Time as Factor in Software Project Development: Necessary Evil
Manas Kumar Yogi¹ Lakshmi Lavanya Boppana²
¹Assistant Professor ²B.Tech Student
¹,²Department of Computer Science & Engineering
Pragati Engineering College Surampalem, Andhra Pradesh, India

Abstract—Time is addressed differently by different people and cultures; for example, in western culture, time is mainly associated with financial profit, i.e., “Time is money.” Time plays a special role in software engineering: the project schedule should be met, the product should be delivered on time, and teammates should complete their tasks on time, and so on. This paper deals with how the time concept is expressed in software engineering in general and in conventional software development in particular. In conventional software development time is boxed for each activity, and when needed, instead of “moving” deadlines, the scope is changed according to customer priorities. This conception is supported by conventional software development methods in different ways that not only enable one to work at a sustainable pace, but that also result in high quality products.

Key words: Development Time, Quality, Estimation

I. INTRODUCTION

This paper discusses how time issues are expressed in conventional software development environments, more specifically, it describes the planning activity that is part of the iteration Business Days.

In a specific iteration planning, the customer presents the relevant stories, and the team, based on a high level design prepared previously, carries out a planning session which includes work distribution, time estimation, and load balance among the team members. The goal of the planning session is to reach a complete yet sustainable plan for the implementation of the customer stories during the coming iteration. At the end of the planning session all teammates are familiar with the customer’s vision and priorities and with the individuals’ tasks for the coming iteration. No change is introduced on the iteration priorities and with the individuals’ tasks for the coming iteration. At the end of the planning session all teammates are familiar with the customer’s vision and priorities and with the individuals’ tasks for the coming iteration. No change is introduced on the iteration.

II. CONSIDERATIONS WHILE DEVELOPING A SOFTWARE

1) Time estimation of development tasks, work distribution among teammates, total time calculation for the iteration, and load balance among teammates.
2) Awareness regarding time and pace notions with respect to conventional software development.
3) Familiarity with how a week in the life of an conventional team works.

A. Below Study Questions Must Be Answered While Moving Forward In Software Development:

1) What is unique about the planning process of a software project?

2) What are the main characteristics of the conventional planning sessions? Of the release? Of the iteration?
3) Describe methods for time estimation of software projects.
4) How do different software development methods relate to time management?
5) What measures can be used for time management in software projects?
6) What is the concept of team velocity? What is its importance? How do conventional methods calculate team velocity?
7) What is the concept of sustainable pace? What is its importance?
8) Why, in your opinion, are many software projects characterized by schedule overrun? How would you suggest overcoming this problem?
9) Do you have personal successful experiences with respect to project schedules? If yes, explain the source of their success.
10) What are the similarities and differences between software projects and non-software projects? How do these similarities and differences relate to time issues?
11) One of the famous rules of software engineering is Brooks’s assertion that “Adding manpower to a late software project makes it later” Can you explain this rule? In your opinion, is it relevant for other professions as well? Why? In what ways does it highlight the importance of the time dimension of software projects?

III. TIME-RELATED PROBLEMS IN SOFTWARE PROJECTS

In his classic paper The Mythical Man-Month, Brooks writes: More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?

First, our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption which is quite untrue, i.e., that all will go well.

Second, our estimating techniques fallaciously confuse effort with progress, hiding the assumption that man and months are interchangeable.

Third, because we are uncertain of our estimates, software managers often lack the precision plan.

Fourth, schedule progress is poorly monitored. Techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering.

Fifth, when schedule slippage is recognized, the natural response is to add manpower.

A. Tasks To Be Done To Face Time Related Problems:

1) Explain each of Brooks’s claims.
2) Do you have personal experiences that strengthen Brooks’s claims?

It is clear that time occupies a crucial place in project management. It seems, however, that in software engineering time plays a special role. It also seems that time is one of the most important factors dominating software development. One reason that makes time so crucial in software development is that software development does not progress linearly. This, in fact, is expressed by Brooks’s statement that, in software projects, months and people are not interchangeable.

B. List of Time-Related Problems of Software Projects:

1) Bottlenecks:

Bottlenecks in software development occur when one or more functions in the process await the output of another function in the process, with teammates having nothing to work on in the meantime. This may happen, for example, when quality assurance people wait for artifacts to work on or, vice versa, when developers wait for artifacts from quality assurance people. Another example is when developers wait for artifacts from system analysts, like the specification of a specific module.

2) Project Planning and Schedule:

Two main problems are associated with schedules, which are, in fact, closely connected. The first is the mere construction of a feasible project schedule. The second problem is to meet the schedule that has been set.

3) Time Estimation:

There are different ways to support time estimation. With respect to the estimation of the development time of a specific module/class/task, it is well known that the greater the module/class/task is, the more difficult it is to estimate its development time. Present evidences report that the smaller the estimated unit is, the more accurate is its time estimation.

4) Time Pressure:

Time pressure is the result of the previous problems. It happens usually toward the end of the development process, when teammates cannot meet the project schedule, either because of poor time estimations or bottlenecks. Time pressure usually leads to the skipping of different testing activities, which in turn leads to a decrease in software quality. For instance, it is said: “The testing activity often does not get the attention it deserves. By the time the software has been written, we are often pressed for time, which does not encourage thorough testing. Postponing test activities for too long is one of the most severe mistakes often made in software development projects. This postponement makes testing a rather costly affair.”

5) Late Delivery:

Late deliveries occur as a result of inappropriate project planning, usually due to poor estimations. Data indicate that the percentage of software projects that fail to accomplish on-time delivery is quite high.

IV. SOFTWARE ORGANIZATIONAL SURVEY FROM THE TIME PERSPECTIVE

This study illustrates some of the above mentioned problems that characterize software projects. The following data were gathered in a large company during an organizational survey conducted at the company in order to understand the roots of the problems the company encountered with respect to software development and to propose possible solutions to overcome those problems. We present here only data related to time.

A. Tasks to Be Performed In the Survey:

For each of the following data pieces, explain its source and suggest how conventional software development attempts to solve it.

1) One of the Questions Presented to Developers was:

Indicate two factors that you would improve in your team in order to develop higher quality software.

Out of 46 answers, 13 developers (28%) indicated the time element as a factor that should be addressed in order to improve software quality. Here are several illustrations:

- Planning a schedule that can be met.
- Allocation of time to learn the things [to be implemented] before we rush to the next coding; allocation of enough time for review and debugging.
- Allocation of time for design and education in design topics.
- Commitment of the software people to the schedule and the product.
- Add time to the development and testing [stages].
- Dedicate additional time to code review.

2) The developers were asked to describe the development process in the organization and to indicate the pros and cons of that process. Following are some of the time-related responses received.

Indicate at least three benefits of the development process that you have just described.

Out of 22 developers who answered this question, none mentioned the concept of time.

At the same time, time appears in the “pitfalls” question, as described below.

Indicate at least three pit falls of the development process that you have just described.

Out of 22 developers who answered this question, 9 (41%) mentioned the concept of time, as is illustrated by the following quotes:

- It is difficult to estimate times (the scope is not really known).
- This process requires a lot of time and resources.
- It lasts too long.
- The schedule is tight and it requires making decisions that are detrimental to the quality of the development.
- Too many bad ideas are accepted because there is not enough time to establish a new process.
- The lack of parallel work leads to some inefficiency and redundancy in the development time; the requirements are not always well defined, a fact that takes a lot of the time of the software engineers and the system engineers during the design review.
- The tests require too much time.
V. TIGHTNESS OF SOFTWARE DEVELOPMENT METHODS
This section shows that conventional software development methods are tight, which means that, though they are flexible in terms of change introduction, they are very sensitive to time frameworks. This awareness of time frameworks is important since software is an intangible product, and accordingly its development should be kept strictly paced; otherwise, problems such as the ones described above emerge in many cases. Also, “tightness” indicates the idea that the tighter a development method is, the more ordered is the software development environment it inspires.

The tightness of conventional methods is achieved in several ways. First, short iterations and releases are set. Second, conventional methods set the time dimension fixed as well as the cost and quality axes, enabling changes only in the project scope. The time setting is determined by the project timetable. Customers and Users, by designing the project roadmap consisting of iterations of a fixed number of development days.

Conventional projects perform time boxing at the release and the iteration levels, so a single conventional project contains multiple time-boxes. Time boxes force the customer to make decisions on the short-term direction of the project; second, they always provide a near-term goal, which can keep the entire team from wandering off target. Finally, time boxes ensure that the team delivers something useful within a short and defined period. The last two arguments closely refer to the first-things-first idea if you have a four months release time you think you have a lot of time and you are not focused on what is important to deliver working software to the customer.

The three variables, time, quality, and scope as variables of which, in some software development processes, two are usually set and the third can vary. This method doesn’t work well in practice. Time and cost are generally set outside the project. That leaves quality as the only variable you can manipulate. Lowering the quality of your work doesn’t eliminate work; it just shifts it later so delays are not clearly your responsibility. You can create the illusion of progress this way, but you pay in reduced satisfaction and damaged relationships. Satisfaction comes from doing quality work. Among other differences, software development methods vary with respect to their tightness level and the culture that the tightness level of the development method inspires.

We should adopt a relatively tighter approach, since we believe that the tightness level that characterizes a development method is one of the main factors that enable software projects to achieve their targets successfully.

We define the tightness level of a software development method along the following five dimensions, whereby the higher the values of these factors, the tighter the software development method.

1) Project plan dimension: number of releases and feedback milestones, level of emphasis placed on planning, number of days for which specific planning is made (the smaller the number of days, the tighter the software development method).

2) Procedures and standards dimension: level of detail that describes the software development method.

3) Responsibility and accountability dimension: level of role performance, level of personal accountability, frequency at which team members are required to report on their progress.

4) Time estimation dimension: importance given to time estimation, resolution level of time estimation (hours, days, months) the smaller the time units, the higher the resolution, and the higher the value of this dimension.

5) Individual need satisfaction dimension: mutual dependency of team members, level of planning that inspires the message “Invest now for the future.”

Tasks to measure the tightness level:
1) Analyze the conventional practices according to their tightness level.
2) For each dimension of the tightness model explain how it is expressed by the conventional approach.
3) How does the tightness model inspire software project management?
4) In your opinion, what tightness level (low, high) of a software development method do teammates prefer? Why?
5) Choose two conventional development methods and analyze them according to the tightness model.
6) Suppose we have a tool that measures the tightness level of a team in a similar way to the way in which we measure the tightness level of a development method. Suggest possible scenarios for the adoption of conventional development when the team tightness and the software development tightness fit each other and when there is a clash between the tightness level of the team and that of the software development method. Address questions such as: What would happen when the team tries to adopt the development method? What is the best situation for the application of a conventional process?

VI. SUSTAINABLE PACE
Tightness is one time-related characteristic of conventional software development. Sustainable pace is another one. This section introduces the importance of the sustainable pace that conventional methods inspire. Sustainable pace means that the development process is carried out in a reasonable number of hours, which are well planned and enable the team to be productive and to produce quality products.

The rationale for keeping a sustainable pace is that overworked programmers are unable to produce quality code. Since several conventional principles, such as the whole team, pair programming, the planning session, and time estimation, ensure productivity during the development hours, conventional programmers can work at a sustainable pace, be productive, and produce quality code.

There is good evidence that productivity and the quality of output goes down when more than about 40 hours a week are worked. It is sometimes necessary to put in extra effort to overcome some temporary obstacle or to deal with an emergency, but if over-time working becomes a way of life then there will be longer-term problems”. Indeed, in some cases this principle is presented as a general guideline or as a recommendation. At the same time, however, in
conventional development environments this concept is one of the core principles of the approach; and for a conventional team, working at a sustainable pace is an integral part of the development framework.

A. Time Measurements:

One of the common measures with respect to software project time management is the time estimated for development tasks versus the actual time to develop them. In order to control the iteration progress, this kind of measure is inspected on a daily basis; in order to control the release progress, this kind of measure is also examined each iteration. The daily progress is measured against the iteration commitment conducted in the Business Day. The iteration progress is measured against the release commitment, shared at the beginning of the release.

The whiteboards of the collaborative workspace constantly present a graph in which the horizontal axis represents the iteration days and the vertical axis indicates the number of hours. Each day, the tracker adds two new points to the graph that represent the project’s progress. The first one—the “total expected” point—represents the cumulative estimations of all tasks that were completed up to the previous day; the second point—the “total done” point—represents the cumulative actual time devoted to those tasks. A completed task is counted only when the developer in charge completes its coding, unit testing, and integration into the developed system.

VII. MEASURING ESTIMATIONS VERSUS ACTUAL DEVELOPMENT TIME

Figure 7.1 presents three graphs: estimations, actual time in a specific iteration of a specific team, and the expected average pace according to available time.

As can be observed, there is a significant difference between the allocated time for development (about 270 hours) and the time that was actually dedicated for development. Several factors may cause this gap: First, only completed tasks are presented; tasks that have been performed but have not been completed before the end of the iteration are not calculated, and the time that has been dedicated for their development is not reflected in the graph. Second, time invested in tasks that appear urgently, like support service to end users who work with deployed modules, is not presented in the graph. Third, there was a sudden absence of developers whose time was taken into account in the iteration.

Figure 7.2 shows another graph of estimations versus actual development time. In this case, all the planned tasks were completed. Accordingly, the total-expected point unites with the expected-pace point, since this was the number of development hours considered in the planning session.

It can also be observed in Figure 7.2 that the time distribution among tasks was reasonably good. Further, the graph shape shows that the integration is continuous; too many dependencies between tasks would have delayed their completion and integration, reflected by a nearly flat total-done line until almost the end of the iteration, and then a sharp increase as many tasks are completed together.

Data such as those presented in Figures 7.1 and 7.2 is examined every day. Thus, if it seems that problems are expected with respect to the completion of the planned stories of the current iteration, the customer can reprioritize the stories and change the iteration scope. This way, stories that remain in the iteration scope can be developed and tested properly.

In addition, these data are examined when the iteration ends in order to learn about the team velocity. Team velocity can be perceived as the amount of productive work units per iteration. Measuring team velocity increases the visibility of the development progress and enables one to make decisions on how to continue with respect to functionalities and priorities. The data are examined also with all the other iterations completed so far in the release, in order to learn about the project’s progress.

A. Importance of Above Measures:

1) Prioritizing Development Tasks:

What is common to refactoring, test-driven development, and the planning activity, which are some of the basic practices of conventional teams? Why are they important? After all, it can be argued that the only important activity is code production. The answer is that they all support the management of the software development process. This idea is further strengthened by the following case study that uses Covey’s concept of First Things First. It shows how
conventional software development helps conventional teams focus on what is important rather than on what is urgent. This idea is manifested by different conventional practices such as refactoring and pair programming which helps the team stay focused and avoid the distractions of unimportant and non-urgent activities.

2) First Things First:
This case study illustrates how time management is manifested in conventional software development environments and how conventional software development increases the team members’ awareness of time management ideas as they are manifested in conventional software development environments. This data set is taken from a team undergoing the transition to conventional software development. The team is used to carrying out a one-hour reflection at the end of each iteration. The reflections that the team conducted at the end of the first two iterations were dedicated to learning processes (the first) and to prioritizing activities (the second).

The rationale for the second reflection is based on the realization that time management is a key element of conventional software development. Accordingly, its objective was first, to emphasize the importance of time management issues in software projects, and second, to help developers grasp how time management is expressed in conventional software development.

More specifically, according to Covey’s concept of First Things First, conventional software development helps teams focus on what is important rather than on what is urgent. As mentioned above, this idea is manifested by different conventional practices such as refactoring and pair programming.

Table 7.2 Reflection on time management:
We maintain a table as shown below to reflect on a role and personal work habits/processes in the project and to complete the following matrix accordingly:

- Urgent and Important
  - Not Urgent but Important
  - Urgent but Not Important
  - Not Urgent and Not Important

Table 7.2 presents a sample of suggestions developers made for each quadrant. As can be seen below, Quadrant II—the quality quadrant—contains conventional activities and practices.

During the discussion that followed the individual work, guidelines for items in the Important but Not Urgent quadrant, such as the following ones, were suggested:

- When there is a problem, “put it on the table” and talk about it (thus, we make it important).
- Allocate time for important but non-urgent issues.
- When the time is dedicated and we sit [to discuss the problem], do not waste time.
- Let as many people as possible obtain a wide perspective.
- Plan everything possible in advance, not to put things off till the last minute.
- As much as possible, do not perform tasks that are not connected to the current iteration.

The project manager can note: “The second quadrant is characterized by teamwork—because of the team, I do what is important and I do not give up.”

The above quotes illustrate that conventional software development helps software developers focus on the second quadrant, inspiring a development process that is composed of important (but not urgent) activities and eliminating the performance of urgent activities (mainly the not important ones) during the course of the project.

B. Summary and Reflective Questions:
1) Select one of your development requirements, and perform the following activities with respect to it:
- Prepare a high level design;
- Break it down into development tasks;
- Estimate the development time for each task, including development and unit testing;
- Calculate the total time—that is, sum up the estimated development time for all the tasks;
- Develop all tasks and measure the actual time invested;
- Sketch a graph of estimations versus actual development times;
- Discuss lessons learned

2) Analyze the concept of time as it is manifested in conventional software development from the HOT—Human, Organizational, and Technological—perspectives.

VIII. CONCLUSION
This paper discusses the concepts of time and time management and presents the planning activities carried out in conventional software development environments. The way the conventional approach refers to time is “tight”—the main activities carried out in conventional development environments are time boxed and are measured with respect to their actual development time in hour resolution. These characteristics ensure a controlled development process that enables the developers to increase the product quality. The concept of sustainable pace is also presented and illustrated in this paper.

REFERENCES
conference on extreme programming and agile processes in software engineering.


