UNIT 4 RISK MANAGEMENT CONCEPTS

Structure **Page Nos.**

4.0 INTRODUCTION

Risk Analysis and control is a crucial factor for any software project and it is a topic of management. Several standard techniques for identifying project risk, assessing their impact, monitoring and controlling them are discussed in this unit. This unit is dedicated to Risk Management Concepts. It discusses typical risks in the software development process, technical planning, project tracking, delivery timings and particle recovery.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- describe Risk Management Concepts and strategies to manage such risks;
- what is Risk Monitoring, Technology Risk, Risk Components and Drivers, Customer Related Risks;
- knowledge of Basic Concepts of Project Scheduling and Tracking, and
- knowledge of Software Testing Fundamentals.

4.2 INTRODUCTION TO RISK MANAGEMENT

Experimental assessment of different organisational structures is difficult. It is clearly impractical to run large software development projects using two different types of organisation, just for the purpose of comparing the effectiveness of the two structures. While cost estimation models can be assessed on the basis of how well they predict actual software costs, an organisational structure cannot be assessed so easily, because one cannot compare the results achieved with those one would have achieved with a different organisation.

Experiments have been made to measure the effects of team size and task complexity on the effectiveness of development teams. In the choice of team organisation, however, it appears that we must be content with the following general considerations:

Just as no-life cycle model is appropriate for all projects, no team organisation is appropriate for all tasks.

Decentralised control is best when communication among engineers is necessary for achieving a solution.

Centralised control is best when speed of development is the most important goal and the problem is well understood.

An appropriate organisation tries to limit the amount of communication to what is necessary for achieving project goals, no more and no less.

An appropriate organisation may have to take into account goals other than speed of development. Among these other important goals are: lower life-cycle costs, reduced personnel turnover, repeatability of the process, development of team members at junior level into senior members, and widespread dissemination of specialised knowledge and expertise among personnel.

4.2.1 Managing Risks

A software engineering project is expected to produce a reliable product, within a limited time, using limited resources. Any project, however, runs the risk of not producing the desired product, overspending its allotted resource budget, or overrunning its allotted time. Risk accompanies any human activity.

Risk analysis and control is a topic of management. Several standard techniques exist for identifying project risks, assessing their impact, monitoring and controlling them. Knowledge of these techniques allows a project manager to apply them when necessary to increase the chances of success of a project.

We have already seen in previous sections, many examples of software development problems that can be viewed from a risk analysis point of view. For example, we have discussed the difficulties of specifying project requirements completely. Given these difficulties, a project runs the risk of producing the wrong product or having the requirements change during development. An effective approach for reducing this risk is prototyping or incremental delivery. A different type of approach to handling the risk of late changes in the requirements is to produce a modular design so that such changes can be accommodated by actual changes to the software. Of these approaches, prototyping tries to minimize changes in the requirements, while modular design tries to minimize the impact of changes in the requirements. Choosing between the two alternatives, or deciding to use both, should involve a conscious and systematic analysis of the possible risks, their likelihood, and their impact.

At different levels of an organisation, different levels of risks can be tolerated. For example, a project manager may not want to tolerate any delay in the schedule, while another member at the level of team leader might be more concerned with the reliability of the product. A project manager might not be able to tolerate the risk of running overbudget by more than 10% while a higher level manager who is more aware of the value of early entry into market may be more concerned with the time taken by the project and might be willing to overspend the budget by much more if the product can be produced sooner.

4.2.2 Typical Management Risks in Software Engineering

By examining the difficulties that arise in software engineering, we can identify typical areas of risk that a software engineering project manager must address. One risk is with the change requests in the midst of the project. Another important risk is

not having the right people working on the project. Since there is a great variability among the abilities of software engineers, it makes as big difference whether a project is staffed with capable or mediocre engineers. If key positions in a project are staffed with inappropriate people, the project runs the risk of delaying deliveries or producing poor quality products, or both.

Risk	Risk Management Techniques
a) Individual shortfalls	Staffing with top talent; job matching; teambuilding; key-personnel agreements; cross-training; pre-scheduling key people.
b) Unrealistic schedules and budgets	Detailed cost & schedule estimation; cost of design, incremental development, software reuse, requirements scrubbing.
c) Developing the wrong software functions	Organisation analysis; mission analysis; concept formulation; user surveys; prototyping early user manuals.
d) Developing the wrong user interface	Prototyping; scenarios; tasks analysis; user characterization.
e) Continuing stream of change requests	High change threshold; information hiding; incremental development (defer changes to later increments)
f) Shortfalls in externally furnished components	Benchmarking; inspections; reference checking; compatibility analysis.
g) Shortfalls in externally performed task	Reference checking; per-ward audits; award-fee contract; competitive design or prototyping; teambuilding
h) Real-time performance shortfalls	Simulation; benchmarking modeling; tuning.

Schedule overrun risks can be reduced by limiting dependencies among tasks. For example, if many tasks cannot be started until a given task is completed, delay in that one task can delay the entire project. If the software development is scheduled to start after the hardware acquisition is completed, then, any delay in completion of the hardware acquisition translates directly into a delay in the entire project. A way to control this schedule risk is to produce a simulation version so that software development can be carried on even if the hardware acquisition is delayed.

PERT charts can help a manager identify schedule bottlenecks immediately, even mechanically. A node with many outgoing arcs is a sign of trouble, and the manager should try to reschedule activities to avoid it. Such a node should be rescheduled especially if it happens to be on the critical path. One way to reschedule the activity is to break it up into smaller activities. Examining the risk items in column I of the table, we can see that they overlap the items that are used in software cost estimation models. If a factor has a high multiplier in cost estimation, it represents a risk that must be managed carefully.

While we only talked about management in the large, that is, management of a group of system engineers who must cooperate to produce a common product, many techniques can be used by an individual software engineer as well, that is, in the small. Indeed, each software engineer must carefully plan, monitor, and execute the plan for his/her own assignment. Staffing and directing are the only two management functions that PERT chart can help individual engineers on non-trivial activities.

While we have discussed the difficulties in measuring software productivity, we have also stated the importance of defining and collecting such metrics. The metrics of existing projects should be studied so that appropriate management principles can be applied to guide through the phases of future projects. In the absence of metrics, there is no way to measure whether progress is being made and, if so, at what rate.

In addition to the technical aspects of management that we have discussed, a manager is also involved in resolving conflicts among competing goals. For example, in assigning tasks to people, should the experienced engineer be assigned to do all the difficult jobs and get them done fast, or should s/he work with less experienced engineers so that they become enough experienced for future projects?

Large software systems exhibit what has been called as progressive and anti-regressive components in their evolution. A software evolves progressively when features are being added and functionality is increasing. But, after adding to the software for a long time, its structure becomes so difficult to deal with that an effort must be undertaken to restructure it to make it possible to make further additions later. Reengineering, which does not add any functionality, is an anti-regressive component of software because its goal is to stop the software from regressing beyond hope. The decision that the manager must make is when it is time to undertake anti-regressive activities. In its logical extreme, this decision amounts to whether a software system must be retired and a new one developed.

A common conflict is known as the mythical man-month conflict. In some disciplines, people and time are interchangeable, that is, the same task can be accomplished by two people in half the time that it takes a single person. In software engineering, as we have seen adding more people increases the overhead of communication on each engineer, preventing a liner increase in productivity with additional people. In fact, after a certain point in the project, and beyond a certain number of people, adding more people to the project can delay the project rather than speed it up. The difficult task of the manager is to determine when those limits have been reached. The cost estimation models that we have discussed are the beginning of the foundation for allowing such as decision to be made quantitatively.

Will the approach that worked for one project work for another project? One of the painfully observed phenomena in software engineering is that many techniques do not scale up; that is, a method that works on smaller projects does not necessarily work on large projects. It is not, in general possible to derive precise scheduling information from a throwaway prototype. For example, if the prototype demonstrates a tenth of the functionality of the final product, the product will not take ten times the development time that the prototypes took.

Should engineers be encouraged to reuse existing software in order to reduce development time? While software reuse reduces coding time, it may cause difficulties in other phases of the life-cycle. If the modules that are reused do not supply the exact interfaces suitable to the design and functionality of the product, then they cause the engineer to go through extra effort just to match the interface, and worse, they lessen the evolvability of the product. These problems point out the immaturity of software reuse technology, rather than an inherent flaw in the idea of software reuse. However, things are fast changing and this may not be the case

always. Whatever the reasons, however, the manager is left with a difficult set of compromises to consider.

Finally, we must recognize that there are no panaceas to software engineering problems. For example, using the latest process an incremental, prototype-oriented, life-cycle model or the latest technology an advanced tool for configuration management or the latest methodology object oriented analysis and design will not solve all software development problems. In truth, software engineering is a difficult intellectual activity, and there are no easy solutions to difficult problems. Using the right process, the right technology, the right methodology, and the right tool will certainly help to control the complexities of software engineering, but it will not eliminate them altogether. In practice, because software engineering is such a difficult task at times, and because costs are rising rapidly, managers tend to grasp at any solution that comes along which promises to solve their real problems.

Panaceas do not exist, however, a manager is best advised to accept the difficulties of the job and carefully evaluate the impact of any newly offered proposed panaceas.

4.2.3 Technical Planning

At the start of the project, it would have been wise to develop documents stating the requirements for the new product. These comments should have been based on a careful and organised interaction with potential users, paying close attention to choosing a representative sample of the user population.

One might argue that it would have been difficult, or even impossible, to write such documents, since the desirable features of the system were not clear in the first place. A possible solution to these problem would have been to put a limited effort into the development of a system that would act as a prototype. The prototype system would then help assess the most critical issues and derive firm requirements by observing users' reactions when using the system. Unfortunately, the designers did not even realise that a problem existed and, therefore, they did not even consciously choose between the first alternative–specifying requirements carefully–and the second–developing fast a exploratory prototype.

Similarly, careful planning of resource allocation should have started, both from the point of view of work assignment to designers and programmers and from the point of view of physical resources management (e.g., hardware acquisition and office space). But, instead, just the opposite happened. An amusing but dangerous "Game" started between the designers and the very few representative clients. In this case, of course, the designers included the company leaders, since they were technical people and they had initiated the idea of the product in the first place.

In fact, everybody was excited with the innovative and challenging features of the product, but nobody paid much attention to fairly obvious but critical details. For instance, a programming language was designed to allow the sophisticated user to define his or her own document composition rules. Very sophisticated and expensive word-processing facilities were included without measuring their cost effectiveness.

For a long time, nobody paid attention to the definition of suitable user interfaces to facilitate the interaction of non-technical people – a lawyer or secretary – with the system. Similarly, sophisticated features for the automatic computation of invoices on the basis of input data (time, service value, travel expenses, etc.) were designed, but no attention was paid to standard office operations such as the filing of large numbers of documents (e.g., records of automobile sales in some offices, however, several hundreds of such documents are produced every day).

From a technical point of view, many typical mistakes were made:

No analysis was performed to determine whether all the product features were needed by all users, or whether it would be better to restructure the functionality of the product based on different classes of users. More generally, no effort was put into determining which qualities of the product were critical for its success. For instance, in the choice of the hardware and of the development software (the operating system, programming language, etc.), little or no attention was paid to their evolution, and no effort was made to prepare for possible changes to them.

No "Design for change" was done, i.e., no design decision was influenced by any analysis of which parts of the product were likely to change during the product's lifetime. Strong pressure was applied to have some (any) code running as early as possible. No precautions were taken to minimise damages due to personnel turnover.

What is perhaps worth pointing out is that everybody in the company was, of course, aware of such classical mistakes in software engineering. This awareness notwithstanding, the mistakes were made. These remarks show that knowing the difficulties is not enough. It is also necessary to have the technical and organisational ability and willingness to face them, even at the cost of doing something that does not appear immediately attractive and productive.

4.2.4 Project Tracking

After a while (about six months after the start of development) some mistakes became apparent both from a technical and from management perspective. For instance, the lack of a clear definition of the product's functionality caused some initial misunderstanding between the potential users–the ones with whom the early contacts has been established–and the designers. It was realised that some features that had been neglected at the beginning were actually quite important.

Also, getting in touch with other potential users showed that not all of them needed the same features. Thus, a modular architecture would have been preferable, even from the user point of view, allowing the product to be customised for different classes of users. Finally, it became apparent that the original cost estimates were off by an order of magnitude. This invalidated the initial economic and financial plans.

The reaction to this situation was even worse than the problem itself: the impact of the mistakes – both technical and non-technical – was again underestimated. In general, the attitude was of the following type, "OK, we made a few mistakes, but, now, we are almost done. So let's put in a little more effort, and we will complete the product soon". That is, no critical and careful analysis of the mistakes was made, nor was a serious re-planning and redesign of the whole project attempted.

Under the pressure of "being almost done and close to delivering the product," the design focused more and more on the very end product i.e., machine code. Classical "patches" on object code were made wildly, no systematic error and correction logs were kept, and communications among designers occurred almost exclusively orally in an attempt to save time.

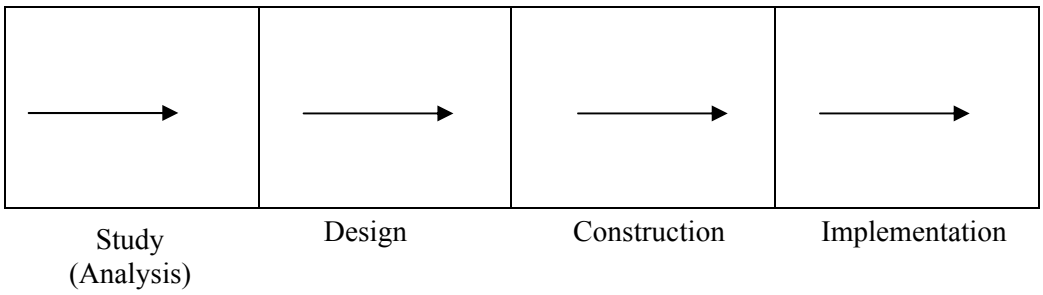

Figure 4.1: Ideal Project Progress

Figure 4.2 : Usual project progress

4.2.5 Project Delivery

It was decided that income could be generated as soon as the company started delivering the first versions of the product or as soon as new contract is signed. In turn, the customers would be good references for the product.

This decision, too, turned out to be a big mistake. In fact, a new dimension was added to the already critical technical and management problems. Since the rule was "sign the contract anyway", while the product was not clearly defined and was only partially developed, early customers had many problems with the product. This caused a lot of internal problems also, because it was not clear whether some activity fell under product development or user assistance; nor was it clear who had to do what. After marketing, some development, some user assistance, some hardware acquisition, etc., according to an unpredictable flow of events.

4.2.6 Partial Recovery

Eventually, it was realised that the way the firm was managing the project was leading to disaster. Thus, a real effort was made, first to define responsibilities clearly (who was responsible for the design, and also responsible for the distribution, etc.), and second, to achieve a clear picture of the state of the product, of its weaknesses, of the effort required to fix them, etc. This was done even at the expense of slowing down the project, increasing costs, and reducing sales. Thus, people had to resist an initial feeling that the restructuring of the project was impeding "real" progress.

After a while, however, the improvement became apparent, so that eventually, the product really existed and full documentation was available. The company actually

started to ship the product and earn money from it, although far less then expected initially, mainly because the delayed introduction of the product caused it to enter a more competitive market than anticipated.

☞ **Check Your Progress 1**

1) _____ is best when speed of development is the most important goal and the problem is well understood.

2) _____ is best when communication among engineers is necessary for achieving a solution.

4.3 BENCHMARK TESTING

The term “benchmark” was derived from the days when the machinist in a factory would use measurements at each bench to determine if the parts he was machining were satisfactory. In the computing field, to compare one system with another, you would run the same set of “benchmark” programs through each system.

A benchmark is a sample program specially designed to evaluate the performance of different computers and their software. This is necessary because computers will not generally use the same instructions,, words of memory or machine cycle to solve a particular problem. As concerned with evaluation of software, benchmarking is mainly concerned with validation of vendor's claims in respect of the following points:

- minimum hardware configuration needed to operate a package.
- time required to execute a program in an ideal environment and how the performance of own package and that of other programs under execution is affected, when running in multi-programming mode.

The more elaborate the benchmarking, the more costly is the evaluation. The user's goals must be kept in mind. Time constraints also limit how thorough the testing process can be. There must be a compromise on how much to test while still ensuring that the software (or hardware) meets its functional criteria.

Benchmarks can be run in almost all types of systems environments including batch and on-line jobs streams and with the users linked to the system directly or through other methods.

Common benchmarks test the speed of the central processor, with typical instructions executed in a set of programs, as well as multiple streams of jobs in a multiprogramming environment. The same benchmark run on several different computers will make apparent any speed and performance differences attributable to the central processor.

Benchmarks can also be centered around an expected language mix for the programs that will be run, a mix of different set of programs and applications having widely varying input and output volumes and requirements. The response time for sending and receiving data from terminals is an additional benchmark for the comparison of systems.

☞ **Check Your Progress 2**

1) A _____ is a sample program specially designed to evaluate the performance of different computers and their software.

2) _____ charts can help a manager identify schedule bottlenecks immediately, even mechanically.

4.4 SUMMARY

An appropriate organisation may have to take into account goals other than speed of development. Among these other important goals are: lower life-cycle costs, reduced personnel turnover, repeatability of the process, development of team members at junior level into senior members, and widespread dissemination of specialised knowledge and expertise among personnel.

At different levels of an organisation, different levels of risks can be tolerated. For example, a project manager may not want to tolerate any delay in the schedule, while another member at the level of team leader might be more concerned with the reliability of the product. A project manager might not be able to tolerate the risk of running overbudget by more than 10% while a higher level manager who is more aware of the value of early entry into market may be more concerned with the time taken by the project and might be willing to overspend the budget by much more if the product can be produced sooner.

Benchmarks can be run in almost all types of systems environments including batch and on-line jobs streams and with the users linked to the system directly or through other methods.

4.5 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Centralised control

2) Decentralised control

Check Your Progress 2

1) Benchmark

2) PERT

4.6 FURTHER READINGS

1) *Software Engineering*, Ian Sommerville; Sixth Edition, 2001, Person Education.

2) *Software Engineering – A Practitiner's Approach*, Roger S. Pressman; McGraw-Hill International Edition.

Reference websites

- http://www.rspa.com
- http://www.ieee.org
- http://www.ncst.ernet.in