.,* 1(4), Dec 1975.
2. V. R. Basili, Ed., *Tutorial on Models and Metrics for Software Management and Engineering,* IEEE Press, 1980.
3. V. R. Basili and H. D. Rombach, The experience factory*, The Encyclopedia of Software Engineering*, John-Wiley and Sons, 1994.
4. K. Beck, *Extreme Programming Explained*, Addison Wesley, 2000.
5. B. W. Boehm. Improving software productivity, *IEEE Computer,* Sept. 1987, 43-57.
6. B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61--72, May 1988.

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

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

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

10. A. Cockburn, *Agile Software Development*, Addison Wesley, 2001.

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

12. J. L. Hennessy and D. A. Patterson, *Computer Organization and Design,* Second Edition, Morgan Kaufmann Publishers, Inc., 1998.

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

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

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

16. 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.

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