SOEN6841
Software Project Management

Dr. C. Constantinides, P.Eng., ITCP/IP3P

Department of Computer Science and Software Engineering
Concordia University

August 27, 2012

Contents

Preface

Once you finish school, the most difficult part of your careers is to start thinking as engineers. What do I mean by this?

My experience from undergraduate school was that some people tend to form likes and dislikes of programming languages very similar to other likes and dislikes: Java vs. C++, IBM's vs. Apple, Canadiens vs. Maple Leafs, Arsenal vs. Manchester Utd, etc. This may be fun for some time, but professionalism is not about having fun.

The first thing that I believe (and suggest) you will have to do is to get emotionally detached from technology and as engineers choose whichever tools are best suited for your tasks.

Furthermore, you may like to specialize in one area (e.g. design, or testing) but this does not make other areas less important. The development of large-scale software is achieved through the synergy of many activities, each one producing collections of artifacts which communicate and document our thoughts and plans, prove a concept as well as supporting other activities.

In my opinion every artifact produced during development is important: We have artifacts produced in a natural language (e.g. English, French), others produced in a modeling language (e.g. UML) and others produced in a formal system (e.g. C++, Java). A good software engineer must be fluent in all different languages because we use each one for a different purpose. Indeed, we can argue that implementation is the one artifact which is vital because code is absolutely necessary for a system to run. However, in the absence of a requirement-, analysis-, and a design model, our level of confidence on the

correctness of the system is rather low (and the maintainability of the system is very low). If we are not 100% convinced, we will not be able (nor will it be ethical to) convince the system stakeholders.

A common saying among programmers is "Go to the code. The code is its own documentation." Yes, we can do that. In fact we should be able to read and understand code. However, the code describes the rationale of developers at a very low level. It is not very helpful because a) as a manager you will have to prove that the design is correct and disallow the implementation of some wrong design, and b) a formal system is rather cryptic when it is the only means of communication.

To maintain a high level of confidence, we first must build confidence during requirements analysis (validation) and maintain it throughout the development life cycle. Remember that as a project manager you are the one with liability for it (see: loosing the contract, being sued, reputation, etc.), so if the use case model is not validated you should not allow developers to proceed to analysis. If the analysis model is not validated you should not allow developers to proceed with architecture and design and if the architecture and design are not validated you should not allow developers to map the design model to implementation. If the implementation is not correct (e.g. if tests fail) or not consistent, then you should not allow developers to consider the verification of requirements successful.

Furthermore, we should realize that there is no such a thing as "documentation and source code." I would love to eliminate the word "documentation" from our vocabulary. Let us talk about artifacts and models. With the exception of the magnetic 1s and 0s, everything else (installation scripts, users manual with screen shots, test suite, source code, supporting libraries, architectural-design model, analysis model, use case model - requirements) can be regarded as "documentation." However, the problem is that many software engineering textbooks (and many instructors in many schools) place emphasis either on requirements / analysis / architecture / design *xor* (that is an exclu-

sive OR) on implementation. I think this creates confusion (and frustration) among students. Especially in the first case (emphasis on early stages only), it is very frustrating to focus on formatting (and dissemination) of reports and other issues as if a model will be mapped to implementation by magic or as if it is not our job as software engineers to map it.

It does not matter if the design model is written on some glossy slide, or on a chalkboard. What really matters is that a model is correct and satisfies some specification.

Because of the above and with all tradeoffs considered, then my view is that all artifacts are equal. In fact, there are artifacts produced even after deployment (e.g. execution traces) which are valuable for post-deployment tasks (maintenance and evolution).

Finally, a common misconception among engineers is that software engineering is some boring and bureaucratic activity whereby one produces long documents *after* implementation in order to please some line manager (or some professor). I agree that in some cases this may be so. I do argue, however, that this *should not* be so. It is *our* responsibility to clear this (and other) misconceptions on software engineering. The adoption of UML as a common and unambiguous vocabulary helps us produce relatively small documents (as compared to those produced in a non-standard modeling language where additional explanations on the meaning of models would be necessary). The advancement of modern (iterative) process methods helps us do the whole thing with confidence. Software Engineering is not about avoiding programming. In fact we cannot be a good software engineers if we are not very good programmers.

Last, please always have in mind that no discipline is static in terms of knowledge (not even palaeontology). This is particularly true for Computer Science. The advancement of areas like Software Engineering is one such example. As software engineers you will have to stay informed, to be able to adapt to new technologies and to *learn how to learn* which is what school is

meant to teach you, among other things. It has always been my strong belief that there is a clear difference between *education* and *training*. The objective of training is to give one the necessary skills to perform one particular job. On the other hand, the objective of education is to provide people with the mental skills to know how to learn: to look for, collect, select and verify information through literature search, to grasp knowledge in an incremental and associative way, as well as to reason about problems, apply positive judgment and provide innovative solutions. It also enables one to perform research in order to contribute to the provision of original knowledge. In doing that, I feel that education makes people ask questions starting with *why* (such as *why something is* and *why something works*) rather than merely *how something works*.

Chapter 1

Introduction

1.1 Software systems

The distinction between the terms *software (system)* and *program* is twofold: First, a software is normally a collection of artifacts, including implementation (code). Second, we use software to describe a large system.

1.2 Putting engineering in software engineering

The development of a software system involves much more than just programming. In this course, and in most projects you will undertake in your careers as software engineers you will have to develop or maintain medium- to large-scale systems, built by a group of people, and sometimes by a large group which can be distributed over different geographical areas.

A given problem statement has to be analyzed to be fully understood. A solution has to be devised and proved correct. Finally the solution would have to be implemented and tested. To document our thoughts throughout these activities, we have adopted different languages at various levels of abstraction. Requirements are normally written in a natural language (or, sometimes, a

combination of a natural language and a mathematical language). Analysis, high-level design (architecture) and low-level design are normally documented in a modeling language (e.g. UML) and the implementation is expressed in some formal system (programming language). As you see, these activities follow a path which starts from a high-level description of a problem all the way down to a low-level description of a solution.

Would you drive over this bridge? Can we just skip everything and start coding? The answer is yes we can. But that would be the equivalent of going to the shore of some river and start putting bricks on top of one another until we reach the other end. Is this a bridge? Would you drive your car over it? The answers are yes (it's a bridge) and no (you would most likely not drive over it). If this is a bridge, why then would you refuse to drive over it? The answer is that you would want to make sure that it would not collapse. You would feel safe(r) if you knew that the engineers who built it had actually performed analysis of the forces involved and generally have proved the concept. In doing that, civil engineers build blueprints in order to document and communicate their thoughts, as well as to proof the concept.

We apply exactly the same principles in software engineering.

Would you buy this car? Imagine you want to buy a car. You fancy some model and you go to the show room where the car dealer tells you that this car works fine "most of the times" but assures you that the next model will be an improved version. Would you buy the car? The answer is most likely no. In using technology, we have (rightfully so) come to expect a high degree of reliability.

We should expect nothing less from software engineering artifacts.

An obvious counterargument is that software engineering artifacts are not as important as other engineering artifacts such as airplanes. After all, when a plane engine fails, people die. To assess this argument (about importance of software engineering artifacts) we need to take a step back:

Software runs planes, spaceships, and radars among others. If any of these

fails, people die. The headlines will of course read "Plane crash." We will never read something along the lines of "Exception thrown but not handled." Software runs the metro and trains. If it fails, people die. Headlines will read "Metro disaster." They will never read "Concurrent protocol not properly implemented." Software runs traffic lights. If it fails people die. Headlines will say "Traffic jam causes chaos." Software runs patient monitoring systems, chemical plants, etc. If it fails, people die. Headlines will write "Chemical plant disaster." They will not read "Stack overflow; Java to be blamed." Software monitors highway traffic, communications, satellites (GPS, etc.) etc. If it fails, people can die or can get lost. Finally, software runs databases with financial, medical, and other sensitive records. If it fails, people may not die, but they may get ruined. What we are saying here is that just because software is intangible, it is no less important than other engineering and technological artifacts. In fact, its importance is increasing because our dependency to it is increasing.

1.3 Software crisis

The term *software crisis* has been popular in the literature since the 1960's. It refers to the difficulty of writing correct, understandable and verifiable computer programs. Reports indicate that many software systems are

- late.
- over-budget.
- buggy.
- hard to maintain.
- "better the next time round."

  "About US$250 billion spent per year in the US on application development. Of this, about US$140 billion wasted due to the

> projects getting abandoned or reworked; this in turn because of not following best practices and standards."
> Source: Standish Group, 1996.
>
> "10% of client/server apps are abandoned or restarted from scratch. 20% of apps are significantly altered to avoid disaster. 40% of apps are delivered significantly late."
> Source: 3 year study of 70 large IT apps among 30 European firms. Compuware (12/95).

The roots of the software crisis are complexity, expectations, and change. Webster dictionary defines *crisis* as "an unstable or crucial time or state of affairs in which a decisive change is impending." (from Greek '$\kappa\rho\iota\sigma\iota\varsigma$', 'krisis', verb '$\kappa\rho\iota\nu\epsilon\iota\nu$', 'krinein', meaning '*to decide*'). If writing correct, understandable and verifiable computer programs is still difficult over 40+ years when the problem was initially observed, then this is not a crisis but an epidemic. In fact, if we want to be strict in our vocabulary, then this problem is rather a pandemic.

1.4 Software projects: Characteristics

The Guide to the Project Management Body of Knowledge[1], defines project as "a temporary endeavor undertaken to achieve a specific and unique product or service."

There are certain characteristics shared by all projects:

- Tasks are non-routine. Between a *job* (which normally involves the repetition of well-defined and well understood tasks with little uncertainty) and *research* (where the outcome may be very uncertain), a project lies rather in the middle.

[1]The PMBOK Guide is an internationally recognized standard, that provides the fundamentals of project management.

- Planning is required. The plan aims at a specific target, namely certain requirements need to be implemented (in the form of an artifact, product or service).

- Subject to constraints: Budget, human and technical resources, time.

- Normally the work involved in a project is carried out for someone else (e.g. clients).

- A project normally involves several people (sometimes scattered across organizations and geographical areas) with different specialisms (e.g. engineers, architects, etc.).

- The work of a project is carried out in several different phases (e.g. analysis, design).

Are software projects different from projects in other domains? In short, no. However, software projects pose certain characteristics which make them inherently different (and perhaps more prone to problems).

- Software is invisible and intangible. As a result, our measures (e.g. of size) are different from the measures used to describe physical artifacts (e.g. bridges, roads).

- As a consequence to the above, the progress in a software project is rather difficult to be assessed because it is not always visible. Consider for example an improvement over the implementation of some computation, where a certain algorithm has been deployed to perform a task. This will hardly show as part of the observable behavior of the system and it might even be difficult to measure (e.g. in terms of response time).

- Software systems are inherently complex. The evolution of programming languages and systems has enables us to implement complex requirements, but it has also sparked out demands for even more complex

requirements to be addressed in the future. As a result, problem domains are getting more complex (e.g. concurrent, distributed systems).

- As a consequence of the above, software systems are subject to a large degree of demands for change (new requirements are added, existing requirements are modified or even dropped).

- Whereas physical artifacts obey the laws of physics, software systems must obey the laws of mathematical logic (consistent and well defined) as well as the laws of humans (not always consistent, not always well defined).

It is also important to note that not all software projects are the same. Distinguishing different types of projects is important as different types of tasks will need different approaches (e.g. information systems versus embedded systems).

1.5 Characteristics of a successful project

Having described the software crisis and the characteristics of projects, it should be straightforward to list the three characteristics of a successful project:

1. On time (schedule).

2. Within budget (cost).

3. Meeting their specifications (quality).

A successful project meets the needs and expectations of its stakeholders. A stakeholder in a project is any person or organization that has any (legitimate) interest in the project.

In general, stakeholders will be users/clients or developers. They normally have different motivations and objectives. Stakeholders may be within the

project team, outside the project team but within the same organization, or outside both the project team and the organization.

"*What do we have to do to have a success?*" We can define the objectives of the project in the form of a sequence of postcondition-like statements and say "The project will be regarded as a success if..." This helps us to focus on the objectives (requirements) rather than on the activities involved.

Very often, objectives are defined in terms of functional and nonfunctional requirements. Functional requirements (FRs) describe the "black-box" behavior of a system. On the other hand, nonfunctional requirements (NFRs) is a collective term used to describe quality attributes or constraints. Note that there may be tradeoffs between several parameters of the project, for example tradeoffs between budget and the time, tradeoffs between FRs, tradeoffs between NFRs, or tradeoffs between NFRs and cost.

The acronym **SMART** is used to assist people in setting down good requirements.

S Specific. Requirements should be concrete and well-defined.

M Measurable. The satisfaction of the requirements can be objectively judged.

A Achievable. Requirements should be within the power of the individual or group concerned.

R Relevant. The requirements must be relevant to the true purpose of the project.

T Time constrained. There must be a defined point in time by which the objective should be achieved.

1.6 A brief review of the ISO 12207 standard

ISO 12207 is an ISO standard for software lifecycle *processes*. It defines all tasks required for developing and maintaining software. It involves five lifecycle

processes. Each of these processes is broken down into *phases* and each phase includes several *activities*.

Acquisition. It covers the activities involved in initiating a project (including system requirements elicitation and analysis, acceptance criteria, preparation of a contract).

Supply. Development of management plan (including activity planning).

Development. It includes software requirements engineering, analysis, (high- and low-level) design, implementation, testing.

Operation. Deployment of the software system (can include assisting users on how to use the system).

Maintenance. Add new requirements, modify (or drop) existing requirements, fix defects.

1.7 Management

Management involves the following activities:

- Planning: Deciding what is to be done.
- Organizing and staffing: Making arrangements for human and technical resources required.
- Directing and monitoring: Giving instructions and checking on progress.
- Controlling and innovating: Taking actions to bring project back on track.
- Representing: Liaising with stakeholders.

1.8 Why software projects fail

We can identify several factors which affect the failure of projects:

- Ignorance of management over engineering (IT) issues.
- Deadline pressure.
- Poor role definition: Who does what. Who has access to what (technical) resources.
- Requirements can be incomplete, ambiguous, inconsistent, or not measurable. Success criteria can be incorrect or undefined.
- Lack of (developer) knowledge of application area.
- Poor estimation and planning. Casual estimates of milestones.
- Ineffective environment. Unavailability of the right tools.
- Lack of reliable documentation (especially for maintenance tasks).

1.9 References

Hughes, B. and Cotterel, M., Software project management, 4th edition, McGraw-Hill, 2006.

Chapter 2

Overview of software maintenance

2.1 Introduction

The evolution of programming paradigms and languages allows us to manage the increasing complexity of systems. Furthermore, we have introduced (and demanded) increasingly complex requirements because various paradigms provide mechanisms to support their implementation. As a result, complex requirements constitute a driving factor for the evolution of languages which in turn can support system complexity. In this circular relationship, the maintenance phase of the software life cycle becomes increasingly important and factors which affect maintenance become vital. In this article we review the notions of software aging and discuss activities undertaken during maintenance. Most of the material is taken from (Constantinides and Arnaoudova, 2008).

2.2 Aging in software

In the literature, many authors tend to have drawn analogies between software systems and biological systems (ISO/IEC 12207:1995(E); Parnas, 1994; Jones, 2007). Two such notable examples are the widely used notions of aging and

software life cycle, implying that we can view software systems as a category of organisms. This analogy is convenient because it creates certain realizations about software. First, we note that systems exist (by operating as a community of intercommunicating agents) inside a given environment. Furthermore, much like their biological counterparts, they evolve (to adapt to their environment) and they grow old. Finally, when speaking of the life cycle of software, we also imply the unavoidable fact that software systems eventually die.

However, the causes of software aging are very different from those of biological organisms or those that cause aging in other engineering artifacts. Unlike biological organisms (such as humans) software systems are not subjected to fatigue or physical deterioration. Unlike other engineering products (such as machinery and structures), software systems are not subjected to physical wear caused by factors such as friction and climate. Aging in software systems is predominantly (but not always) caused by changes that take place in their surrounding (operating) environment.

In his seminal paper on aging, author David Parnas (1994) describes two causes of software aging: The first factor, referred to as lack of movement, is the failure of owners to provide modifications to the software in order to meet changing needs (requirements) of its environment which results in end-users changing to newer products. The second factor, referred to as ignorant surgery, is the careless introduction of changes in the implementation which can cause the implementation to become inconsistent with the design, or even to introduce new bugs. This latter factor is associated with two significant implications: The first is a bloating of the implementation, resulting in a reduction in performance (memory demands, throughput and response time). This weight gain makes new changes difficult to be introduced quickly enough to meet market demands. The second implication is a phenomenon known as bad fix injection (Jones, 2007) which refers to the introduction of errors during maintenance resulting in a decrease in reliability. As a result, software systems become unable to be competitive in the market, thus loosing customers

to newer products.

2.3 Measures to prolong aging

Certain measures are proposed in the literature (Parnas, 1994) to prolong aging such as:

2.3.1 Retroactive documentation

The quality of documentation can be upgraded (retroactive documentation). For example, reverse engineering is a model transformation activity which can read implementation and produce an up-to-date design model.

2.3.2 Retroactive modularization

Since we cannot really predict the actual changes, predictions can be made about the types of changes, such as changes to the graphical user interface. Parnas (1994) recommends re-organizing the software in such a way so that elements which are most likely to change, such as the user interface, are confined to small amounts of code (retroactive modularization). A similar view is shared by Fayad and Altman (2001) through an architectural pattern to support software stability where the architecture is built around two notions, conceptualized as two concentric circles. In the inner circle, we have aspects of the environment that will not change. These aspects will constitute a stable core design (and thus a stable software product). In the outer circle, or periphery, we define a design which will allow changes to be introduced.

2.3.3 Amputation and major surgery

We can eliminate components which are of very low quality (amputation) or eliminate redundant components (major surgery).

These measures take place during the period of operability of a software system and are explicitly treated as a separate phase of the software life cycle which is discussed subsequently.

2.3.4 Aging is inevitable

It is important to stress here that our objective is merely to prolong, but not to permanently reverse or stop aging. Our ability to design for change depends on our ability to predict the future. We can do so only approximately and imperfectly. Over a period of years changes that violate original assumptions will be made, design and other artifacts will never be perfect, reviewers are bound to miss flaws, etc.

2.4 Software maintenance

ISO/IEC and IEEE define maintenance as the modification of a software product after delivery to correct faults, improve performance (or other attributes) or to adapt the product to a modified environment (ISO/IEC 14764:2006(E); IEEE Std 14764-2006). The importance of maintenance lies on the following observations: 1) Surveys indicate that it is an activity which tends to consume a significant proportion of the resources utilized in the overall life cycle (consequently consuming a large part of the costs) and 2) Reliable changes to software tend to be time consuming. Prolonged delays during software change may result in a loss of business opportunities.

The objective of maintenance is not to stop the unavoidable effects of aging, but to provide techniques and tools to understand its causes, to limit its effects and to prolong the life of software systems.

Maintenance is not a uniform activity and as the type of required changes may vary, four different types of maintenance can be identified which are also defined in the ISO/IEC; IEEE international standard. Corrective maintenance includes all changes made to a system after deployment to correct prob-

lems. Preventive maintenance includes all changes made to a system after deployment to correct faults in order to prevent failures. Perfective maintenance includes all changes made to a system after deployment to provide enhancements, or recoding to improve performance, maintainability and other attributes. Adaptive maintenance includes all changes made to a system after deployment to support operability in a different (software or hardware) environment. The ISO/IEC; IEEE international standard provides a classification scheme by grouping the former two under correction and the latter two under enhancement. Adaptive and perfective types of maintenance are shown in the literature to consume a significantly large proportion of all maintenance effort. Corrective and preventive types of maintenance are reported to consume a relatively small proportion of the overall maintenance effort. It is important to note, that the different types are not mutually exclusive but rather they can be combined concurrently to be mutually supportive.

Also, the four maintenance types do not refer to single activities. Jones (2007) lists twenty three discrete topics which involve a modification of an existing system often described under maintenance.

2.4.1 Stages of maintenance and the staged model of the software life cycle

In the literature, Bennett and Rajlich (2000) define a model whereby a software system undergoes distinctive stages during its life: Initial development, evolution, servicing, phase-out, and closedown (See Figure 2.1).

Initial development would produce a deployable system (the first operating version). After deployment, evolution would extend the capabilities of the system, possibly in major ways. To evolve easily, software must have both an appropriate architecture and a skilled development team. When these are lacking, evolution is no longer viable and the software enters the servicing or saturation stage when it is considered aging, decayed, or legacy. Only small

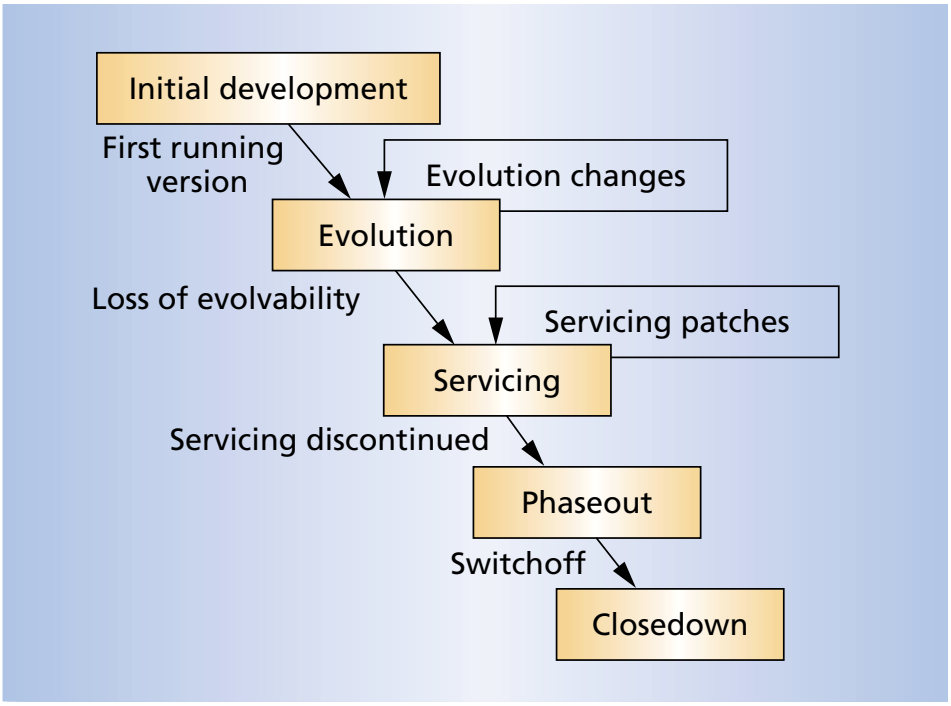

Figure 2.1: Staged model.

changes are possible during this stage. Changes are both difficult and expensive, so developers minimize them or do them as wrappers, which are simply modifications to inputs and outputs, leaving the old software untouched. Still, each change further degrades the architecture, pushing it deeper into servicing.

Maintainers often encounter what Bennett (1995) describes as the legacy dilemma: On one hand, a system (or component) is valuable and replacing it may not be a viable (cost effective) solution (e.g. large volumes of data may have to be converted). On the other hand, the cost of maintenance is becoming high and requests for changes cannot be sustained. When faced with legacy systems, organizations have to adopt a strategy which is based on economics (i.e. cost of coping with the current system vs. the cost of investment of improvement) and management (e.g. a replacement system would normally require training of end-users). Figure 2.2 summarizes various options based on two factors, namely business value and quality (adopted from Sommerville, 2007).

		QUALITY	
		Low	High
BUSINESS VALUE	Low	Expensive to keep, with a low return. Scrap. Would require a modification of business processes which rely on it.	Continue with maintenance while it remains cost effective. Once maintenance becomes expensive, then scrap.
	High	Expensive to maintain. Options include reengineering or replacement by off-the-shelf systems.	No special measures required. Continue normal system maintenance.

Figure 2.2: Maintenance options based on business value and quality factors.

Finally, once servicing is no longer viable the system enters a phase-out stage where deficiencies are known but not addressed. During phaseout, no more servicing is undertaken. Users of the software must work around its deficiencies. At close-down, the system is withdrawn from the market: The company shuts down the software and directs users to a replacement system, if one exists.

In an alternative model (versioned staged model), during evolution at certain intervals, a company completes a version of its software and releases it to customers. That version subsequently enters the servicing stage whereas the system continues to evolve in order to produce the next version (See Figure 2.3).

Central to any maintenance activity is the notion of change, discussed in the next section.

2.5 Software change

Whether new requirements are introduced or existing requirements are refined or dropped, the notion of change is a fundamental activity during evolution

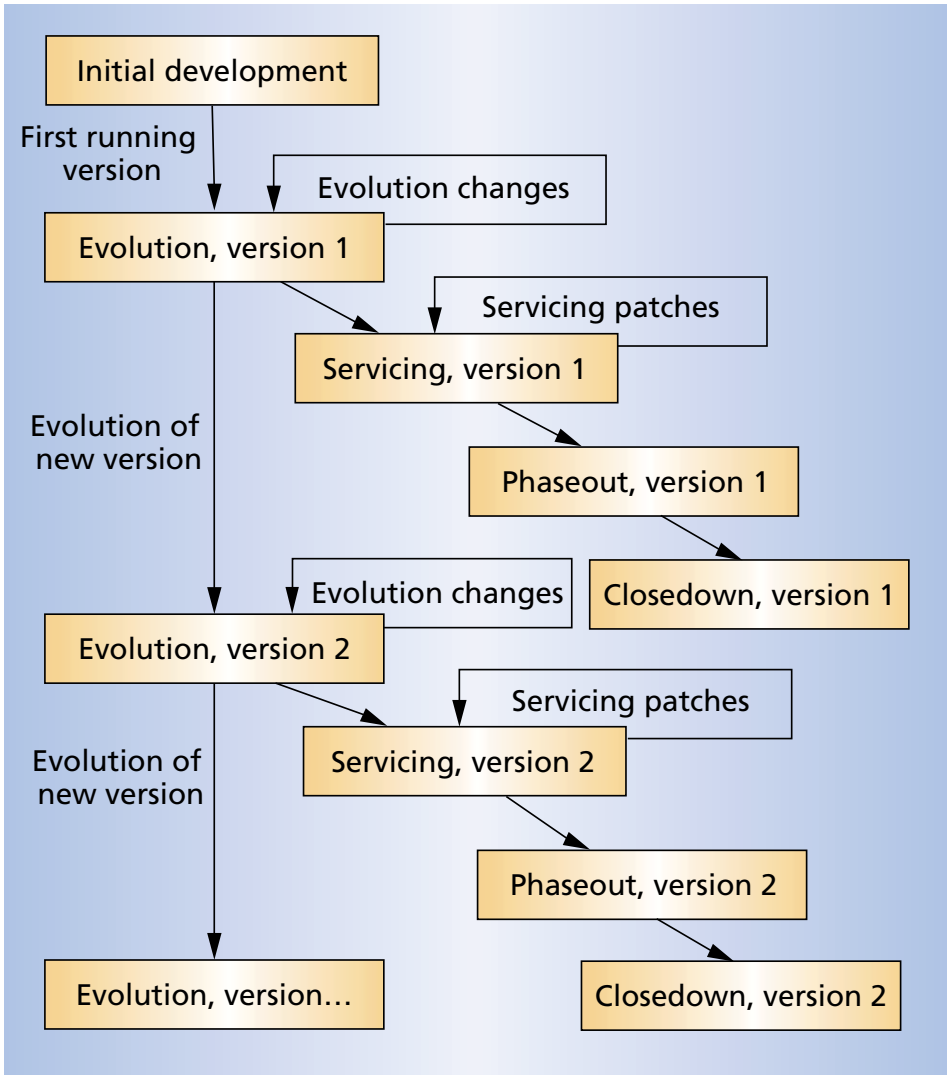

Figure 2.3: Version staged model.

and servicing. Bennett and Rajlich (2000) describe a change mini-cycle as one which involves a number of activities: request for change, planning phase, change implementation, verification, and documentation update.

2.5.1 Request for change

Requests for changes originate from a number of sources including bug reports, system enhancement requests from end-users or changes in standards.

2.5.2 Planning phase

The planning phase involves comprehension and change impact analysis.

Comprehension

Comprehension refers to activities that humans perform in order to understand, conceptualize and reason about a program or software system. It normally consumes a large proportion of resources during the overall maintenance phase.

Often it is the case that maintainers are not the initial designers. Furthermore, design artifacts, if at all present, are often incorrect or incomplete. For systems that have been undergone ignorant surgery, comprehension becomes particularly difficult, as it has to bridge the gap between what the original designers had envisioned (initially represented by a coherent and structured description of a model) and the actual system (whose structure may have disintegrated over time). This tendency of systems to experience a structural disintegration over time (also known as entropy) and the subsequent increase in complexity was discussed by Lehman (1980) as one of the laws that govern evolving systems (known as Lehman's second law).

One method to achieve comprehension is through reverse engineering (sometimes the two terms are used interchangeably), which transforms a representation of the system to a higher level of abstraction. This activity involves

analysis only, without any modification. The objectives of reverse engineering include the recovery of information which is either not apparent in the code or it is difficult to detect as well as to facilitate reuse by detecting candidates for reusable components. A variation (or specialization) of reverse engineering is design recovery whose objective is to identify meaningful high-level abstractions, which are not identifiable by examining the implementation. Reverse engineering falls under the notion of translation in the model transformation taxonomy (Mens and Van Gorp, 2006), as the source and target languages are at a different level of abstraction.

For complex systems, comprehension methods rely on the study of the dependencies between program (or software) elements, such as program slicing and formal concept analysis.

Change impact analysis

Software change impact analysis estimates which parts of a software and its related documents will be affected if a proposed change is to be made. It also provides essential information that can be used to estimate the cost of a modification request (impact of change) and to plan its implementation. Arnold and Bohner (1996) divide change impact analysis into two major types, namely *dependency analysis* and *traceability analysis*.

Dependency analysis: Identifies potential consequences of a change, and estimates modifications needed in order to accomplish a change. Dependency analysis can be performed on source code, models, execution traces, or a combination of them. An example of dependency analysis of source code is to use the combination of data flow and control flow graphs in order to identify program dependencies. Performing dependency analysis on a class diagram is an example of model based dependency analysis. Dependency analysis on execution traces can be used to locate related features in code.

Traceability analysis: Focuses on concern identification from requirements to design artifacts, followed by concern identification from the design artifacts to the source code or vise versa (i.e. the identification starts from the source code and ends with the requirements artifacts).

2.5.3 Change implementation

The implementation of change involves restructuring and change propagation. Often deployed to perform preventive maintenance and to simplify or optimize a model, restructuring (or its object-oriented equivalent: refactoring) involves the transformation of one representation form into another at the same level of abstraction. As such, it falls under the notion of rephrasing in the model transformation taxonomy. One important property of restructuring is that it must guarantee the preservation of the behavior of the system. It may also detect problems which would call for changes. In the object-oriented context, strategies for refactoring have been extensively documented in the literature (Fowler et al., 1999) and are currently supported by a number of tools , perhaps the most notable of which is currently the Eclipse IDE .

Change propagation involves changes which may have to be made to dependent components to make sure that changes made in the previous step have not violated the integrity of the system.

In implementing changes, we often involve a sequence of three activities: reverse engineering - restructuring - forward engineering, which are together referred to as reengineering (or renovation). Reengineering addresses system functionalities which need to be added, refined or deleted. Essentially reengineering uses program comprehension to reimplement the program in a new form (Figure 2.4). Reengineering can be advantageous only when a return of investment can be projected (cost of reengineering vs. increase of product and process quality and business value) (Sneed, 1995) and best candidates to undergo this transformation are components with low quality but strategic to the organization (high business value) (see Figure 2.2).

2.5.4 Verification

Once a change has been implemented, we need to argue about the preservation of the system behavior and correctness.

2.5.5 Documentation update

Artifacts need to retain their inter-consistency. Computer-aided software engineering (CASE) tools can automate this process.

2.6 Transformations

We have already seen that at any level of abstraction models are expressed in some language (e.g. UML, Java, etc.). Transformations are used in many areas of software engineering, including compiler construction, software visualization and artifact generation. Based on the language in which the source and target models of a transformation are expressed, we can distinguish between *endogenous* and *exogenous* transformation (See Figure 2.5):

1. Exogenous transformation (Translation): Where the source and target languages are different (e.g. reverse engineering)

2. Endogenous transformation (Rephrasing): Where they are the same (e.g. refactoring).

2.6.1 Translation (or exogenous transformation)

A first type of translation is synthesis (or forward engineering). This involves the transformation of a higher level specification into a lower level, e.g. the transformation of a set of model elements into a set of corresponding implementation. A second type of translation is reverse engineering which is the inverse of synthesis, where the transformation creates a representation of the system at a higher level of abstraction. A third type of translation is program

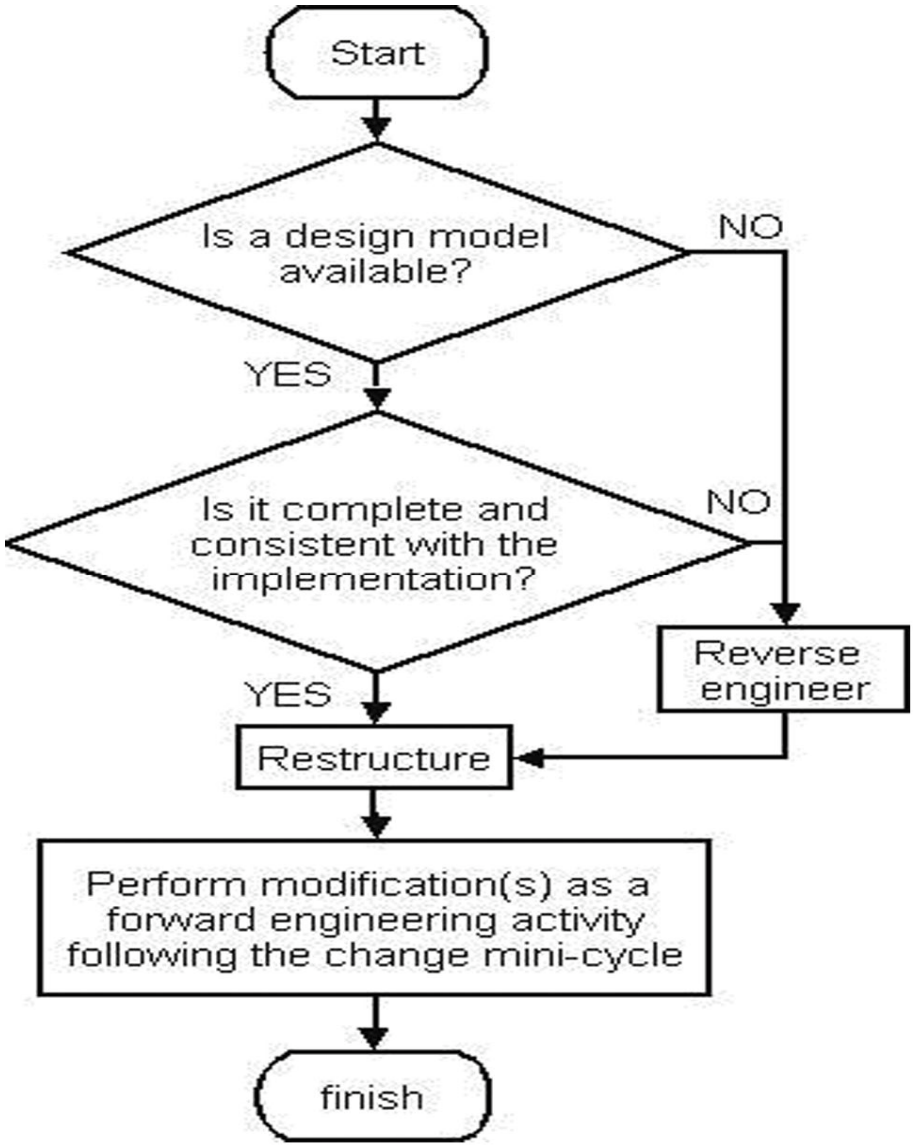

Figure 2.4: Integration of reengineering activities in the change mini-cycle.

migration where a program written in one language is transformed into a program written in another language (but keeping the same level of abstraction).

2.6.2 Rephrasing (or endogenous transformation)

A first type of rephrasing is program optimization which is a transformation aimed at improving operational qualities (e.g. performance) of a program. A second and perhaps a more commonly known type is restructuring (refactoring) which involves the transformation from one representation form to another (at the same abstraction level), while preserving the external behavior of the subject system. A change of the internal structure that aims at the improvement of quality attributes of the model. A third type of rephrasing is reengineering (renovation) which is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. It is often presented as consisting of three steps, involving: Reverse engineering, restructuring, forward engineering.

2.7 Horizontal vs. vertical transformations

A *horizontal transformation* is one where the source and target models reside at the same abstraction level. One typical example is *refactoring*. On the other hand, a *vertical transformation* is one were the source and target models reside at different abstraction levels. A typical example is synthesis. Note that the dimensions *horizontal vs vertical* and *endogenous vs. exogenous* are orthogonal.

2.8 Reverse engineering

Reverse engineering can be seen as a common translation transformation. Most programmers work on software systems that other people have designed and developed. In systems with poor architectural and design artifacts, the code is

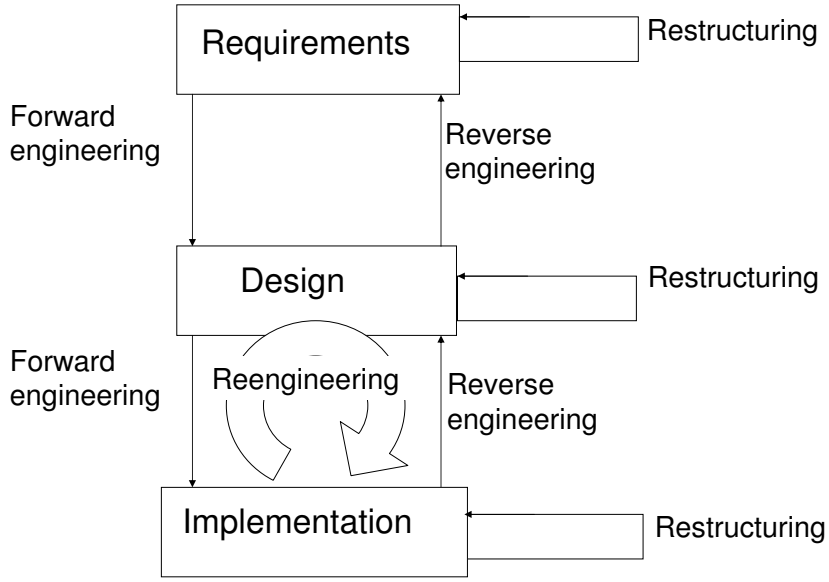

Figure 2.5: Transformations.

the only reliable source of information about the system. Reverse engineering is the process of analyzing a subject system to identify its components and their interrelationships and create representations of the system in another form, or (usually) at a higher level of abstraction. Representations include static (class diagram) and dynamic (interaction diagrams) model. Note that reverse engineering involves only analysis, not change.

2.8.1 Objectives and goals of reverse engineering

The main objective of reverse engineering is to cope with complexity and obtain understanding (comprehension) of code. In order to cope with this complexity, goals include the recreation of the model for an existing system (either because the model was lost or never created, or because it became out of sync with the source code), to generate alternate views, to detect side effects, and to facilitate reuse (detect candidates for reusable components).

2.9 Reengineering

Reengineering can be seen as a common rephrasing transformation. It uses program understanding to reimplement the code in a new form. It involves a combination of reverse engineering for comprehension and forward engineering to reexamine which functionalities need to be retained, deleted, added or modified.

2.9.1 When to reengineer

When does reengineering provide the most viable option? We can identify a number of situations. Initially, when system changes are mostly confined to a part of the system then reengineer that part. Also, when hardware or software support becomes obsolete and when tools to support restructuring are available.

There are also a number of cost factors to take into consideration. The quality of the software to be re-engineered, the tool support available for reengineering, the extent of the data conversion which is required, and the availability of expert staff for reengineering.

2.10 Examples

Example 2.1. "It makes no sense to talk about software aging." Provide arguments for and against this statement.

Solution:

Arguments in favor can be the following:

- Software is a mathematical product, and mathematics does not decay with time.
- If a theorem was correct in the past, it will be correct in the future.
- If a program is correct today, it will be correct in the future.

- If a program will be wrong in the future, it must have been wrong when it was written.

A rebuttal can focus on the causes of aging:

- Failure to modify software systems to meet changing requirements.
- The result of changes that are made: degrading structure, introduction of bugs, inconsistent/inaccurate documentation.

Example 2.2. Briefly describe what is meant by "reusable components" and provide three examples. Describe four motivations for their deployment.
Solution:

A "reusable component" is one that can be used in many different contexts. Examples include operating systems, database management systems, and office applications.

Motivations for their deployment include 1) they can speed up the development process, 2) they can cut down costs, 3) they can increase productivity, and 4) they can improve the quality of software.

Example 2.3. Define "reverse engineering" and compare it with "reengineering".
Solution:

Reverse engineering: A transformation of a model representation to a higher level of abstraction in order to help with comprehension. It involves the identification of system components and their interrelationships.

Re-engineering: A transformation that involves 1) reverse engineering to obtain comprehension, 2) possibly restructuring and 3) forward engineering to add/delete/modify requirements.

Example 2.4. Briefly describe any factors which can constitute a motivation to adopt reengineering.
Solution:

- Current software becomes obsolete.
- Changes are confined only to part of the current system.
- If the cost to reengineer is significantly less than the cost to develop new software. If there is a higher risk in new development (e.g. staffing problems).

Example 2.5. Briefly address the "legacy dilemma." (as discussed in K. Bennett, "Legacy systems: Coping with success", IEEE Software, January 1995.)

Solution:

The "legacy dilemma" can be described as follows: On the one hand the system is very valuable and replacing it may be too expensive. On the other hand, the system is becoming too expensive to maintain and the demands of the marketing department for alterations cannot be sustained. Business opportunities are lost.

Example 2.6. List the steps involved in the "change mini-cycle" as discussed in (Benett and Rajlich 2000).

Solution:

1. Request for change.
2. Planning phase.
   (a) Comprehension.
   (b) Change impact analysis.
3. Change implementation.
4. Verification.
5. Documentation update.

Example 2.7. With the use of a table, classify the following transformation instances based on a) the source and target languages and b) the level of abstraction: "refactoring", "program optimization", "language migration", "forward engineering", "reverse engineering" and "code generation."

Solution:

		Level of abstraction:	
		Horizontal	**Vertical**
Source and target languages:	**Same**	Refactoring.	Optimization.
	Different	Language migration.	Forward engineering. Reverse engineering. Code generation.

Example 2.8. 1. Provide three brief arguments to support why software maintenance is difficult.

2. Provide an example where a task undertaken would involve more than one type of maintenance.

Solution:

1. Arguments include:

   - Supporting documentation may not be available, or be inconsistent or incorrect.
   - Rationale may not be clear (you are maintaining code someone else has written).
   - Comprehension of code is difficult (especially if software engineering guidelines were not followed during initial development).

2. Examples include:

   - While you are modifying the system to address new requirements (perfective maintenance), you may detect some anomalies which you would have to repair (corrective maintenance).

- A combination of perfective as above with adaptive maintenance (migrating it to a new platform).

Example 2.9. 1. Briefly describe what is meant by "legacy" in the context of software systems.

2. Describe the condition(s) under which a system would be regarded as "legacy."

3. Briefly describe viable strategies for the evolution of legacy systems. For each one, clearly describe the factors which can determine the strategy.

Solution:

1. A legacy system is one which is vital to an organization but expensive to maintain.

2. No more evolvable (major changes). Only small changes are feasible. The above conditions can be the results of lack of ignorant surgery (degraded structure; changes per-formed are now inconsistent and may even have invalidated the original concept).

3. Factors and strategies are as follows:

   Low-quality, low-business value: scrap (and modify business process so that it no longer is required).

   Low-quality, high business value: re-engineered (if feasible) or replaced (if suitable systems exist).

   High-quality, low-value: replace, scrap, or continue maintenance.

   High-quality, high business value: continue maintenance.

Example 2.10. 1. Provide an example where a task undertaken could involve more than one type of maintenance (and describe the types of maintenance involved).

2. Even though some authors use the terms “maintenance” and “evolution” as synonyms, the *staged model* of maintenance makes a distinction. Briefly describe how this distinction is made.

Solution:

1. While you are modifying a system to address new requirements (perfective maintenance), you may detect some anomalies which you would have to fix (corrective maintenance). Another example would be a combination of perfective maintenance (as above) with adaptive maintenance (migrating it to a new platform).

2. Maintenance is described as any activity undertaken over the software system after its deployment. In the staged model, evolution is described as any activity which results in possibly major changes in the capabilities and the functionality of the system.

Example 2.11. Your new company is using a relational database management system (DBMS) written by its IT department several years ago. You have accepted the position of head of the IT department. During your first briefing, you are being told about a problem which you have to take a decision on how to handle. In the DBMS there exists a relatively large component performing some vital task to the business and directly interacts with the database. However, extensive modifications over the years have resulted in the component to become very slow because of code bloating. Its core architecture has also deteriorated. As a result, demands of the marketing department for alterations (to meet new requirements) cannot be sustained (*people are afraid to touch it to perform any major modifications*) and business opportunities are lost.

1. How would you characterize the component, what is its relation to the property of *evolvability* and what dilemma are you facing?

2. Under these conditions, briefly describe two recommendations you would provide on how to best deal with this component. In your recommen-

dations, briefly describe any and all applicable cost factors under which each recommendation can be a viable solution.

Solution:

1. This is a typical case of a legacy system. Evolvability seems to have been lost since major changes do not seem to be feasible. The *legacy dilemma* is that on one hand the component is very **valuable** to the organization. On the other hand, the component is becoming too **expensive to maintain**.

2. Two recommendations are as follows:

   (a) **Reengineer** the component. Cost factors include existing tools, expert personnel being present and being available, and the extend of any necessary data conversions.

   (b) **Replace** the component. Cost factors include a suitable component being available, and the cost to acquire this component is preferred over the cost of reengineering.

Example 2.12. You are hired as the manager of an IT team. Your first job is to solve problems with a software system at a client's side, delivered ten years ago and written by members of the team who have now retired. Your team is made up of young graduates who have just been hired, with some prior experience in maintenance but with no experience with the particular system. The system provides value to a client organization and over time requests for changes were sustained. Significant changes are likely to be requested very soon with the introduction of new laws and regulations. Problems with the system are mostly confined to few of its major subsystems and they include slow response time, low throughput and high memory demands. Additionally, the client has complained that some parts of functionality have over the years acquired several bugs. You read about the history of the system and

you find out that any changes made after the system was deployed were handled by temporary staff and were made only to the source code. Supporting documentation is available in the form of architectural and design models.
During your first meeting, Ann, a member of the team, says that the software has aged and that the architecture and design have deteriorated to the degree that she is "afraid to touch it", particularly to introduce major features. This is immediately disputed by Gregory, a member of upper management who attends the meeting, who claims that to talk of "aging" does not make any sense, as the software is relatively new. To support his claim, Gregory says that the software's deployment date is indicated on the report containing the architectural and design documents which lies on the shelf. You check the dates, and this is indeed true. The system was developed ten years ago. You confirm that this is the original report delivered at the time of deployment. Gregory further claims that there is nothing wrong with the architecture and design (after all, the system used to provide a solid service once it was first deployed and most people made any changes to it after its deployment were descent coders) and suggests that the best advice he can give to Ann is to spend time carefully studying the code. Finally, Gregory suggests that in order to address the performance problems, new hardware must be purchased. As far as the bugs are concerned, he claims that bugs are inherent in software systems and these can be easily addressed in the next version.

1. (4 points). What response can you provide to Gregory as far as the "aging" argument is concerned? In your answer, describe the particular cause of aging and show how this can be a cause to the problems mentioned above (hint: there are three problems).

2. (4 points). Briefly describe the maintenance stage at which the system is in, relative to the following: a) the characteristics that bring a system into this stage, b) the degree to which changes can be made.

3. (4 points). What do you respond to Gregory's suggestion to Ann about

how to best comprehend the system? What alternative(s) can you suggest, if any, to achieve comprehension?

4. (4 points). As a manager of the team, what type of dilemma are you facing with this system?

5. (4 points). Faced with the current system, what strategies can you adopt and what conditions would make each strategy a viable option? (hint: two strategies) If a strategy entails any type of model transformation, list every step and provide a brief one-line description of the objective of each step.

Solution:

1. This is a typical example of "ignorant surgery" which refers to the careless introduction of changes to the implementation which causes it 1) to become inconsistent with the architecture and design, 2) to become bloated resulting in low performance, and 2) to introduce new bugs.

2. a) The system lies in the servicing stage, as 1) appropriate architecture does not exist and 2) team is not experienced (lacks knowledge of the problem domain and of maintenance tasks), b) As is, major changes are difficult and expensive, and only small changes are possible

3. As is, there seems to be a large gap between what original designers produced and the actual system, i.e. architecture and design have disintegrated. Besides, reading extensive amounts of code is neither practical nor feasible. We can reverse engineer the code and obtain an up-to-date picture of the design which can be more readable.

4. The "legacy" dilemma is when on one hand you have a system that provides value to an organization and replacing it may not be viable. On the other hand, the cost of maintaining it is high and as is, any requests for changes cannot be sustained.

5. The system is placed under "High business value - Low quality." One option is to replace it, should a suitable candidate is available (in terms of functionality) and the cost of purchasing outweighs the cost of reengineering (see below).

   Another option is reengineering, especially the component which tends to be most problematic and provided tools are available as well as data conversions do not take place or are kept to minimum. The team may also require some training especially since it lacks prior experience with maintenance tasks.

   Reengineering entails reverse engineering (to recreate the design model; to achieve comprehension and cope with the complexity), restructuring (to improve quality) and forward engineering (to map the new design into implementation).

2.11 References

1. Arnold, R., and Bohner, S., Software change impact analysis, Wiley-IEEE Computer Society, 1996.

2. Bennett, K. H., and Rajlich, V. T. (2000). Software maintenance and evolution: A roadmap. In Proceedings of the 22nd International Conference on Software Engineering, Future of Software Engineering Track (pp. 73 - 87). New York, NY, USA: ACM.

3. Chikofsky, E. J., and Cross II, J. H., Reverse engineering and design recovery: A taxonomy, IEEE Software, January 1990, pp. 13-17.

4. Constantinos Constantinides and Venera Arnaoudova*, Prolonging the aging of software systems, In M. Khosrow-Pour (editor), Encyclopedia of Computer Science and Information Technology Management, Hershey, PA: IGI Global, October 2008, pages: 3152 - 3160.

5. Fayad, M. E., and Altman, A. (September 2001). Thinking objectively: An introduction to software stability. Communications of the ACM, 44(9), 95 - 98.

6. Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). Refactoring: Improving the design of existing code. Menlo Park, CA, USA: Addison Wesley Longman, Inc.

7. International Standards Organization/ International Electrotechnical Commission; Institute of Electrical and Electronics Engineers (2006). ISO/IEC 14764:2006(E); IEEE Std 14764-2006: International Standard: Software Engineering - Software Life Cycle Processes - Maintenance (2nd ed.). Geneva, Switzerland and Piscataway, NJ, USA: ISO/IEC; IEEE.

8. Jones, C., Geriatric Issues of Aging Software, CrossTalk - The Journal of Defense Software Engineering, 20(12), December 2007, pp. 4 - 8.

9. Lehman, M. M. (September 1980). Programs, life cycles and laws of software evolution. Proceedings of IEEE: Special Issue on Software Engineering, 68(9), 1060 - 1076.

10. Mens, T., and Van Gorp, P. (March 2006). A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science, 152. Retrieved February 11, 2008, from http://www.sciencedirect.com

11. Parnas, D. L. (1994). Software aging. In Proceedings of the 16th International Conference on Software Engineering (pp. 279 - 287). Los Alamitos, CA, USA: IEEE Computer Society Press.

12. Sneed, H. M. (January 1995). Planning the reengineering of legacy systems. IEEE Software, 12(1), 24 - 34.

13. Sommerville, I. (2007). Software engineering (8th ed.). Harlow, UK: Pearson Education Limited.

Chapter 3

Software development processes

3.1 Introduction

The objective of software development is to efficiently and predictably deliver a software product that meets the needs of the community of its stakeholders. A *process* is a set of ordered steps intended to reach an objective. A *software development process (or model)* is an approach to building, deploying and maintaining software. The motivation behind defining a process for software development is to manage the size and complexity of software systems.

There is no single approach to such a process, and various models exist, some of which are discussed in the subsequent subsections. However, a key principle behind any process is the breakdown of a project into manageable components. This idea lies also at the heart of the differences between processes. On one hand, a breakdown can be performed in terms of activities. As an example, one may consider a 6-month project to break down into a 1-month requirements elicitation and analysis activities, a 1-month analysis, a 2-month architecture and design activities and a 2-month coding and testing activities. On the other hand, breakdown can be performed in terms of subsets of functionality, where one goes through all activities in successive cycles or ("iterations") where each iteration produces a subset of the overall functionality. For example, the first iteration may produce 20% of the overall

functionality, the second iteration may produce an additional 30%, etc. A common technique to iterative approaches is *timeboxing* which refers to forcing a fixed length to iterations. Should it be the case that not all planned functionality can be delivered in the current iteration, timeboxing implis that some functionality is slipped to a future iteration rather than having to slip the deadline of delivery.

The two different ways to project breakdown divides processes into two categories: *Linear process models* and *iterative process models*. The latter comes in many variations (some of which are very similar), but it is important to note that there is no such thing as a good or bad process model(s). In the subsequent subsections we will discuss some of the major models together with criteria for their deployment as well as possible advantages and disavdantages.

3.2 The waterfall software development process

The "waterfall" is a software development model in which development is seen as flowing steadily downwards (like a waterfall) through the activities (called "phases") of requirements, analysis, design, implementation, testing (verification), integration, and maintenance. The model follows a linear path of development (as opposed to "iterative" – see subsequent sections). When compared to its alternatives, it is referred to as the "pure" model (See Figure 3.1).

The waterfall model proceeds from one phase to the next in a sequential manner, and the model advocates that one should only move to a new phase once the preceding phase has been fully completed (and perfected). For example, one first completes and freezes requirements specification before proceeding to design. When the design is complete, one proceeds to implementation, etc. In the waterfall model, the software system is developed not in releases but as a whole and feedback is available once the system is delivered. The

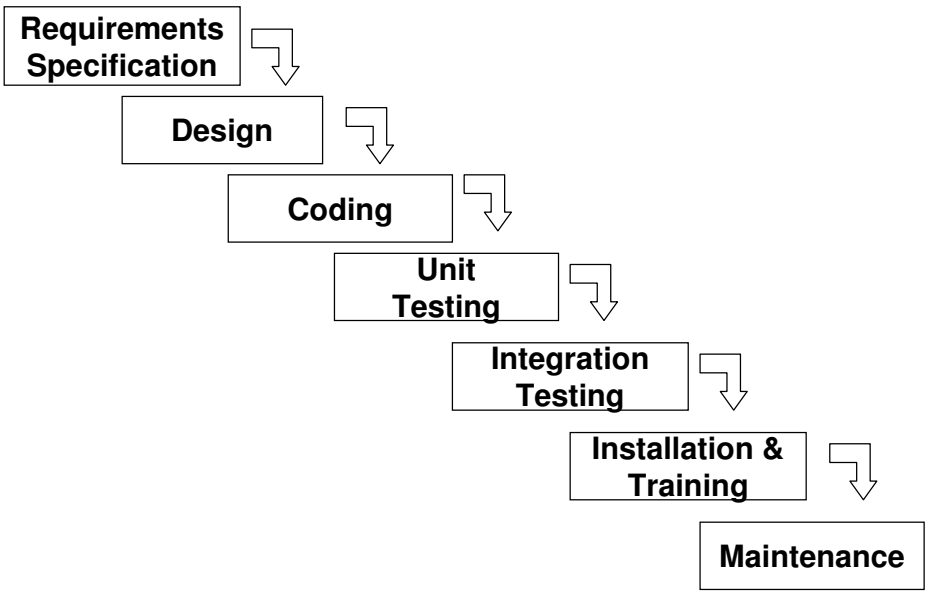

Figure 3.1: The pure linear ("waterfall") development process. Adapted from W. Royce, *Managing the development of large software systems*, IEEE Computer Society, 1970.

model provides limited (if not minimal) stakeholder feedback typically in the form of reviews and "sign-offs" (formal conclusion of a phase) between between adjacent phases of development. The model has its origins in the manufacturing and construction industries in which changes are either very costly or simply impossible. The model can be adopted in cases where both functional and nonfunctional requirements are well defined and the product is planned to be delivered in a single version.

3.2.1 Criticisms of the waterfall model

As requirements tend to change throughout development, it is usually not feasible to fully complete and "freeze" the requirements specification, then fully complete the design, then build, and only then proceed to test the software. Two associated problems with the linear model are as follows:

1. Risks are not explicitly addressed. As such, they can be encountered late in the development, and

2. Lack of flexibility of changing requirements, particularly in dynamic domains.

3.2.2 The modified waterfall software development process

In response to the criticisms of the "pure" waterfall model, the modified waterfall model realizes that feedback can lead from code testing to design (as testing of code may uncover flaws in the design) and from design back to requirements specification (as design problems may uncove conflicting or unsatisfiable requirements).

3.3 Spiral development model

Introduced by Boehm[1] in 1986, the *spiral development model* (see Figure 3.2) was the first to address the importance of iteration in software development. In this model the system requirements are defined in as much detail as possible. The notion of "iteration" (or "cycle") lies at the heart of this model. An iteration are an important concept as it is applicable in other development models (see later). An iteration follows the waterfall model and it is essentially a mini-project, containing the following development activities:

1. Determine objectives; Specify constraints: Define the problem addresses by the current iteration.

2. Generate alternatives: Define solution(s).

3. Identify and resolve risks. Risks are addressed in order of priority.

4. Develop and verify product: Work through requirements analysis, architecture, design, implementation and testing, producing a prorotype.

[1]Barry Boehm, A Spiral Model of Software Development and Enhancement, ACM SIGSOFT Software Engineering Notes, Volume 11 Issue 4, August 1986, ACM New York, NY, USA.

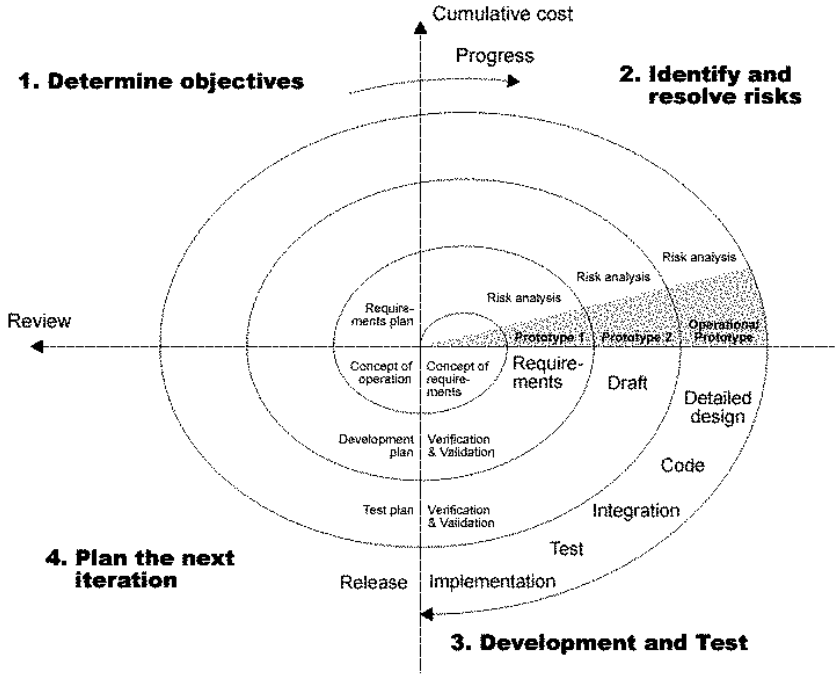

Figure 3.2: The spiral development model.

5. Plan: Prepare for the next iteration.

An iteration addresses only a subset of the initial set of requirements. Development involves a number of iterations whereby each one addresses a subset of requirements and producing a prototype (whose strengths and weaknesses are evaluated) until eventually an operational prototype is delivered. Interim prototypes may or may not be deliverables.

Additionally, each iteration includes four phases: During the first phase (upper left quadrant) developers determine objectives, constraints and alternatives. During the second phase (upper right quadrant), developers identify and resolve risks associated with the solutions defined during the first phase. We should note that (together with the cleanroom model: see later), the spiral model is the only one that explicitly includes risk management. Risk identification and analysis is performed during each iteration of the spiral. During the third phase (lower right quadrant) developers implement a prorotype associated with the risks identified in the previous phase. During the fourth phase

(lower left quadrant) developers plan the next iteration based on the results of the current iteration. The distance from the origin represents the cost accumulated by the project and the angular coordinate indicates the accomplished progress within each phase.

In its purest form, the spiral model bears a similarity to the waterfall model in that the system is available only when it is complete, not in phased releases (even though it could be adopted to provide releases). However, unlike the spiral model, the waterfall model makes no attempt to identify and tackle riskiest requirements first.

3.4 Evolutionary development model

In evolutionary development, the software requirements for the first version of the product are established, and the product is developed and delivered to its stakeholders. During development or after the first product delivery, the requirements for a second or third etc., product release are defined and implemented for the next release. These evolutions of the product are referred to as evolutionary spirals. Each evolution has the characteristics of a waterfall process. Ideally, there should be no overlaps between developments. However, if overlaps must exist, then it is vital that the development of the second release does not start until the design of the first release is or has become stable. Microsoft Windows operating system and Microsoft Word wordprocessor are examples of evolving products.

3.5 Iterative and incremental development model

An important point in this model is that the complete set of requirements is elicited, analyzed and allocated to individual small scale projects (called increments, or iterations) before full-scale development begins. The main idea behind the iterative and incremental development model is to build a system

through repeated development cycles and in relative small portions. An iteration represents a complete development cycle and it includes its own treatment of all activities (normally in varying workloads): requirements, analysis, architecture, design, implementation and testing activities. Each activity produces a set of artifacts, and the outcome of each iteration is a tested, integrated and executable system.

In the iterative and incremental development model, only partial testing is possible at the completion of each increment. The full functionality and the ability to test all requirements cannot be completed until all increments are complete by which time the system is ready for production.

It is important to note that the executable of each iteration is not some experimental throw-away prototype but a production subset of the final system. The executable of each iteration is made available to stakeholders who are expected to provide feedback. This creates an opportunity for adaptation. As a result, the system is successively enlarged and refined though multiple iterations.

Which criteria (if any) can determine the choice of requirements for a particular iteration? We can identify two possible criteria (even though these are neither explicitly addressed nor in any way enforced by the model):

1. Based on criticality of requirements (a combination of risk and value). This implies that high-risk/high-value requirements constitute critical requirements and are implemented during initial iterations. Low-risk/low-value requirements are left for last.

2. Based on stakehoders' opinions. This implies that stakeholders determine which requirements should be implemented initially and which should be left for last.

3.6 Rapid development model

The rapid application development (RAD) model is a variation of the iterative and incremental development model that emphasizes very short development cycles. The model assumes that requirements are well understood.

3.7 Agile development model

Development methods tend to become "heavy-weight" when certain characteristics are present, such as heavy regulations, delay for the production of an executable, lack or low degree of adaptation, and infrequent stakeholder feedback. As an alternative to this, "light-weight" (or agile) methods have been proposed.

Agile methods are a variation of the iterative and incremental development model, advocating short iterations. In agile development, tests are written before the functionality is coded. Further, an agile team will contain a customer representative (appointed by stakeholders) to participate during iterations in order to answer questions. At the end of each iteration, stakeholders and the customer representative review progress. Several instances of the agile model exist, perhaps the most popular one being *extreme programming* (XP). Extreme programming advocates programming in pairs, extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, and frequent communication with stakeholders.

3.8 Cleanroom software engineering techniques

Borrowing its name from the semiconductor manufacturing industry, "cleanroom" software engineering focuses on the prevention of defects (as opposed to defect removal). To achieve this, one aims to write code correctly the first time rather than identifying and remove bugs once the code is written. Cleanroom

development adopts the iterative and incremental development model deploying formal methods and quality control and it follows a top-down design with stepwise refinement, where the entire program is viewed as a black-box. Functional decomposition produces formal descriptions of modules as black-boxes. These descriptions are normally mathematical functions. Once code is written for a module, it is verified against its intended behavior. The cleanroom model explicitly addresses risks by deploying formal methods to eliminate the possible introduction of errors.

3.9 Code-and-fix

As its name suggests, code-and-fix involves writing code and correct possible errors. Strictly speaking this is not a development process model, but rather an infomal description on how software can be written. The model does provide feedback but only in an informal manner.

On one hand this model has no overhead as it does not entail any activities other than implementation. This can perhaps be advantageous for very small projects. However, for any project of substantial size, the model is not viable as it does not include solid engineering principles (such as planning, analysis, design, quality control).

3.10 A framework for software development: The Unified Software Development Process (UP)

Strictly speaking, the Unified Software Development Process (or Unified Process, or just UP) is not a development model, but rather a framework for software development, based on the iterative and incremental development model, where iterations are grouped into the following four *phases*:

1. Inception. The goals of this phase include establishing a justification (business case) and the scope for the project, outlining the use cases and key requirements, outlining candidate architectures, define risks, and prepare a preliminary schedule and cost estimate. The inception phase should provide a justification on the feasibility of the project.

2. Elaboration. Implements the core architecture through the implementation of critical use-cases. Critical use cases contain high-risk and high-value requirements and the UP addresses them during the Elaboration phase. The motivation is to drive down the risk of the entire development so that if the project is going to fail it will do so early, not late.

3. Construction. Implements secondary (non-critical) use cases. Testing during Construction is referred to as *alpha testing*. Construction is complete when the system is ready for operational deployment and all supporting artifacts are complete (user guides, etc.)

4. Transition. Pleases the system into production use. Testing during Transition is referred to as *beta testing*.

With the possible exception of the inception phase, all other phases contain possibly several iterations. Even though an iteration includes work in most (if not all) disciplines, the relative effort and emphasis change over time. Early iterations tend to apply greater emphasis to requirements and design, and later ones tend to put more emphasis on implementation. A picture that illustrates the four phases (broken down into several iterations) and the activities involved in each iteration together with an indicative workload in each activity is shown in Figure 3.3.

Two notions are vital to the whole concept of UP: Timeboxing and milestones.

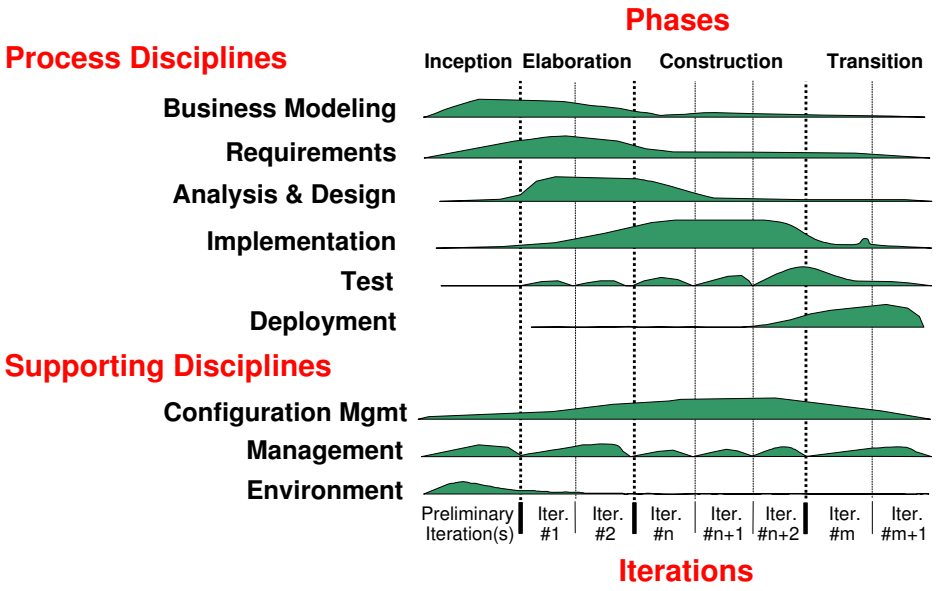

Figure 3.3: The Unified Software Development Process (UP).

Timeboxing

The notion of *timeboxing* is central to the UP. The UP recommends short and fixed ("timeboxed") iteration lengths to allow for rapid feedback and adaptation. The idea is that long iterations increase project risk. Moreover, if meeting a deadline seems to be difficult, UP recommends to remove some tasks (or requirements) from the current iteration and include them in a future iteration.

The main motivation behind timeboxing is that the development team is forced to focus and produce a tested executable partial system on-time. The team must prioritize requirements and tasks based on business and technical value. Furthermore, complete mini projects can build confidence both for the team as well as for the stakeholders.

Milestones

Each iteration concludes with a well-defined milestone, a point at which a review provides an assessment of how well the objectives of the iteration have been met, and if not, how the project can be brought back to track. The end

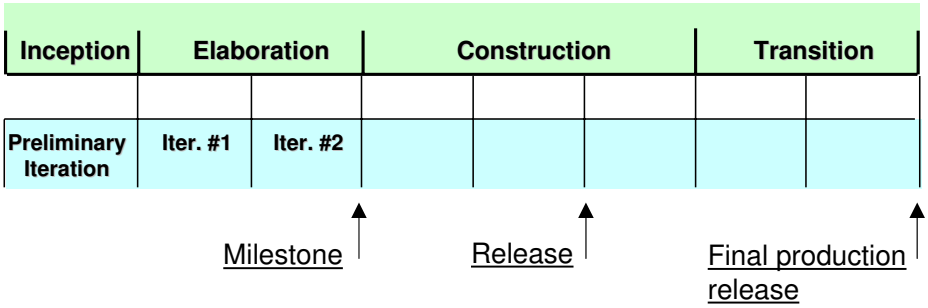

Figure 3.4: Milestones in the Unified process.

of each iteration produces a minor release, which is a stable executable subset of the final product (see Figure 3.4). We can also distinguish between minor and major milestones. The latter would signify the end of a phase.

It is also important to note that occasionally an iteration may not necessarily add new requirements, but it may modify existing requirements (or even drop existing requirements). Naturally, feedback from stakeholders in the form of requests (for additions, modifications or deletions) cannot be unconditionally implemented but it is taken into consideration by the development team which now has to decide on the feasibility of the request.

The concept of iterative development can be applicable to maintenance as well. Any form of maintenance can be addressed iteratively.

Advantages of the UP

We can identify three main advantages of the Unified Process:

1. Risks are identified early.
2. Handle evolving requirements (through feedback and adaptation).
3. Attain early understanding of the system.

How to get it wrong in UP

Consider the following statement: "During the Inception phase all requirements are defined and frozen. The purpose of the Elaboration phase is to

produce a design of the system. Construction, as its name suggests, builds the system." What is wrong with this statement? The answer is that the statement misinterprets the Unified Process and superimposes it over the linear model. Recall that the Inception phase defines the scope of the project, the Elaboration phase builds the core architecture of the system and the Construction phase builds the remaining features. Iterations in all phases contain work in most (if not all) disciplines (activities) in varying degrees of workload.

3.11 Examples

Example 3.1. In the literature it is claimed that for medium- to large-scale systems, a linear software development process "is not realistic." Briefly explain what it meant by "not realistic" and describe two problems usually associated with the linear ("waterfall") process model.

Solution:

As requirements tend to change, it is usually not feasible to fully complete and "freeze" the requirements specification, then fully complete the design, build and then test the software. Two associated problems with the linear model are:

- Risks are not addressed early but can be encountered late in the development.
- Lack of flexibility of changing requirements.

Example 3.2. Briefly describe what is meant by "timeboxing" and what is the motivation behind short iterations.

Solution:

Timeboxing is a term that refers to the fixed length iterations. Long iterations increase project risk. On the other hand, short length iterations allow the rapid feedback and adaptation.

Example 3.3. Consider and comment on the following quote: "During the Inception phase all requirements are defined. The purpose of the Elaboration phase is to produce a design of the system. Construction, as its name suggests, builds the system."

Solution:

The above quote misinterprets the Unified Process. The Inception phase defines the scope of the project, the Elaboration phase builds the core architecture of the system and the Construction phase builds the remaining features. Iterations in all phases contain work in most disciplines.

Example 3.4. Provide brief definitions for the two broad categories of requirements. In the context of the Unified Software Development Process (UP), briefly describe when requirements are captured, and which artifacts are used to capture them.

Solution:

Functional requirements deal with the observable behavior of the system. In the UP functional requirements can be captured in use case scenarios as they describe the interactions between end-users (actors) and the system. Nonfunctional requirements describe constraints and qualities of the system. NFRs are normally captured in the Use Case Model and in the Supplementary Specification. Both artifacts start during the preliminary iteration (of the Inception phase) and can be refined in subsequent iterations.

Example 3.5. Briefly explain what is meant by "critical use cases" within the context of the Unified Software Development Process (UP) and what is the motivation behind this notion.

Solution:

Critical use cases contain high-risk and high-value requirements and the UP addresses them during the Elaboration phase. The motivation is to drive down the risk of the entire development so that if the project is going to fail it will do so early, not late.

Example 3.6. Consider the Unified Software Development Process (UP). Briefly explain what is meant by the term "artifacts" and describe the two dimensions over which we can view artifacts. What is the relation between artifacts and deliverables.

Solution:

The term "artifact" is used to describe the documentation of an activity. We have two dimensions of artifacts. Horizontal dimension: the set of artifacts (across all disciplines) over a single iteration, and Vertical dimension: the set of artifacts of one discipline (activity) throughout the entire development (throughout all phases). A deliverable is a document that describes sets of artifacts by the end of an iteration.

Example 3.7. List the phases of the linear model. Where do these phases get their names from? How does this model relate to the phases of the Unified Software Development Process (UP)?

Solution:

The names of the phases of the linear model are: Requirements, Analysis, Design, Implementation, and Testing. The names refer to the activities entailed in each phase. On the contrary, the phases of the UP contain all of the activities (within iterations) in various workloads.

Example 3.8. Briefly describe three advantages of an iterative process.

Solution:

Three advantages of an iterative process can be the following:

- Risks are identified early.
- Handling evolving requirements (through feedback and adaptation).
- Attaining early understanding of the system.

Example 3.9. Briefly describe the primary objectives of the Elaboration phase of the Unified Software Development process. In your answer take into consideration the outcome of the previous phase.

Solution:

The primary objectives of the Elaboration phase are:

- To produce an executable core of the system
- To address the critical use cases identified in the Inception phase.

Example 3.10. Consider the case where as a project manager you need to make a decision on the type of process your team should follow for the development of a large-scale distributed and concurrent software system (to handle marketing and sales) which is expected to aid a company to become competitive in a dynamic environment. You may assume that stakeholders are confident about all their goals well in advance. Briefly describe what your decision will be and what factors have determined your decision.

Solution:

Despite the fact that all requirements are currently known and you may be tempted to consider them as frozen, there is still no guarantee that they will remain frozen throughout development. This is due to the dynamic nature of the operating environment. We will follow an iterative development process and factors which determine this decision are as follows:

- Complexity of the system.
- Risks involved.
- (Potentially) changing requirements.

Example 3.11. Briefly describe two common factors between the Unified Software Development Process (UP) and Extreme Programming (XP) as well as two factors which are unique to each one.

Solution:

Common factors are:

- Both of them has an iterative and incremental development process.
- They both are getting feedbacks from developers and stakeholders.

Differences are:

In UP:

- We have time-boxed iterations.
- Critical use cases are handled first (in Elaboration phase).

In XP:

- Stakeholders define stories to be addressed.
- Encourages test-first development.

Example 3.12. Various iterative development processes like the Unified Software Development process (UP) recommend that critical requirements should be addressed in early iterations. One may argue that this does not make much sense since all requirements must be implemented anyhow and all requirements are important. Respond to this argument by briefly describing what is meant by the term "critical requirements" and provide a rationale for their early implementation.

Solution:

The term "critical" (in the context of software requirements) is not a synonym to "important." In this context, "critical" refers to those requirements which provide high-value and pose high risk. One reason they have to be implemented in early iterations is to not delay risk and is to drive down the risk of the entire development so that if the project is going to fail it will do so early, not late.

Example 3.13. Briefly describe what is meant by "iteration", by including any and all (engineering) activities. What management activities would you include in an iteration, and how do these relate to the engineering activities? What is the end result of an iteration? What are the benefits and obligations of stakeholders at the end of an iteration?

Solution:

An iteration is a mini-project: it includes its own set of requirements elicitation and analysis, (object-oriented) analysis and design, mapping from design to implementation, testing (and integration).

Further, an iteration would include activity planning and risk assessment in the beginning and risk control throughout all the activities. The end result is a deliverable which consists of a set of artifacts (such as design model, source code etc.) and a tested executable.

Stakeholders would be interested as they would want to attain an understanding over the project, and assess the executable against their requirements. These constitute their benefits. Their obligations are implied here through their engagement in the provision of feedback.

Example 3.14. Consider use cases:

1. In the context of iterative development, what makes a use case *critical*?

2. In the context of the unified software development process (UP), during which phase are critical requirements addressed and what is the rationale behind this decision?

3. Who are the parties involved in a use case?

4. How are each party's benefits guaranteed and how are its obligations specified?

5. Regardless of the development process model followed, how can use cases be deployed to participate in requirements verification?

Solution:

1. Critical use cases contain high-risk and high-value requirements.

2. In the UP critical use cases are addressed during the Elaboration phase. The motivation is to drive down the risk of the entire development so that if the project is going to fail it will do so early, not late.

3. The parties in a use case are actor(s) and the system seen as a black box.

4. The actor must guarantee preconditions; benefits from postconditions, and the system must guarantee postconditions and benefits from preconditions.

5. The executable can be demonstrated against the use case scenarios in order to answer "have we built the product right."

Example 3.15. In the literature we frequently read of *iterative development.*

1. Provide a brief description of what is meant by "iteration" and briefly describe how one can deploy an iterative model to achieve incremental development.

2. What is the outcome of an iteration (in terms of a deliverable) and how does this outcome relate to the overall system under development.

3. What are the benefits and obligations of stakeholders at the end of an iteration?

Solution:

1. An iteration is a short fixed-length mini-project: it includes its own set of requirements elicitation and analysis, (object-oriented) analysis and design, mapping from design to implementation, testing (and integration). One can adopt iterative development by organizing a series of (normally

short and fixed-length) iterations, each one addressing new requirements. As a result, the system can grows incrementally over time, iteration by iteration.

2. The outcome of an iteration is a deliverable which consists of a set of artifacts (such as design model, source code etc.) together with a tested, integrated and executable system which is a subset of the overall system under development.

3. Benefits of stakeholders include an understanding over the project, and an assessment of the executable against their requirements. Their obligations are implied here through their engagement in the provision of feedback.

3.12 References

1. Kruchten, P., The Rational Unified Process: An Introduction, 3rd edition, Addison-Wesley Professional, 2003.

2. Royce, W. Software Project Management: A Unified Framework, 1st edition, Addison-Wesley Professional, 1998.

Chapter 4

Configuration management

A software system is normaly being developed by several people. It is not uncommon that different teams of developers are situated in different physical and geographical locations. Furthermore, during development certain needs may arise: First, more than one version of the software may have to coexist and maintained. Reasons or this include the fact that different custom configurations may have to be supported, or different releases may be in use. Second, a given version would normally have to function under various operating environments such as hardware configurations, platforms, and operating systems.

Software configuration management (SCM) is the task of managing (tracking and controlling) changes in the evolution of a software system. This definition does not confine itself to code artifacts, but it applies to the entire collection of artifacts. Furthermore, the definition does not confine itself to a single point in time, nor does it apply exclusively to the development phase. Configuration management is applied at predefined points of time throughout the lifecycle of a software system. The ISO/IEC 12207 standard requires that a configuration management plan should be established to include a description of all relevant procedures, schedules and responsibilities.

As a motivation for configuration management, we can consider some simple scenarios:

1. A developer wants to see the most recent version of some component and its change history over the last 10 days.

2. In trying to optimize performance, a developer is trying a new algorithm in a component which eventually does not bring any benefits. He decides to roll back to the previous version.

3. A developer is preparing a release of the project. He needs to know what items to include and in which version.

4. In the most recent release of the project, a bug was detected. The manager wants to track what changes caused the bug, who made those changes and when.

Configuration management is a collective term that is used to describe four related activites: *change management*, *version management*, *system building*, and *release management*. These activities are discussed in the subsequent sections.

4.1 Change management

As change is a central activity during development and maintenance, the task of change management is to keep track of requests for change, to perform change impact analysis and to record any and all changes.

4.2 System building

System building is the task of compiling and linking system components, configuration files and external libraries in order to create an executable system.

4.3 Version management

The task of version management (also known as *revision control*) is to keep track of different versions of software components as well as the systems and platforms that these components are deployed. We need to establish some terminology:

- A *version* is an initial distribution of an item associated with a complete compilation. Different versions have different functionality.

- A *revision* is a change to a version that corrects errors, but does not affect the functionality of the version.

- The set of versions of a software component (and other configuration items in which the component depends, such as libraries) is referred to as a *codeline*.

- The collection of component versions that make up a system is referred to as a *baseline*. A baseline can mark a milestone during development or maintenance. Version management can thus be viewed as managing codelines and baselines. A *software configuration audit* is a procedure to ensure the completeness and correctness of a baseline. A sequence of baselines representing different versions of the system is referred to as a *mainline*.

- A version of the system that is made available only to other developers is called a *promotion*. A *master directory* is a library of promotions.

- Versions that are intented to coexist are called *variants*. Variants are versions that are functionally equivalent but are designed for different hardware and software operating environments.

- A *branch* is a sequence of versions along the time line.

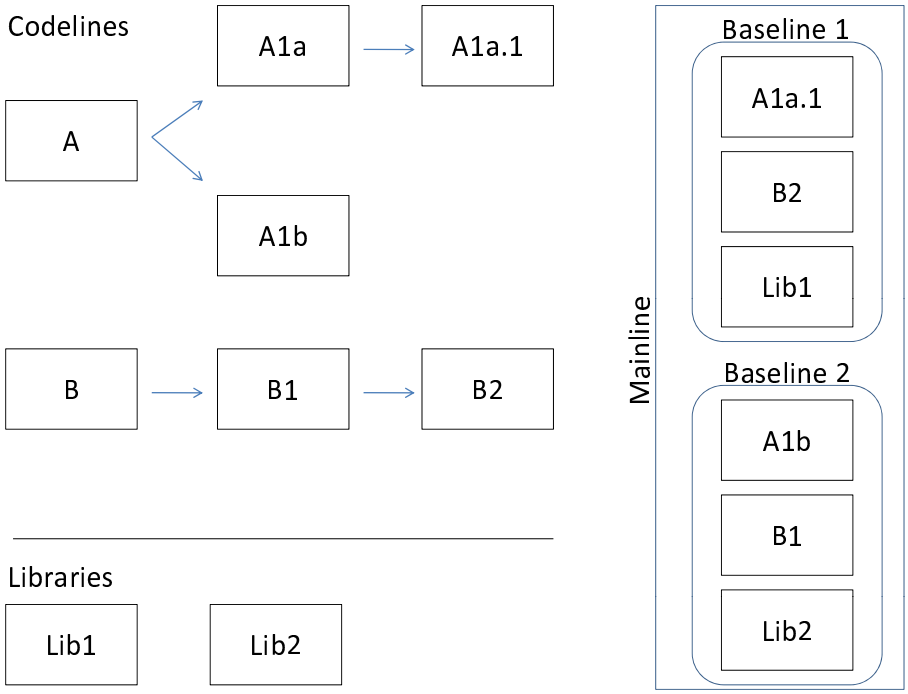

Figure 4.1: Formation of baselines with codelines.

- A *release* is the formal distribution of an approved version. A *repository* is a library of releases.

Example 4.1. Figure 4.1 illustrates the codelines for components `A` and `B` and the existence of two external libraries. Versions `A1a` and `A1b` form two variants. Two baselines are formed, each utilizing different codelines and libraries. The two baselines form a mainline.

4.3.1 Workspaces

Developers can work in an isolated environment without interfering other developers. Such isolated environment is referred to as a *workspace*. Developers can obtain a version into their workspace (*checkout*), or commit changes to the repository (*checkin*).

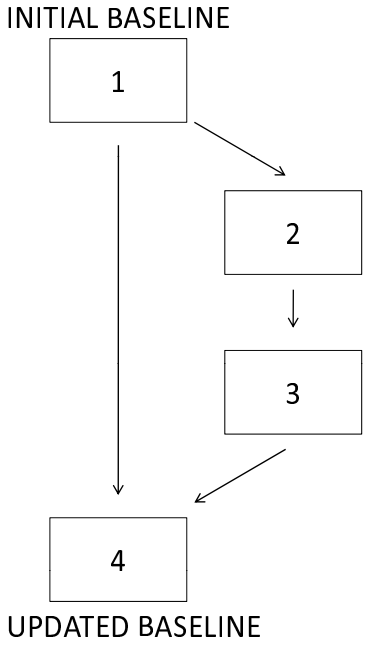

Figure 4.2: History tree.

4.3.2 Version history tree

We can visualize version management with a *version history tree*. Vertices in the tree illustrate baselines or codelines and edges illustrate activities, such as creating a branch which is a duplicated file (can be a source code file or an entire directory) that can allow parallel development, or merging branches into baselines.

Example 4.2. As an example, consider two developers, Jack and Jill, working on the same project (see Figure 4.2). The initial baseline is illustrated as a node (1) in the version history tree of the project. While Jack is working on the initial baseline, Jill is assined to work on some directory in parallel with Jack. Jill's directory is illustrated as a node (2) which defines a branch. Jill will develop a codeline for this directory illustrated by nodes (2)-(3). Upon completion of her task, Jill has to merge her branch into the baseline and address possible conflicts. This results in a new baseline (node 4).

4.3.3 Benefits of version management

We can identify a number of benefits of version management. The procedure maintains project/ file history and allows the accessing of previous versions. It ensures that every developer uses the correct version of an object (file, or directory). In fact, developers have the option to roll back to the most recent baseline, should something fails.

4.3.4 Concurrent access: File locking and merging

Some method of managing concurrent access to a file is required in order to maintain the integrity of data. This problem is addressed by one of two different models: File locking and vesion merging. File locking views developers are readers and writers. Readers with to obtain access without modifying the file, and writers need to obtain access possibly in order to modify the file. Writers operate in self exclusion, i.e. the file is made available ("checked out") to only one writer at a time. During checkout, possibly several readers may obtain access to the file. Additionally, subsequent writers would have to wait until the current writer "checks in" the updated version of the file (or cancels the checkout). Merging methods deploy sophisticated algorithms to allow multiple writers to access the file at the same time. The first writer to check in changes to the central repository always succeeds. Further changes to the file are merged into the central repository while preserving the changes from the first writer. While file locking can be fully automated, merging normally requires human intervention.

4.4 Release management

A *system release* is a version of the software system that is made available to stakeholders. We can distinguish between a *major release* which contains significantly new functionality and a *minor release* which does not contain

significantly new functionality but has repaired bugs and fixed problems.

A version which is selected for release must be validated (by a designated team) to ensure that it is complete and consistent. This task is referred to as *auditing*. A release can be made available as source code (together with build and installation instructions), as an executable for a given platform, or as an (automatic or manual) update.

Chapter 5

Software quality

5.1 Introduction

5.2 A review of the ISO/IEC 9126-1 international standard

ISO 9126-1, classifies software quality into six characteristics (functionality, reliability, usability, efficiency, maintainability and portability), which are further subdivided into subcharacteristics (See Figure 5.1).

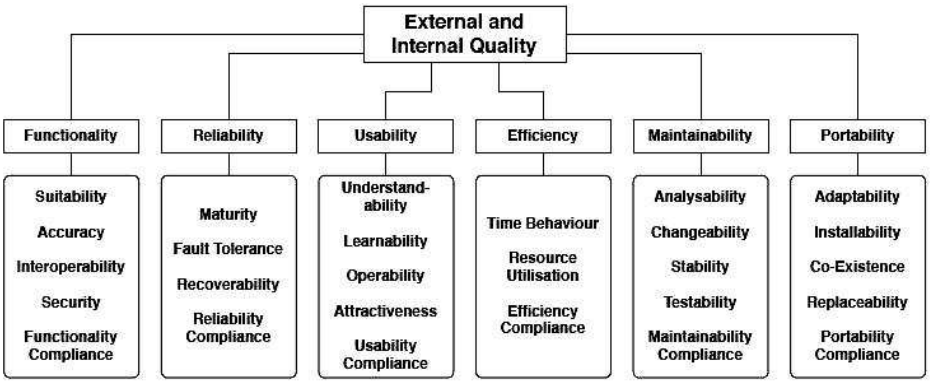

Figure 5.1: Classification in the ISO-9126 standard.

5.2.1 External and internal quality requirements

The standard classifies quality attributes into external and internal:

External quality requirements specify the required level of quality from the point of view of the users of the software.

Internal quality requirements specify the required level of quality from the point of view of the developers (or maintainers) of the software.

Functionality

Functionality is an attribute which deals with the degree to which a software system satisfies user needs. We can distinguish between functionality and the rest of the attributes (hence our distinction between functional and nonfunctional requirements) since it is concerned with what the software does in order to fulfill users needs, whereas the other characteristics are mainly concerned with when and how the software fulfills needs. Its subcharacteristics include *suitability*, *accuracy*, *interoperability*, and *security*.

Reliability

Reliability is a property of the software which reflects the degree to which the software can maintain its level of performance when used under specified conditions. Limitations in reliability are due to faults in development. Its subcharacteristics include *maturity*, *fault tolerance*, and *recoverability*.

Usability

Usability is a property of the software which reflects the degree of ease of use as well as the capability to be understood. Its subcharacteristics include *understandability*, *learnability*, and *operability*.

Efficiency

Efficiency is a property of the software which reflects the degree to which the software system provides performance relative to the physical resources of the system. Its subcharacteristics include *time behavior*, *resource behavior*, and *resource utilization*.

Maintainability

Maintainability is a property of the software which reflects the degree to which the software can undergo any type of maintenance (corrective, preventive, perfective and adaptive). Its subcharacteristics include *analyzability*, *changeability*, *stability*, and *testability*.

Portability

Portability is a property of the software which reflects the degree to which the system can be ported to a new (hardware or software) environment. Its subcharacteristics include *adaptability*, *installability*, *coexistence*, and *replaceability*.

5.3 Tradeoffs between quality attributes

Very often, individual quality attributes may conflict with each other and project managers must work with stakeholders to provide tradeoffs. Example conflicts may include the following:

- Integrity vs Efficiency: Control of access requires additional code and processing.
- Usability vs Efficiency: Improvements in HCI may increase the amount of required code and power.
- Maintainability and Testability vs Efficiency: Well-structured code is easy to test but less efficient.

- Portability vs Efficiency: Optimized software leads to a decrease in portability.
- Flexibility, Reusability & Interoperability vs Efficiency: Reusable code and code which is designed to be easily changed can hardly be optimized.

5.4 Errors and defects

Usually the terms *bug*, *error*, and *defect* are used as synonyms to refer to a quality problem. Sometimes we can distinguish between errors and defects based on the following distinction:

Error: Problem found before the software is deployed.

Defect: Problem found after the software is deployed.

The distinction is made because of the economic and business impact of a problem. As we move from internal to external failure costs, the relative costs to detect and repair a problem increases.

5.5 Classification of the causes of software errors

We can classify the various causes of software errors, and grouping related causes as follows:

Requirements (Faulty definition of requirements; Client-developer communication failures; Deliberate deviations from requirements). The Carnegie Mellon Software Engineering Institute reports that at least 42-50 percent of software defects originate in the requirements phase.[1] The Defense Acquisition University Program Manager Magazine reports on a U.S.

[1]Carnegie Mellon Software Engineering Institute, The Business Case for Requirements Engineering, RE 2003, 12 September 2003.

Department of Defense study that over 50 percent of all software errors originate in the requirements phase.[2]

Logical design errors. The design should be a faithful map of the (validated) requirements. Reviews of design artifacts produced should be part of the process.

Coding (Coding errors; Non-compliance with documentation and coding instructions; Shortcomings in testing). Usually, programmers resent the procedure of having their code reviewed by others, even though in other disciplines this is a standard practice. The objective of code reviews is the discovery of errors. Any code produced in the process may be reviewed. Review teams should be relatively small and reviews should be fairly short.

5.6 Faults and failures

When an error causes improper functioning of the software, it becomes a *fault*. However, not all errors in software are faults. Once activated, a fault becomes a *failure*. In some situations, a fault is never activated during the execution of a given scenario due to the fact that the combination of conditions necessary to activate it never occurs. Such faults are called *dormant* or *latent* faults. The following causal relation exists between errors, faults and failures:

$$errors \rightarrow faults \rightarrow failures$$

The relationship is not necessarily one-to-one, as a single error may cause more than one fault. Additionally, more than one errors may be combined to cause a single fault. Similarly a single fault may cause one or more failures, or more than one fauls may be combined to cause a single failure.

[2]Defense Acquisition University Program Manager Magazine, Nov-Dec 1999, Curing the Software Requirements and Cost Estimating Blues.

Failures can be from merely annoying to catastrophic. Further, one failure can be corrected within seconds, another within weeks or months. We can classify failure as follows:

Transient Failure that occurs only with certain input.

Permanent Failure that occurs with any inputs

Recoverable This is the type of failure that allows the system to recover without manual help, as opposed to unrecoverable.

Corrupting This is the type of failure that has alters system state or data, as opposed to non-corrupting.

The case where an ATM machine cannot read the magnetic stripe on a given (undamaged) card is an example of a transient and non-corrupting failure.

5.7 Examples

Example 5.1. 1. Studies show that the relative cost to fixing errors increases disproportionately as one moves throughout the activities of software development. Briefly provide an explanation to this fact.

2. What is the difference between an 'error' and a 'defect' and why is it important to distinguish between them?

Solution:

1. Fixing an error would entail modifications to several artifacts from potentially several activities until we reach the origin of the error. As we move from one activity to another, this tracing activity becomes more expensive.

2. Both refer to quality problems. Errors are those problems found before deployment and defects are those found after deployment. The motivation behind this distinction is the impact each category would have.

Example 5.2. 1. Studies show that the relative cost to fixing errors increases disproportionately as one moves along the stages of software development. Briefly provide an explanation to this fact.

2. What is the difference between an "error" and a "defect" and why is it important to distinguish between them?

Solution:

1. Fixing an error would entail modifications to several artifacts from potentially several activities until we reach the origin of the error and all places where it has propagated. As we move from one activity to another, this tracing activity becomes more expensive.

2. Both "error" and 'defect" refer to quality problems. Errors are those problems found before deployment and defects are those found after deployment. The motivation behind this distinction is the impact each category would have on the overall business and the costs involved.

Chapter 6

Risk management

6.1 Introduction

We can informally define *risk* as "the chance of exposure to the harmful consequences of (future) events." Put it differently, a risk is a possibility that some abnormal situation will occur.

We are dealing with risks in our everyday lives. For example, we are using email knowing that they may be attacked by viruses. We maintain electronic data on hard disks knowing that they are subjected to physical deterioration or even virus attacks and thus there is always the risk of disk crash.

We can use the above definition to distinguish between risk and *problem* (or *issue*). If a situation has already occurred or it is certain that it will occur, then the situation is not a risk but a problem.

For example, if you now that a member of your team will be out of town and unable to be reached and work on the project over a certain period of time, then this is a problem which you have to manage beforehand (not after it happens).

When a risk happens, it then becomes a problem.

6.2 Risk management

Our project plans contain a number of assumptions. We need to plan and control situations where these assumptions turn out to be wrong. For example, we may assume that the integrity of a database will not be violated (intentionally or unintentionally). Furthermore, some things which seem obvious when the project is over might not have been obvious during project planning. The definition of risk includes a number of key elements associated with risk items: First, a risk item relates to the future. Second, a risk item involves speculating about future events and third, a risk item involves cause and effect: For example, poor communication with stakeholders will likely result in an unclear scope of the project.

In project management, we are concerned with the identification of risks and the preparation of plans to minimize their effect on four attributes: project, process, product and business.

Project risks affect schedule, resources, or costs.

Process risks affect the underlying development process.

Product risks affect the quality (such as performance) of the software being developed.

Business risks affect the organization developing the software.

The risk management process involves two steps:

1. Risk assessment: The things we do as we plan the project.
2. Risk control: The things we do during the implementation of the project.

6.2.1 Risk assessment

The assessment of risk involves identification, analysis and prioritization of risk items.

Risk identification: Identify risks related to the overall project, to the product and to the business. The outcome of this step is a collection of risk items.

Risk analysis: Assess the likelihood of occurrence and the impact of each risk item identified above. Which ones do we consider really serious?

Risk prioritization: Naturally, not all risk items are equally serious. Certain items would call for our immediate attention, and other can be given low priority.

6.2.2 Risk control

The control of risk involves planning and mitigation, resolution and monitoring of risk items.

Risk planning and mitigation: We can draw plans either to avoid a risk item (i.e. to minimize or eliminate the likelihood of occurrence), or to minimize its impact on the project, or product, or business. In doing that we also have to draw contingency actions: What do we have to do should a risk item (despite our plans for mitigation) has occurred?

Resolution: The assignment of a risk item to a person/ date by which it has to be resolved.

Risk monitoring: We assess our plan for mitigation and contingency. Has the plan worked? We have to monitor risks throughout the project.

When to perform risk management Note that risk assessment and risk control work together. We can perform identification, analysis, prioritization, planning and mitigation at the beginning of a development iteration, and we can perform resolution and monitoring throughout the iteration.

6.3 Risk assessment

In the next subsections we will discuss all steps involved in risk assessment.

6.3.1 Risk identification

How can we identify possible risk items? If present, experience of past projects can be useful through the creation of checklists. if experience from past projects is not present, we can still brainstorm knowledgeable stakeholders to pool together lists of concerns. Risk items and concerns of stakeholders should never be kept secret. Stakeholders should be fully informed on potential risks.

Another approach for risk identification can be causal mapping: the identification of possible chains of cause and effect. For example, we can indicate the following cause-effect pair: "*The project may be late* due to *employee turnover*." The sentence clearly specifies both cause and effect.

We can identify certain types of risks that are associated with software projects:

Estimation risks

- Schedule slip(s): Not being able to meet a deadline either for a minor milestone or for a major milestone where a deliverable should be made available to stakeholders. This can be due to a number of reasons, including underestimating the time or effort needed, or bad (or lack of) process procedures. As a result, a software may be delivered late, or not adequately tested, or even with less functionality than expected.

Organizational risks

- Procedural risks. This can be a lack of clear assignment of roles and responsibilities, or conflicting priorities.

- Project cancelation: Stakeholders may decide to cancel the project. This can be due to a number of factors, including budget cuts, loss of confidence, high defect rate, or the inability of software udner development to meet changing requirements.

- Unanticipated financial problems result in budget cuts.

Technology risks

- Lack of adequate testing.
- Reuse. For example, consider a component which we planned to reuse which contains certain defects, or some technology which we have to adopt but we do not have enough experience with, or a database management system which does not fully support a major requirement such as concurrency.
- Poor supporting (hardware/software) environment. This can be because hardware or supporting software (operating system, tools, etc.) do not perform as expected, or change during the progress of development. Another example is that computer-aided software engineering (CASE) tools may not be inter-compatible and cannot be integrated, or may produce inefficient models (e.g. inefficient code generation).
- Maintenance. Experience shows that maintenance tends to consume a large proportion of the software life cycle. A software system should be built with the ability to change and evolve. Furthermore, lack of vendor support after system delivery is a risk.

People risks

- Key staff may be unavailable at times when they are mostly needed, or being impossible to find and recruit staff with skills required.
- Lack of skill of current staff. Also, lack of (adequate) training if required. It is not the case that a good project manager can manage any project. Software project managers need domain knowledge and experience, as well as knowledge on the underlying process.
- Poor communication (including lack of good collaboration) between members of development team, management and stakeholders. This includes

lack of committment, as well as lack of stakeholders' feedback when required. These points can be factors of interpersonal conflicts.

- Staff turnover: This can be a problem if no artifacts document the architectural and design rationale of the project. As a result, the learning curve for new staff members will be steep. Even in cases where development artifacts do exist, staff members need time to learn the system under development.

Requirements risks

- Requirements can be incomplete, inaccurate, vague, or conflicting. As development progresses to later stages, the cost of disambiguating or resolving conflicting requirements gets significantly higher. Furthermore, there may be a lack of agreement on the requirements or on the vision of the project, or a failure to assign priorities to requirements. These risks are due to bad analysis of requirements.

- Impossible or unrealistic requirements. The feasibility of requirements should be discussed and agreed upon during analysis.

- Stakeholders may fail to realize the impact of requests for changing requirements.

- "Inflation" of requirements: As development progresses, more and more requirements that were not identified at the beginning of the project tend to emerge, or current requirements tend to constantly change. This can have an impact on effort, time and cost. For example, major changes in requirements may either not be feasible or may cause major delays to the project.

- "Gold-plating" of requirements. This refers to the addition of unecessary features, perhaps in order to show off one's skills, or just because

developers may think that "they know what stakeholders want or should need."

Of $35.7 billion spent by the US DoD for software,[1] only 2% of the software was able to be used as delivered. The vast majority, 75 percent, of the software was either never used or was cancelled prior to delivery. The remaining 23 percent of the software was used following modification. A similar study conducted by the Standish Group on non-DoD software projects in 1994[2] produced very similar results: In over 8,000 projects conducted by 350 companies, only 16% of the projects were considered successful. Success in this study was considered software delivered on time and within budget. Further research by the Standish Group indicates the following major reasons for the high failure rate in software development:

- Poor requirements.
- Lack of understanding that cost and schedule are engineering parameters based on requirements.
- Lack of understanding and following a process and a life cycle.

The above studies indicate that the way we define, analyze, and manage requirements is imposing serious risk to the success of our software projects.

Requirements elicitation and analysis is an iterative activity. It is not enough to obtain the stakeholders requirements once and assume that they are correct, complete and non-conflicting (*"But We Gave You Exactly What You Asked For!"*).

[1] Proceedings of the 5th Annual Joint Aerospace Weapons Systems Support, Sensors, and Simulation Symposium (JAWS S3), 1999.

[2] K. Pradip and M. Bailey, *Characteristics of good requirements*, INCOSE Symposium, 1996.

We are also inviting schedule relays and cost overruns if we are not involving project stakeholders throughout the development effort (*"I Think I Know What Your Requirements Are!"*).

Furthermore, it is wrong to assume that all stakeholders "speak the same language" when they talk about their goals and needs. Some stakeholders may have vague goals, or conflicting requirements. It is our responsibility, during requirements analysis, to disambiguate and resolve all possible conflicts. We should also not expect all stakeholders to be technically oriented people. We can refer to this as "overboard assumptions."

We need to take into consideration that if the expectations of stakeholders are not met, future services will not be requested.

It is unrealistic to assume that software requirements are not going to change during the development process. The risk of requirements changing is that the changes will spiral out of control and prevent an application from ever being completed.

There are times when stakeholders assume they have the luxury of having developers perform reengineering tasks over and over until they get something they like. A further risk with this attitude is that the software project is not perceived to belong to the stakeholders (*"I Don't Know What I Want, but I'll Know It When I See It"*). One measure to address this problem is with *prototyping*. A *prototype* is an incomplete version of the software system being developed which typically simulates a few aspects of the features of the final product.

Unavoidable risks

- Unavoidable risks are those that cannot be controlled, such as changes in legislation (e.g. taxation, privacy issues), changes in standards or regulations, etc.

6.3.2 Risk analysis

The first step in risk analysis is to make each risk item more specific. In doing that, we need to specify, for each risk item its likelihood of occurrence and its impact on project, product and business.

How do we build scales to express likelihood and impact?

To describe likelihood, in the case where numerical data is present (perhaps though historical data) we can speak in terms of probability of occurrence, thus building a quantitative model. Where a quantitative model cannot be made available, then we rely on a qualitative model, by specifying qualitative descriptors in terms of level of likelihood such as High, Significant, moderate and Low.

A quantitative model can be built to describe impact, where data is available. Alternatively, a qualitative model can be build by specifying qualitative descriptors in terms of level of impact such as for example High, Significant, Moderate and Low.

Identify contributing factors

Many risks can occur in several ways (from several causes). If we are not careful, we will only be looking for one of the ways. We need to get to the actual causes.

6.3.3 Risk prioritization

Once risk items have been identified and analyzed (in terms of their likelihood of occurrence and impact), we need to set priorities in order to determine where to focus risk mitigation efforts. Furthermore, some of the risk items may be unlikely to occur, and others may not be serious enough to raise any concern.

To determine the priority of each risk item in a quantitative model, we combine the two values, likelihood and impact. This priority scheme helps

push the big risks to the top of the list and the small risks to the bottom.

Risk exposure For a quantitative model, we calculate the product of impact of the risk item and its probability of occurrence. This product is referred to as *risk exposure* (or *composite risk index*). Once the risk exposure is calculated, risk items can be ranked.

Naturally, both factors can be difficult to estimate. The probability of risk occurrence is difficult to estimate if no past data on frequencies are not readily available. Similarly, the impact of the risk can be difficult to estimate since the potential (financial) loss in the event of risk occurrence may be difficult to determine. Furthermore, both factors can change in magnitude depending on the adequacy of risk avoidance and prevention measures taken and due to changes in the external business environment. One limitation with the calculation of risk exposure is that it assumes that the amount of damage sustained will always be the same. Hence it is absolutely necessary to periodically re-assess risks and intensify/relax mitigation measures as necessary.

Example: Consider a fire which can cause 0.5 millions of damage in a facility. Let the probability of this event be 0.01. Then, the risk exposure is

$$RE = \$0.5m \times 0.01 = \$5,000$$

which would be analogous to the amount needed to purchase insurance.

For a qualitative model, we can tabulate the two factors (See Figure 6.1 and identify areas in the table that signify various risk severity levels).

6.4 Risk control

In the next subsections we will discuss all steps involved in risk control.

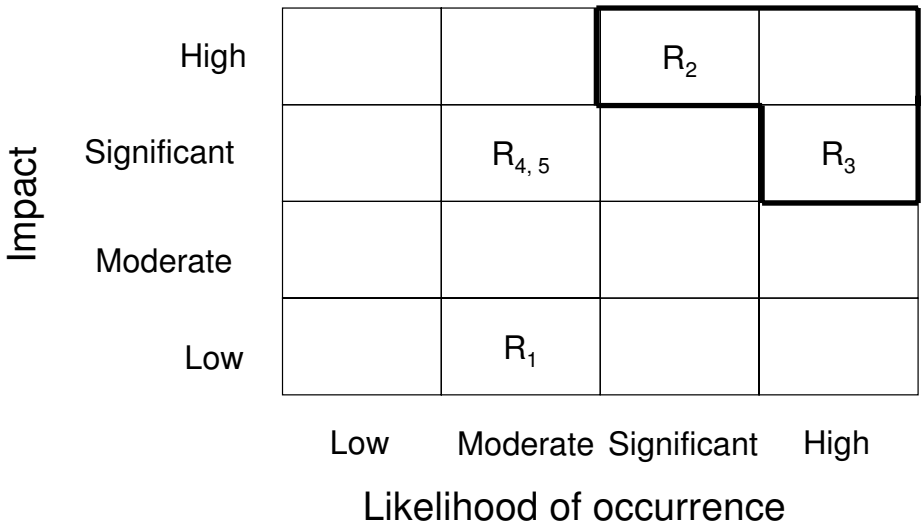

Figure 6.1: Impact vs Likehihood of occurrence. The tolerance line separates risks which are of high concern.

6.4.1 Risk planning

During risk planning, we consider each risk item and develop a strategy to manage that risk. We can distinguish between the following different types of strategies:

- Acceptance: We may decide to accept some risk item(s). In this case, we essentially decide that the damage inflicted by some risk item(s) would be less than the costs of action that might reduce the likelihood of the risk item(s) occurring.

- Avoidance: We avoid activities which are prone to risk. For example, we choose to save sensitive data on a different server provided one can be made available.

- Reduction: We decide to proceed with a task which involves a risk item, but we take precautions to reduce the likelihood of the risk. For example, upgrade server hardware, migrate to a more reliable web server, or purchase a new (reliable) database management system software.

- Mitigation: We take actions to ensure that the impact of risk is lessened

when it occurs. One measure would be to purchase a second storage device (hard drive) to maintain backups of database with completed transactions. Another measure would be to purchase/acquire a secondary database management system to act as a parallel server. We may also decide to place the server under constant monitoring by an administrator to minimize the mean-time-to-repair (or mean-time-to-restore).

- Risk transfer. We can outsource a risk item (normally for a fee). For example, buying insurance.

- Contingency measures. We take actions to reduce the impact once the risk is realized, despite our efforts for risk reduction and mitigation. Examples include migrating to a parallel (more reliable) server (provided one exists) once the database fails.

Risk reduction leverage

We can take measures to reduce the risk exposure factor of a given risk item. However, we need to take into consideration that these measures contain cost. We define *risk reduction leverage* as the ratio of the reduction in risk exposure over the cost of the reduction (countermeasure).

$$Risk\ reduction\ leverage\ (RRL) = \frac{RE_{before} - RE_{after}}{Cost\ of\ risk\ reduction}$$

According to the definition, values of RRL greater than 1 indicate cost effective risk reduction measures because the reduction in risk exposure achieved by taking some measure is greater than the costs involved. On the other hand, values of RRL less than 1 would indicate non cost effective measures.

6.4.2 Risk monitoring

How do we determine if a risk item becomes a problem? Monitoring involves the establishment, of thresholds that can indicate, for each risk item, when we should act. Experience (perhaps from similar projects) is a good basis to judge when things are getting out of hand. Monitoring would provide feedback to previous steps (See Figure 6.2).

During iterative development, risk management is performed iteration by iteration, each iteration addressing different types of risks which are associated with the requirements to be implemented. Furthermore, at the end of an iteration we need to assess our risk management: Do certain risk items manifest themselves over various iterations? Does their prioritization remain the same? Have new risks been discovered which will have to be addressed in later iterations?

6.5 Examples

Example 6.1. Consider a server which maintains a repository of data. The probability of losing the data is 15%. The cost of losing the data is measured in terms of the cost of its reproduction and re-entry into the database which is estimated at $30,000. In order to reduce the risk of losing the data, the company requested the services of a consultant who proposed the following two options:

Option 1. Frequent backup.

Option 2. Replication of database to a parallel server.

The first option is estimated to reduce the risk to 10%. It is estimated to cost $2000. The second option is estimated to reduce the risk to 5%. It is estimated to cost $2500. As a project manager you need to make the following decisions: Calculate the Risk Reduction Leverage factor for the two

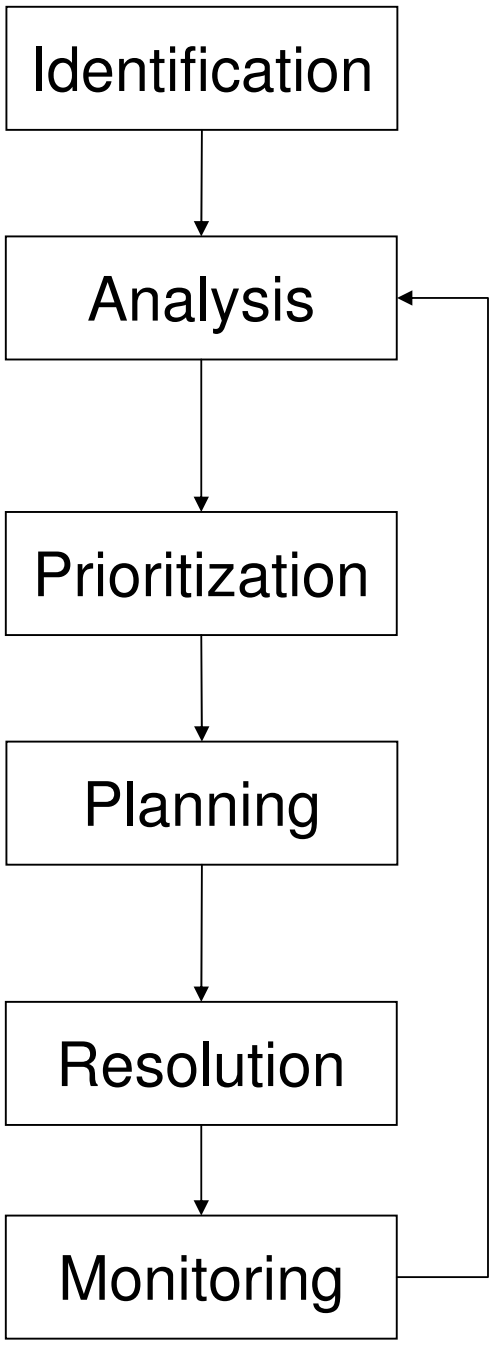

Figure 6.2: Risk monitoring providing a feedback loop.

options above. Discuss which option you would adopt, clearly explaining your reasoning behind your decision.

$$Risk\ reduction\ leverage\ (RRL) = \frac{RE_{before} - RE_{after}}{Cost\ of\ risk\ reduction}$$

$$Option1 : RRL = \frac{(0.15 \times 30,000) - (0.10 \times 30,000)}{2000} = 0.75$$

$$Option2 : RRL = \frac{(0.15 \times 30,000) - (0.05 \times 30,000)}{2500} = 1.2$$

Replication of the database to a parallel server is a more cost-effective option to adopt because its $RRL = 1.2 > 1$. This implies that by spending the cost (of risk reduction) of \$2500, we are getting more benefits in terms of reducing the cost of risk.

Example 6.2. Assume that you are in charge of designing some facility. During risk assessment, you have identified fire as a risk item. The probability of fire is 3%, and it can cause a damage of \$18,000 to the facility.

1. Briefly describe how you can deploy a factor to provide a single numerical value for this risk item and describe the significance and applicability of this factor.

   Recall that:

$$\begin{aligned} Risk\ Exposure(RE) &= (probability\ of\ occurrence) \times (impact) \\ &= 0.03 \times 18000 \\ &= 540. \end{aligned}$$

The RE factor is applicable because it provides a single numerical value to quantitatively describe a risk item, combining probability and impact (when both parameters are available). It is significant in risk prioritization as well as for risk planning (e.g. for the calculation of Risk Reduction Leverage).

2. Can you identify one possible limitation of the above factor?

   One possible limitation for RE is that it would give the same value to an item with high-probability/low impact and one with low-probability/high impact resulting in an equal prioritization. To distinguish between the two we may want to adopt a different approach to evaluate these two risk items. Another possible limitation is that this factor (for a single risk item) may not retain the same values for probability/impact throughout a given time period (e.g. iteration, or phase).

3. To protect against a possible fire, you have decided to install water sprinklers in the facility. The system would cost \$3000. Clearly, the water can also inflict damage to the facility. This damage is estimated to be \$5,000. Is this measure cost effective? In making this decision, please make sure you define any and all applicable terms, and discuss their applicability and significance.

   The new RE is $0.03 \times \$5,000 = \$150$. Let this be RE_{after}.

   $RE_{before} = \$540$

   Risk Reduction Leverage is defined as the ratio of the reduction in Risk Exposure (RE) over the cost of the reduction (countermeasure).

$$\begin{aligned} \textit{Risk reduction leverage } (RRL) =& \frac{RE_{before} - RE_{after}}{Cost\ of\ risk\ reduction} \\ =& \frac{540 - 150}{3000} \\ =& 0.13. \end{aligned}$$

The system is not cost effective.

The factor relates RE_{before} to RE_{after} and allows us to assess (risk) planning.

4. What kind of risk planning is the above measure?

   Installing a water sprinkle system is a type of risk mitigation (to reducing the possible impact of the risk item).

Example 6.3. Consider a server which maintains a database over which many transactions take place. During risk assessment you have identified "database management system (software) crash" as a risk item. Such a crash may result in inconsistent data and transactions over the database. Briefly recommend one possible measure for risk reduction, risk mitigation, and contingency plan.

- Reduction: We decide to proceed with a task which involves a risk item, but we take precautions to reduce the likelihood of the risk. For example, upgrade server hardware, migrate to a more reliable web server, or purchase a new (reliable) database management system software.

- Mitigation: We take actions to ensure that the impact of risk is lessened when it occurs. One measure would be to purchase a second storage device (hard drive) to maintain backups of database with completed transactions. Another measure would be to purchase/acquire a secondary

database management system to act as a parallel server. We may also decide to or place the server under constant monitoring by an administrator to minimize the mean-time-to-repair (or mean-time-to-restore).

- Contingency measures. We take actions to reduce the impact once the risk is realized, despite our efforts for risk reduction and mitigation. Examples include migrating to a parallel (more reliable) server (provided one exists) once the database fails.

Example 6.4. Consider a database management system (DBMS) which maintains client records in a company. The probability of losing the data is 20%. The cost of losing the data is measured in terms of the cost of its reproduction and re-entry into the database which is estimated at $40,000. Your line manager (outside the IT department) has recommended to choose *any single one* of the following two options:

1. Frequent backup. It is estimated to reduce the cost of data reproduction and data re-entry to $20,000. It is also estimated to cost $2000.

2. Migrate to a more reliable DBMS. It is estimated to reduce the risk to 15%. It is also estimated to cost $2500.

1. Briefly describe what *type of planning strategy* each option provides.

2. Discuss which option you would adopt, clearly explaining the reasoning behind your decision.

Solution:

1. Frequent backup is a risk mitigation strategy, as it addresses the impact (cost) of the risk if and when it occurs. On the other hand, the migration to a new DBMS is a risk reduction strategy as it addresses the likelihood (probability) of the risk happening.

2. Use can use risk reduction leverage (RRL) to reason about which option to adopt:

$$RRL = \frac{RE_{Before} - RE_{After}}{Cost}.$$

$$\begin{aligned} RRL_1 &= \frac{(0.2 \times \$40,000) - (0.2 \times \$20,000)}{\$2000} \\ &= 2. \end{aligned}$$

$$\begin{aligned} RRL_2 &= \frac{(0.2 \times \$40,000) - (0.15 \times \$40,000)}{\$2500} \\ &= 0.8. \end{aligned}$$

3. Frequent backup is a better choice as its RRL value is 2. On the other hand, migration to a new DBMS is not at all a cost-effective option as its RRL value is $0.8 < 1$.

Example 6.5. Consider an information system which needs to store data. The data are stored on a serv-er. During a meeting, your manager brings up the issue of a server crash and decides to consider it as a risk item, especially since the hardware over which it runs is very old.

Describe applicable strategies to deal with this risk. For each strategy, briefly describe a) the motivation and b) the conditions under which it can be viable.

Solution:

- Acceptance: Only if the damage inflicted would be less than the costs of action that might reduce the probability of risk happening.

- Avoidance: Avoid activities which are prone to risk. Save data on a different server pro-vided one can be made available.
- Reduction: Take precautions to reduce the probability of the risk. For example, upgrade server hardware, or migrate to a more reliable server.
- Mitigation: Actions taken to ensure that the impact of risk is lessened when it occurs. Maintain backups so that you can restore data from backup should a crash occur. Or, maintain a parallel server.
- Contingency measures: What to do once the risk occurs. Examples include migrate to a more reliable server (provided one exists), or move the server under constant monitoring by the systems administrator to minimize the mean-time-to-repair (or mean-time-to-restore).

6.6 References

Hughes, B. and Cotterel, M., Software project management, 4th edition, McGraw-Hill, 2006.

Chapter 7

Activity planning

7.1 Introduction

Why is many software delivered late? We can trace most reasons to one or more of the following:

- Unrealistic deadlines, normally established by someone outside the development team and forced on managers and developers within the team.
- An underestimate of the amount of effort and/or necessary resources.
- Risks which were not considered or underestimated (ranked as low priority).
- Human/technical difficulties not foreseen or underestimated.
- Miscommunication among team members and among team and management.

The task of *activity planning* (or *scheduling*) is to distribute estimated effort across the planned project duration by allocating effort to tasks. In essence, activity planning is to decide in advance, *what* to do, *why* do it, *how* to do it, *when* to do it and *who* is to do it.

7.2 Fundamental principles of activity planning

Initially, the project must be broken down into a number of manageable tasks. This is referred to as *compartmentalization*. Each task must be allocated some number of work units (e.g. person-hours of effort). Following this, we need to determine the interdependency of each (compartmentalized) task. Some tasks must work in sequence, while some others may work in parallel. Certain tasks may work independently. Finally, each task must be assigned a start-date and a completion-date.

Every project has a defined number of people on the development team. The project manager must ensure that the appropriate number of people have been scheduled at any given time. For example, consider three assigned engineers (i.e. three person-days are available per day of assigned effort) for seven parallel tasks to be accomplished, on some given day, each requiring 0.5 person-days of effort. The problem here is that more effort has been allocated than there are people to do the job, since

$$\begin{aligned} Effort &= 7\ tasks \times 0.5\ person\ days \\ &= 3.5 person\ days \end{aligned}$$

Every scheduled task must be assigned to a specific team member and it should have a defined outcome. The outcome is normally a *work-product* (or *artifact*). Work products are combined into *deliverables*. For example, the artifacts produced by the activities of a an iteration are combined into the deliverable for that particular iteration. Additionally, every task (or group of tasks) should be associated with a project milestone. For example, the end of an iteration (group of tasks) defines a milestone.

7.2.1 The relationship between people and effort

There is a common myth that if we fall behind schedule, we can always add more developers and catch up later in the project. Unfortunately this is not so simple and this policy most likely would create more problems than the ones it aims at solving. Adding people late in a project often has a disruptive effect on the project (causing schedules to slip even further) as new people must learn the system, and people who teach them are the same people who do the work. During teaching, no work is done. Furthermore, more people increase the complexity of communication throughout the project.

7.3 Motivation behind an activity plan

A detailed plan for the project must include a schedule indicating the start and completion times for each activity. This will enable managers to ensure that appropriate resources will be available when needed and avoid having different activities competing for the same resources at the same time. This will also enable managers to allocate staff to each activity.

7.4 Objectives of activity planning

There are three main objectives of activity planning. The first one is *feasibility assessment*, during which managers have to address the following question: Is the project possible within the required resource constraints? For example, if a project requires 12 months, can it be done in 6 months with twice as many people? A second objective is the *allocation of resources* during which managers have to coordinate staff and allocate them to resources. What are the most effective ways of allocating resources to the project? When should resources be available? A third objective is the production of a *detailed costing*. After allocating resources, we can obtain good estimates of costs.

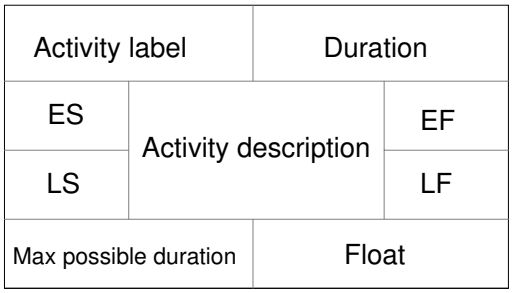

Figure 7.1: A node representing an activity.

7.5 Activity networks

Activity networks are mathematical graphs that help managers to visualize the plan of a project, to assess the feasibility of the completion date, to identify how resources would need to map to activities and how they will need to be deployed to them, and to calculate when costs will be incurred. Activity networks provide a tool for the coordination (and the motivation) of the project team.

7.6 Activity-on-node networks

An *activity-on-node network* is a directed mathematical graph where, as the name suggests, nodes represent activities. Edges represent transitions from one activity to another, explicitly illustrating their sequence. A node includes certain information about an activity, such as earliest and latest times (See Figure 7.1).

We will illustrate activity-on-node networks with the following example:

The requirements specification of an IT application is estimated to take two weeks to complete (a week is seven days). When this activity has been completed, work can start on three software modules, A, B and C. The design/implementation of A, B and C will need five, ten and ten days respectively. Modules A and B can only be unit-tested together as their functionality is closely associated. This joint testing should take two weeks. Module C will

require eight days of unit testing. Once all unit-testing has been completed, the planning of integrated system testing must take place and it would require a ten days. The activity itself would take three weeks. The project manager has decided not to allow any holiday for the duration of this project.

We can translate the problem statement into the following table:

Activity	**Duration**	**Precedence**
Specification (S)	14 days	
Design/Coding A (DCA)	5 days	S
Design/Coding B (DCB)	10 days	S
Design/Coding C (DCC)	10 days	S
Unit Test A,B (UTAB)	14 days	DCA, DCB
Unit Test C (UTC)	8 days	DCC
Planning of Integrated System Test (P)	10 days	UTAB, UTC
Implementing Integrated System Test (IST)	21 days	P

We now have to translate the above specification into a directed graph (see Figure 7.2) illustrating all activities, their durations and their interdependencies.

For each activity we must determine the following parameters:

Earliest start time (ES): the earliest time at which the activity can start given that its precedent activities must be completed first.

Earliest finish time (EF), equal to the earliest start time for the activity plus the time required to complete the activity.

Latest start time (LS), equal to the latest finish time minus the time required to complete the activity.

Latest finish time (LF): the latest time at which the activity can be completed without delaying the project.

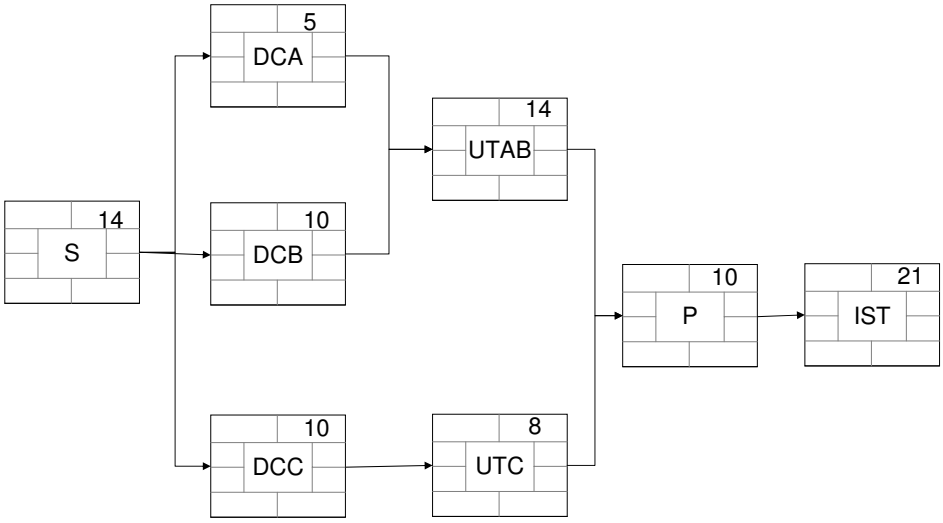

Figure 7.2: Activity-on-node graph: Activities, durations and dependencies.

7.6.1 Determining earliest times: Forward pass

The earliest start and finish times of each activity are determined by working forward[1] through the network and determining the earliest time at which an activity can start and finish considering its predecessor activities. For each activity, we calculate the earliest times by applying the *forward pass rule*.

> **The forward pass rule**: The earliest start date for the current activity is the earliest finish date for the previous. When there is more than one previous activity, we take the latest between all previous activities.

Performing a forward pass

Activity S can start immediately (we follow the convention to write "the end of day 0"). It will take 14 days, which is the earliest it can finish. Thus,

```
ES(S) = 0
EF(S) = 14
```

Activities DCA, DCB and DCC can start as soon as S completes. Activity DCA will take 5 days, so the earliest it can finish is (at the end of) day 19.

[1] According to the direction of the edges.

Similarly, the earliest finish for activities DCB and DCC (both of which will take 10 weeks) is (at the end of) day 24.

```
ES(DCC) = ES(DCB) = ES(DCC) = 14
EF(DCA) = 19
EF(DCB) = EF(DCC) = 24
```

Activity UTAB cannot start until both its preceding activities can finish. The earliest start for UTAB is the latest between the earliest finish of its preceding activities, i.e. the latest between EF(DCA), and EF(DCB) which is (at the end of) day 24. Activity UTAB will take 14 days to finish, and so the earliest it can finish is (at the end of) day 38.

```
ES(UTAB) = 24
EF(UTAB) = 38
```

Activity UTC cannot start until activity DCC finishes, i.e. ES(UTC) = EF(DCC) = 24. Activity UTC will take 8 days, so the earliest it can finish is (at the end of) day 32.

```
ES(UTC) = 24
EF(UTC) = 32
```

Activity P cannot start until both its preceding activities finish. The earliest activity P can start is the latest between the earliest finish of its preceding activities UTAB and UTC, i.e. the latest of EF(UTAB), and EF(UTC), which is 38. Activity P will take 10 days, so the earliest it can finish is (at the end of) day 48.

```
ES(P) = 38
EF(P) = 48
```

Activity IST cannot start until activity P finishes. The earliest start of IST is the earliest finish of P. Activity IST will take 21 days, so the earliest is can finish is (at the end of) day 69.

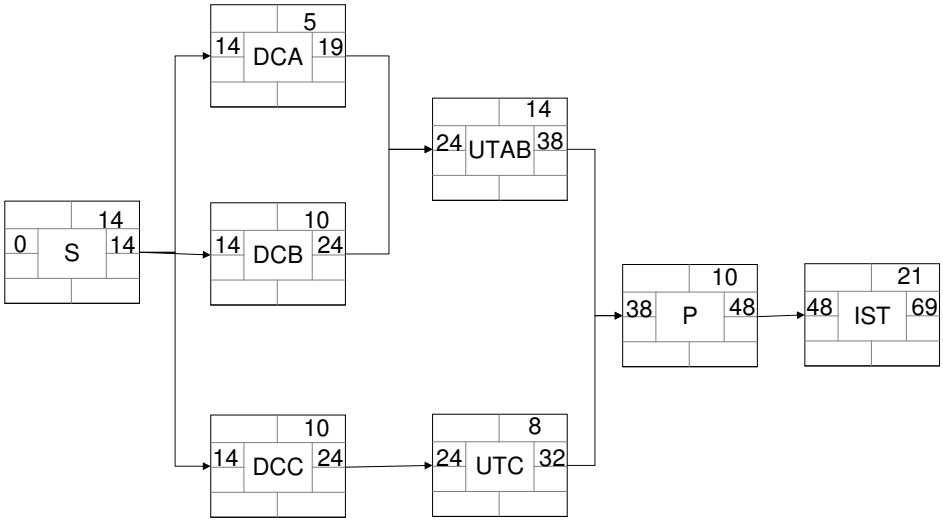

Figure 7.3: Activity-on-node graph after the forward pass.

```
ES(IST) = 48
EF(IST) = 69
```

The project will be complete when activity IST is complete. The earliest the project can complete is (at the end of) day 69.

The forward pass concludes with Figure 7.3 which adds earliest times to the initial graph.

7.6.2 Determining latest times: The backward pass

The latest start and finish times are the latest times that an activity can start and finish without delaying the project and they are found by working backward[2] through the network.

We assume that the latest finish date for the project is the same as the earliest finish date, i.e. we wish to complete the project as early as possible. To calculate the latest times, we apply the *backward pass rule*.

[2]Against the direction of the edges.

Backward pass rule: Start from the last activity: The latest finish (LF) for the last activity is the same as the earliest finish (EF) of that activity.

We now work backwards for each subsequent activities: The latest finish (LF) for a given activity is the latest start of its following activity, i.e. LF (activity) = LS (following activity).

If more than one following activity exists, we take the earliest of the latest start dates of its following activities, i.e. LF (activity) = min (LSs of following activities).

The latest start (LS) of a given activity is given by the difference between its latest finish (LF) and its duration, i.e. LS(activity) = LF(activity) - duration(activity).

The latest completion date for activities IST is assumed to be (the end of) day 69. This assumption is based on the fact that we do not want to delay the project more than its earliest possible completion date. Since the duration of IST is 21 days, its latest start is the difference between its latest finish and its duration.

```
LF(IST) = 69
LS(IST) = LF(IST) - duration(IST) = 48
```

The latest completion date for activity P is the latest date at which the following activity, IST, can start. Also, the latest start for activity P is the difference between its latest finish and its duration i.e.

```
LF(P) = LS(IST) = 48
LS(P) = LF(P) - duration(P) = 48 - 10 = 38
```

The latest completion date for activities UTAB and UTC are the latest day at which the following activity, P, can start.

```
LF(UTAB) = LF(UTC) = LS(P) = 38
```

The latest start for activity UTAB is the difference between its latest finish and its duration, i.e.

```
LS(UTAB) = LF(UTAB) - duration(UTAB) = 38 - 14 = 24
```

Similarly the latest start for activity UTC is the difference between its latest finish and its duration, i.e.

```
LS(UTC) = LF(UTC) - duration(UTC) = 38 - 8 = 30
```

The latest finish dates for activities DCA and DCB are the latest day at which the following activity, UTAB, can start, i.e.

```
LF(DCA) = LF(DCB) = LS(UTAB) = 24
```

The latest start date for DCA and DCB are the differences between their corresponding latest finish dates and their duration, i.e.

```
LS(DCA) = LF(DCA) - duration(DCA) = 24 - 5 = 19
LS(DCB) = LF(DCB) - duration(DCB) = 24 - 10 = 14
```

The latest finish date for activity DCC is the latest day at which the following activity, UTC, can start, i.e.

```
LF(DCC) = LS(UTC) = 30
```

The latest start date for activity DCC is the difference between its latest finish date and its duration, i.e.

```
LS(DCC) = LF(DCC) - duration(DCC) = 30 - 10 = 20
```

The latest finish date for activity S is the earliest of the latest start dates of its following activities, DCA, DCB and DCC, i.e. the minimum of LS(DCA), LS(DCB), and DC(DCC).

```
LF(S) = 14
```

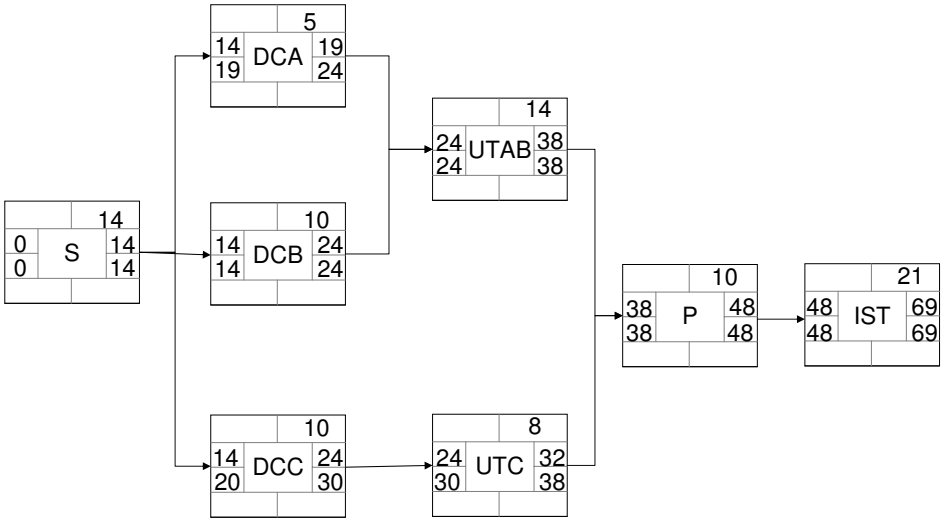

Figure 7.4: Activity-on-node graph after the backward pass.

The latest start date for activity S is the difference between its latest finish date and its duration, i.e.

```
LS(S) = LF(S) - duration(S) = 0
```

The backward pass concludes with Figure 7.4 which adds latest times to the graph.

7.6.3 Activity float and critical path

The difference between an activity's earliest start date (ES) and its latest start date (LS) (or the difference between earliest and latest finish dates) is known as the activity's *float*, i.e.

$$Float = LS - ES$$

The longest possible duration of an activity is given by $LF - ES$.

But $LS = LF - Duration$, so

$$Float = LF - Duration - ES$$

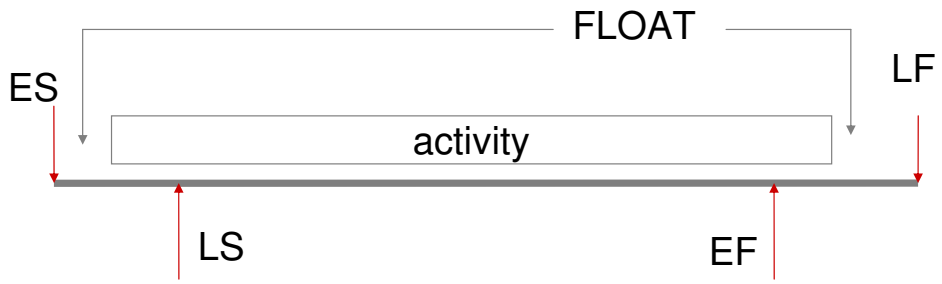

Figure 7.5: Activity float.

The relation between earliest and latest times and float is shown in Figure 7.5.

In essence, float is a measure of how much the start or completion of an activity may be delayed without affecting the end date of the project. Any activity with zero float is critical as any delay will affect the completion of the entire project.

Recall (from discrete mathematics) that on a graph G which includes nodes v and w, we have the following definitions:

A *walk* from v to w is a finite alternating sequence of adjacent nodes and edges.

A *path* from v to w is a walk from v to w that does not contain a repeated edge.

A *simple path* from v to w is a path that does not contain a repeated vertex.

In the activity network, there will be (at least) one (simple) path through the network joining critical activities known as the *critical path*. The critical path is determined by adding the times for the activities in each sequence and determining the longest path in the project. The critical path determines the total calendar time required for the project. If activities outside the critical path speed up or slow down (within limits), the total project time does not change.

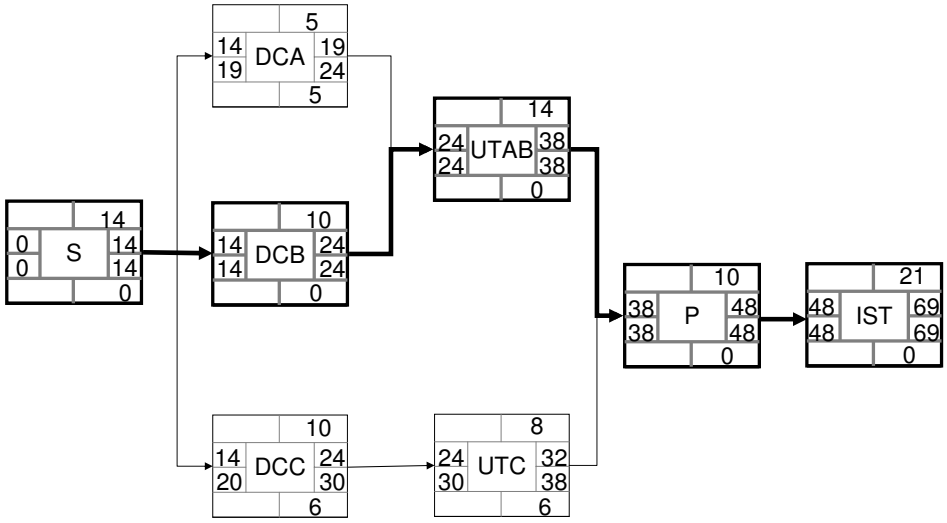

Figure 7.6: Activity-on-node graph: Illustrating the critical path.

Figure 7.6 adds float information to the graph together with an illustration of the critical path.

7.6.4 Maximum duration

There is one parameter we still have not mentioned: As a project manager you need to plan the resources required for each activity. To do that, you need to calculate, for each activity, the maximum possible duration. This is given by $LF - ES$.

Example 7.1. Derive a formula to relate the longest possible duration of an activity to its float.

Solution:

The longest possible duration of an activity is given by LF - ES.

Also,

$$Float = LS - ES[1]$$

$$ButLS = LF - Duration[2]$$

So [1] becomes

$$Float = LF - Duration - ES$$

$$Float + Duration = LF - ES = longestpossibleduration.$$

7.7 Activity-on-arrow networks

An *activity-on-arrow network* (also referred to as *Program, or Project, Evaluation and Review Technique, PERT*) is a mathematical graph where nodes represent events of activities (or groups of activities: starting or finishing), and edges represent activities (may also include durations).

We will demonstrate the steps involved in building such a graph through an example.

Initially we need to specify the individual activities. We make a listing of all the activities in the project, and add duration information. In this example, the listing of activities together with their durations are given:

Activity	**Duration**	**Precedence**
A	7	
B	4	
C	2	A
D	4	B
E	3	B
F	5	D
G	6	D
H	2	C, F
I	3	E, G

We need to determine the sequence of those activities. Some activities are dependent on the completion of others. A listing of the immediate predecessors

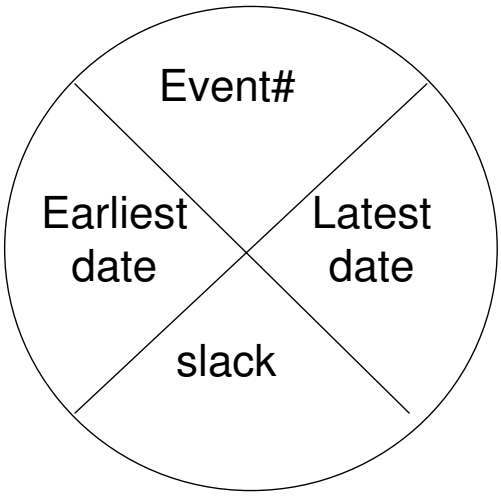

Figure 7.7: A node representing an event.

of each activity is useful for constructing the network diagram. The sequence of activities is given above.

7.7.1 Some notation, rules and conventions

We may have only one start and end node. Since an edge represents an activity, it has a duration. On the other hand, a node (representing some milestone) has no duration. The node notation is shown in Figure 7.7. Nodes are numbered sequentially. If we choose not to show direction of edges, we must follow a convention to read the graph properly. In our convention, time moves from left to right. Furthermore, a graph may not contain loops.

The initial graph is shown in Figure 7.8.

We need to estimate the completion time for each activity. We define the following four quantities for each activity:

ES: Earliest Start time.

EF: Earliest Finish time.

LS: Latest Start time.

LF: Latest Finish time.

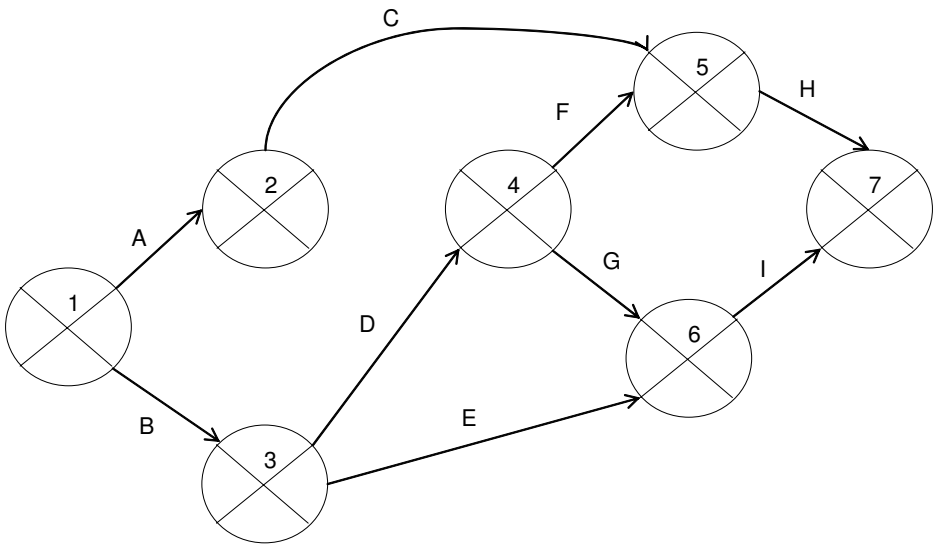

Figure 7.8: Activity-on-arrow graph.

7.7.2 Determining the earliest times: The forward pass

We perform a forward pass on the graph to obtain 1) the earliest date for events and 2) the earliest dates (ES and EF) of activities. The earliest date for an event, is the earliest finish date of the activity terminating at that event. Where more than one activity terminates at a common event, we take the latest of all earliest finish dates for those activities which terminate at that event. Event dates are recorded on the graph and activity dates on an activity table.

The forward pass rule:

1. Earliest Date (Event) : Latest of EFs of all preceding activities

2. ES(activity) = Earliest Date (Event) in which the activity originates from.

3. EF(activity) = ES(activity) + Duration(activity)

Activities A and B can start immediately, so the earliest date for event 1 is zero and the earliest start date for these three activities is also zero. In this

context, zero implies (the end of) day 0 (i.e. the beginning of the first day of the project).

```
Earliest date (1) = 0
ES(A) = 0
ES(B) = 0
```

Activity A will take 7 weeks, so the earliest it can finish is (at the end of) week 7. The earliest we can achieve event 2 is (at the end of) week 7.

```
EF(A) = 7
Earliest date (2) = 7
```

Similarly, activity B will take 4 weeks, so the earliest it can finish is (at the end of) week 4. The earliest we can achieve event 3 is (at the end of) week 4.

```
EF(B) = 4
Earliest date (3) = 4
```

Activity C can start as early as (the end of) week 7 and it will take 2 weeks, so the earliest it can finish is (at the end of) week 9.

```
ES(C) = 7
EF(C) = 9
```

At this point we cannot conclude about the earliest start of event 5, since this event will start (at earliest) at the latest between the early finish of activities C and F.

Activities D and E can start as early as (the end of) week 4.

```
ES(D) = 4
ES(E) = 4
```

Activity D will take 4 weeks so the earliest it can finish is (at the end of) week 8. This is the earliest we can achieve event 4.

```
EF(D) = 8
Earliest date (4) = 8
```

Activity E can start as early as (the end of) week 4 and it will take 3 weeks so the earliest it can finish is (the end of) week 7.

```
ES(E) = 4
EF(E) = 7
```

At this point we cannot conclude about the earliest start of event 6, since this event will start (at earliest) at the latest between the early finish of activities G and E.

Activity G can start as early as (the end of) week 8 and it will take 6 weeks, so the earliest it can finish is (at the end of) week 14.

```
ES(G) = 8
EF(G) = 14
```

The earliest date of event 6 is the highest between the EFs of G, E. The earliest start of event 6 is (the end of) week 14.

```
Earliest date (6) = 14
```

Activity F can start as early as (the end of) week 8 and it will take 5 weeks, so the earliest it can finish is (at the end of) week 13.

```
ES(F) = 8
EF(F) = 13
```

The earliest date of event 5 is the latest finish between activities C and F, i.e. highest between (9, 13) = 13.

```
Earliest date (5) = 13
```

Activity H can start as early as (the end of) week 13 and it will take 2 weeks. S the earliest it can finish is (at the end of) week 15.

```
ES(H) = 13
EF(H) = 15
```

Activity I can start as early as (the end of) week 14 and it will take 3 weeks, so the earliest it can finish is (the end of) week 17.

```
ES(I) = 14
EF(I) = 17
```

Event 7 can be achieved at the highest between the early finish of activities H and I, i.e. max EF(H), EF(I) = max(15, 17) = 17.

```
Earliest date (7) = 17
```

The result of the forward pass is shown in the activity table below and illustrated on Figure 7.9.

Activity	**Duration**	**ES**	**EF**
A	7	0	7
B	4	0	4
C	2	7	9
D	4	4	8
E	3	4	7
F	5	8	13
G	6	8	14
H	2	13	15
I	3	14	17

7.7.3 Determining the latest times: The backward pass

We perform a backward pass to obtain 1) the latest date for events and 2) the latest dates (LS and LF) dates of activities. We assume that the latest finish date for the project is the same as the earliest finish date. The latest date for an event is the latest date by which all immediately following

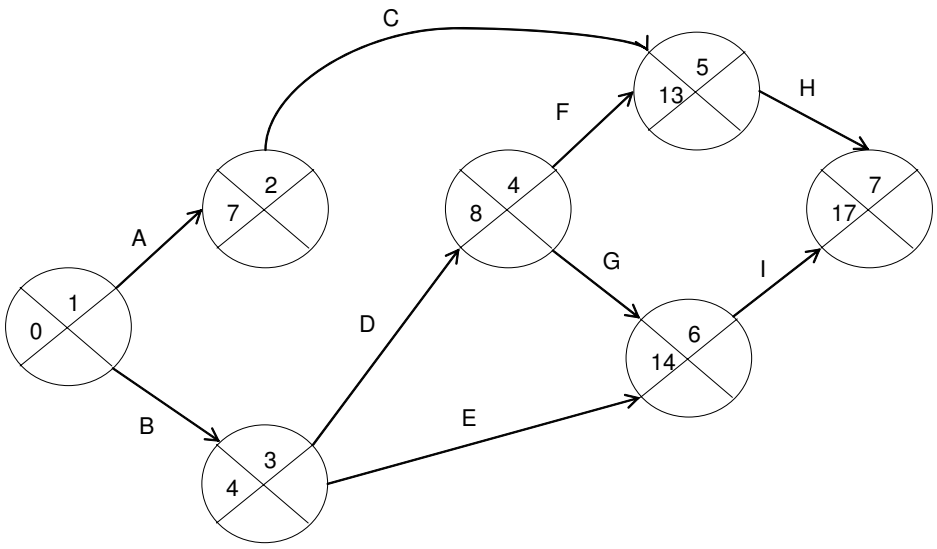

Figure 7.9: Activity-on-arrow graph after the forward pass.

activities must be started (for the project to be completed on time). Where more than one activity originates from a common event, we take the earliest of the latest start dates for those activities. As in the forward pass, event dates are recorded on the graph and activity dates are recorded on the activity table.

The backward pass rule:

1. Latest Date (Event) : Earliest of LS of all following activities.

2. LF(activity) = Latest Date (Event) in which the activity terminates at.

3. LS(activity) = LF(activity) - Duration(activity).

To avoid unnecessary delays, we want the latest date of event 7 to be the same as its earliest date.

```
Latest date (7) = 17
```

This implies that the latest date at which activities H and I can finish is (at the end of) week 17.

```
LF(H) = 17
LF(I) = 17
```

The latest start of activity H is given by

```
LS(H) = LF(H) - duration(H) = 17 - 2 = 15
```

The latest date of event 5 is the latest start of activity H.

```
Latest date (5) = LS(H) = 15
```

The latest start of activity I is given by

```
LS(I) = LF(I) - duration(I) = 17 - 3 = 14
```

The latest date of event 6 is the latest start of activity I.

```
Latest date (6) = LS(I) = 14
```

Activities C and F terminate at the same event. The latest finish of activities C and F is the latest date of event 5.

```
LF(C) = LF(F) = Latest date (5) = 15
```

The latest start of activity C is given by

```
LS(C) = LF(C) - duration(C) = 15 - 2 = 13
```

The latest start of activity F is given by

```
LS(F) = LF(F) - duration(F) = 15 - 5 = 10
```

The latest date for event 4 is given by

```
Latest date (4) = LS (F) = 10
```

The latest dates for activity G are given by

```
LF(G) = latest date (6) = 14
LS(G) = LF(G) - duration(G) = 14 - 6 = 8
```

The latest date for event 2 is given by

```
Latest date (2) = LS(C) = 13
```

The latest dates for activity A are given by

```
LF(A) = latest date (2) = 13
LS(A) = LF(A) - duration(A) = 13 - 7 = 6
```

The latest dates for activity D are given by

```
LF(D) = Latest date (4) = 10
LS(D) = LF(D) - duration (D) = 10 - 4 = 6
```

The latest dates for activity E are given by

```
LF(E) = latest date (6) = 14
LS(E) = LF(E) - duration (E) = 14 - 3 = 11
```

The latest date for event 3 is the earliest of the LS of its following activities D and E, i.e.

```
Latest date (3) = min of LS(D), LS(E) = min (6, 11) = 6
```

The latest dates for activity B are given by

```
LF(B) = Latest date (3) = 6
LS(B) - LF(B) - duration (B) = 6 - 4 = 2
```

The latest date for event 1 is the is the earliest of the LS of its following activities A and B, i.e.

```
Latest date (1) = min of LS(A), LS(B) = min (6, 2) = 2
```

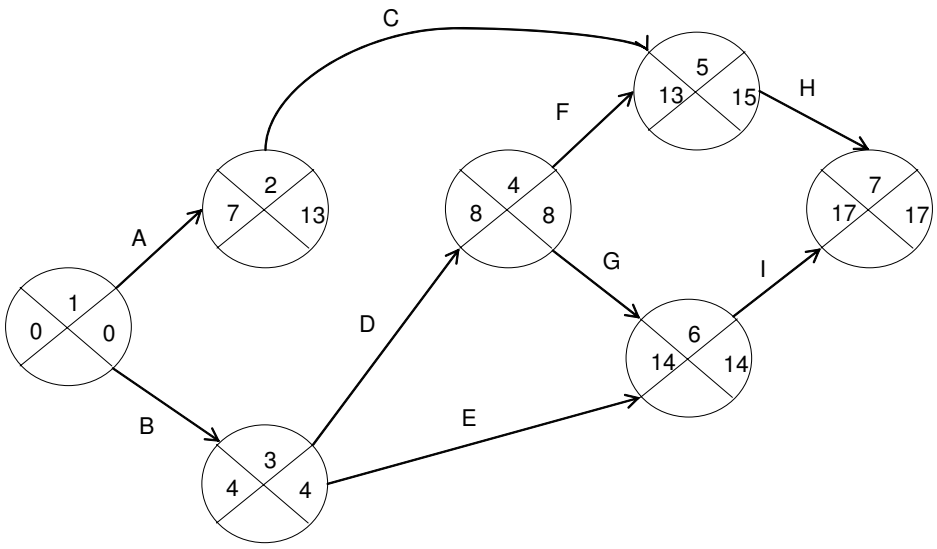

Figure 7.10: Activity-on-arrow graph after the backward pass.

The result of the backward pass is shown in the activity table below and illustrated on Figure 7.10.

Activity	**Duration**	**ES**	**EF**	**LS**	**LF**
A	7	0	7	6	13
B	4	0	4	0	4
C	2	7	9	13	15
D	4	4	8	4	8
E	3	4	7	11	14
F	5	8	13	10	15
G	6	8	14	8	14
H	2	13	15	15	17
I	3	14	17	14	17

7.7.4 Slack times and critical path

The *slack time* for an event is the time between its earliest and latest start time, or between its earliest and latest finish time. Slack is a measure of how late an event may be, without delaying the project. The *critical path* is the path joining all activities with a zero float (See Figure 7.11).

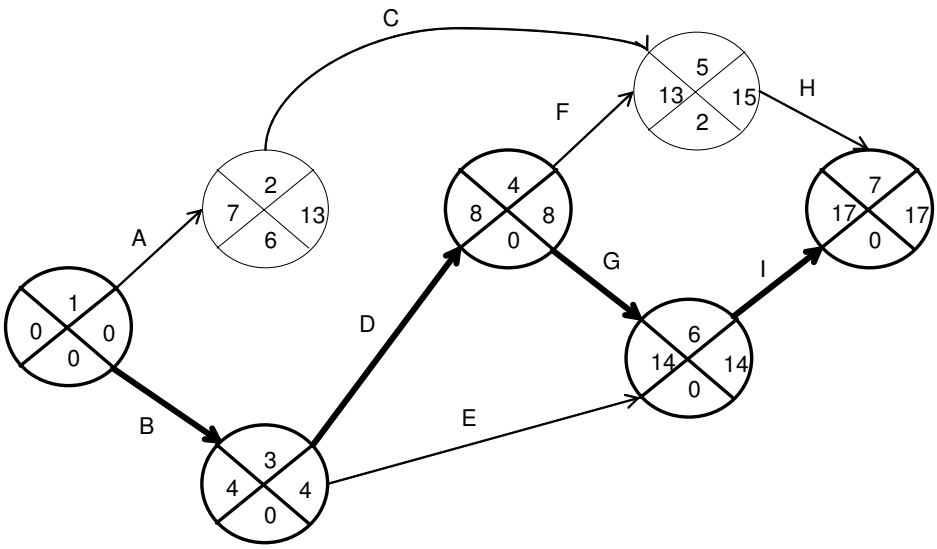

Figure 7.11: Activity-on-arrow graph with slack times and critical path.

The significance of the critical path is that the activities that lie on it cannot be delayed without delaying the project. Because of its impact on the entire project, critical path analysis is an important aspect of project planning. To accelerate the project it is necessary to reduce the total time required for the activities in the critical path.

7.8 A graph is not static

We can build a graph at the beginning of the project (iteration). Initially the graph contains estimates times. However, as the project progresses, task completion times become known and may deviate from our estimations. Furthermore, as the project progresses we may have unanticipated events occurring which can delay certain activities. For these reasons during the project (iteration) we need to update the graph with any and all changes, during which a new critical path may emerge.

7.9 References

Hughes, B. and Cotterel, M., Software project management, 4th edition, McGraw-Hill, 2006.

Chapter 8

Monitoring and control

8.1 Introduction

"Where are we on schedule?" "Where are we on budget?" "Are tasks get accomplished according to plan?" To be able to answer these questions, we need to develop a system of performance reporting. This would involve the collection and dissemination of information on *status*, *progress*, as well as *forecasting*:

Status reporting is a description on where the project currently stands regarding schedule and budget. Examining one of these parameters alone may be misleading.

For example, consider a project which is on budget. Is this enough to consider the progress successful? The answer is No. We cannot draw any conclusion unless we have information about the time involved. Have all planned tasks been completed? If not, then this implies that we are spending the money we planned to spend but we are not accomplishing the tasks we planned to accomplish by this moment in time.

As another example, consider a project which is on time. Without obtaining information about the budget, we cannot draw any conclusion. The project may be on-time only because additional resources have been deployed and it is over budget.

Progress reporting is a description on what tasks have been accomplished by the time of reporting (such as the percentage of completed tasks).

Forecasting is a prediction of future project status and progress.

Not only we need the above information to make some judgment about the state of the project, but we may also need to apply proper controls to bring the project back on track.

8.1.1 Planned value

For a given task k, the *planned value (PV_k)* is defined as the effort planned for that task. The effort is initially described in terms of person-days. Based on the rate of payment of the people involved, the effort will eventually be translated into cost. As a project is essentially a collection of tasks, in order to determine the progress at any given point along the project schedule, the PV of the project is the sum of the PV_k values of all tasks that should have been completed by that point in time on the project schedule.

Example: Please remember that in all reports, we are taking a "snapshot" of the project at a single point in time. Consider a project with the following (linear) tasks (in parentheses are the associated planned values): A(20), B(5), C (10), D (20), E (20), F (10), and G (15). If the tasks are placed on a time line (See Figure 8.1), then on the time indicated we plan to have spent the amount of 35.

8.1.2 Budget at completion

The *budget at completion (BAC)* is the summation of the PV values for all tasks:

$$BAC = \Sigma(PVk)\ for\ all\ tasks\ k.$$

In the previous example, the BAC is 100.

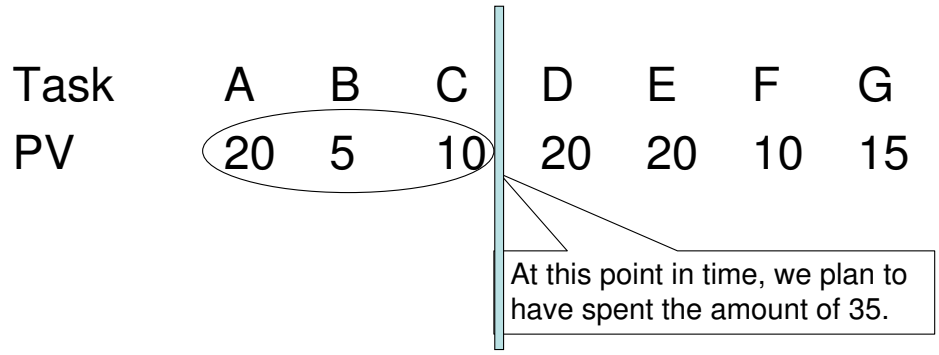

Figure 8.1: Planned value.

8.1.3 Percent scheduled for completion

The *Percent scheduled for completion* is an indication of the percentage of work that should have been completed by a given point in time and it is given by

$$Percent\ scheduled\ for\ completion = \frac{PV(project)}{BAC}$$

In the example above, $PV(project) = 35$, and $BAC = 100$, so the percent scheduled for completion is 35%.

8.1.4 Actual cost

The *actual cost (AC)* is defined as the total of costs on tasks that have actually been completed by a given point in time on the project schedule.

8.1.5 Earned value

The notion of earned value is essential in monitoring and control to such a degree that in many cases, authors refer to this topic as *earned value analysis*. The main idea behind earned value analysis is that the value of the product increases as tasks are completed. We define *earned value (EV)* as the planned value of the work actually completed, i.e. the sum of the PV values of all tasks

that have actually been completed by a given point in time.

8.1.6 Percent complete

Percent complete provides a quantitative indication of the percent of completion of the project at a given point in time:

$$Percent\ complete = \frac{EV}{BAC}$$

8.2 Performance indicators

The values for PV, AC and EV are used in combination to provide measures of whether or not work is being accomplished as planned:

1. Cost Variance (CV)
2. Schedule Variance (SV)
3. Cost Performance Index (CPI)
4. Schedule Performance Index (SPI)

8.2.1 Cost variance

The *cost variance (CV)* is defined as

$$Cost\ Variance\ (CV) = EV - AC$$

Cost variance determines the difference between the earned value and the actual cost of work performed, or the difference between what we should have paid for work actually performed, and what was actually paid for work actually performed. It is an absolute indication of cost savings (against planned cost) or shortfall at a particular stage of a project. A positive variance implies less money was spent for the work accomplished than what was planned to

be spent. A negative variance means more money was spent for the work accomplished than what was planned.

8.2.2 Schedule variance

The *schedule variance (SV)* is defined as

$$Schedule\ Variance\ (SV) = EV - PV$$

Schedule variance determines the difference between earned value and the planned budget, or the difference between what we should have paid for work actually accomplished, and what we intend to pay for work accomplished. A positive value for SV implies that we are ahead of schedule. A negative value implies that we are behind schedule.

8.2.3 Cost performance index

The *cost performance index (CPI)* is defined as

$$Cost\ Performance\ Index\ (CPI) = \frac{EV}{AC}$$

The CPI determines the ratio of the earned value over the actual cost of completed work, or a quotient of what we should have paid for work performed, and what was actually paid for work actually performed. CPI values close to 1.0 provide a strong indication that the project is within its defined budget. CPI values greater than 1 indicate that the cost of completing the work is less than planned. Similarly, CPI values less that 1 indicate that the cost of completing the work is higher than planned.

8.2.4 Schedule performance index

The *schedule performance index (SPI)* is defined as

$$Schedule\ Performance\ Index\ (SPI) = \frac{EV}{PV}$$

SPI is a quotient of the earned value over the planned value and it indicates the rate of progress: the efficiency with which the project is utilizing scheduled resources. SPI values close to 1.0 indicate efficient execution of the project schedule. SPI values greater than 1 are favorable. Values less than 1 are not favorable.

8.2.5 Estimate at completion

The *estimate at completion (EAC)* is defined as

$$EAC = \frac{BAC}{CPI}$$

Initially (before the project starts) our estimate is given by BAC. As we embark on the project, EAC is the quotient of what we planned to spend over what we are actually spending and it indicates what we expect the job to cost.

8.2.6 Estimate to complete

Estimate to complete is defined as

$$ETC = EAC - AC$$

At any given point in time, ETC is a comparison of the estimate of the final cost (EAC) to what we have spent to date (AC). ETC indicates how much more will have to be spent in order to complete the project.

8.3 Getting the project back on track

Usually, projects run into delays or other unexpected events. A project manager has the responsibility to recognize when this is happening (or when this

is about to happen). With minimum delay and minimum disruption to the project, the project manager has the responsibility to mitigate the effects of these events over the project. The overall duration of a project is determined by the current critical path. Speeding up non-critical path activities will not have any effect on the project completion date. What options might be available?

- Allocate more efficient resources on activities on the critical path (or swapping resources between critical and non-critical activities), e.g. swap experienced programmers with junior programmers. Note that adding a programmer to a team might be counter-productive due to the time (and resources) required to bring the new people on board with the project.
- Resources can be available for longer (e.g. on weekends).
- People can work overtime.

Other options include overlapping certain activities so that the start of one activity does not have to wait for completion of another. This can also apply to iterations, i.e. iteration $N + 1$ can start before iteration N is completed. Yet another option can be to renegotiate the contract.

Example: Consider the activity diagram of Figure 8.2 and assume that the project is being prepared for a progress review at the end of the second quarter (today). According to the budget plan, at the end of second quarter, tasks 1, 3, 5, 6 must have been completed. Task 2 should have been completed by 2/3, task 4 should have been completed by 40%, and task 7 should have been completed by 50%. The project manager reports that tasks 1-6 have been completed as planned, except task 7, which is 10% complete. The project manager also reports that the amount of money already spent to date is $10,000. Apply any and all appropriate performance indicators to provide an analysis of the project status as of today in terms of 1) schedule and 2)

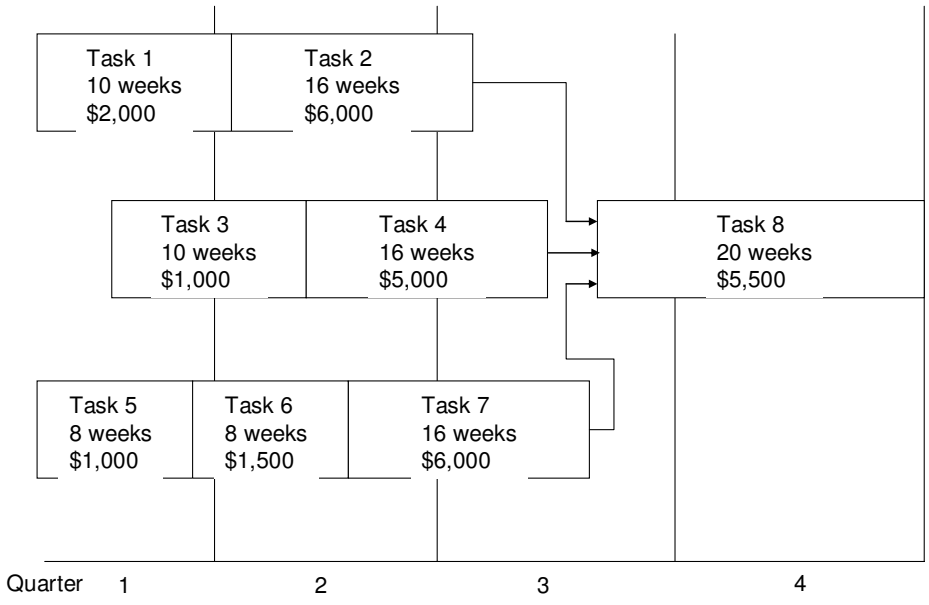

Figure 8.2: Monitoring and control: Example 1.

budget. If your analysis presents a problematic situation, briefly describe your recommendations to bring the project back to track.

- $PV = (100\% \times 2,000) + (2/3 \times 6,000) + (100\% \times 1,000) + (40\% \times 5,000) + (100\% \times 1,000) + (100\% \times 1,500) + (50\% \times 6,000) = 14,500$
- $EV = (100\% \times 2,000) + (2/3 \times 6,000) + (100\% \times 1,000) + (40\% \times 5,000) + (100\% \times 1,000) + (100\% \times 1,500) + (10\% \times 6,000) = 12,100$

The project is behind schedule (since $EV < PV$). The project is under budget (since $AC < EV$). The performance indicators are calculated as follows:

- The Schedule Variance $(SV) = EV - PV = 12,100 - 14,500 = -2,400$.
- The Schedule Performance Index $(SPI) = \frac{EV}{PV} = 0.82 < 1$.
- The Cost Variance $(CV) = EV - AC = 12,100 - 10,000 = +2,100$.
- The Cost Performance Index $(CPI) = \frac{EV}{AC} = 1.21$.

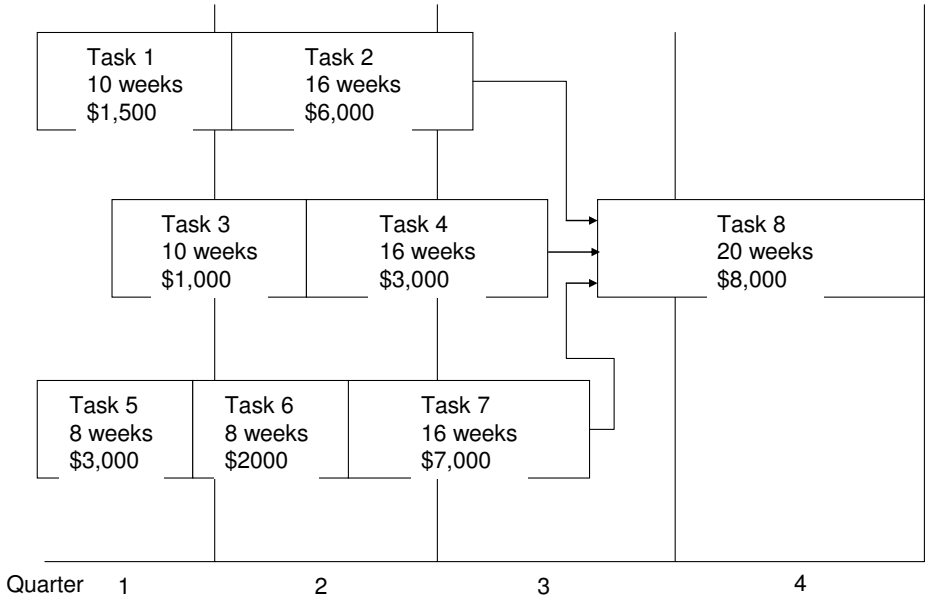

Figure 8.3: Monitoring and control: Example 2.

As we are under budget, we can afford to pay people overtime (to complete more tasks), or recruit more people (even though this may not be desirable), or even to replace junior developers with senior developers (to have more tasks accomplished).

Example: Consider the activity diagram of Figure 8.3 and assume that the project is being prepared for a progress review at the end of the second quarter (today). According to the budget plan, at the end of second quarter, tasks 1, 3, 5, 6 must have been completed. Task 2 should have been completed by 60%, task 4 should have been completed by 1/2, and task 7 should have been completed by 40%. The project manager reports that tasks 1-6 have been completed as planned, except task 7, which is 15% complete. The project manager also reports that the amount of money already spent as of today is $10,000.

We can perform an analysis as follows:

Project budget. The project budget, BAC, is given by

$$\begin{aligned}\Sigma(PV_k) &= 1,500 + 6,000 + 1,000 + 3,000 + 3,000 + 2,000 + 7,000 + 8,000 \\ &= 31,500.\end{aligned}$$

Planned value. The planned value, PV, of the project at this moment in time is given by

$$\begin{aligned}PV &= (100\% \times 1,500) + (60\% \times 6,000) + (100\% \times 1,000) + \\ &\quad (1/2 \times 3,000) + (100\% \times 3,000) + (100\% \times 2,000) + \\ &\quad (40\% \times 7,000) \\ &= 15,400\end{aligned}$$

Interpretation: As of today (the end of second quarter), team members are expected to have completed $\$15,400$ worth of work.

Percent complete. The percent complete is given by $\frac{PV}{BAC} = \frac{15,400}{31,500} = 0.48$

Interpretation: As of today, the project is planned to be 48% completed.

Earned value. The earned value, EV, is given by

$$\begin{aligned}EV &= (100\% \times 1,500) + (60\% \times 6,000) + (100\% \times 1,000) + \\ &\quad (1/2 \times 3,000) + (100\% \times 3,000) + (100\% \times 2,000) + \\ &\quad (15\% \times 7,000) \\ &= 13,650.\end{aligned}$$

Interpretation: As of today, the work that the team members performed, cost $\$13,650$.

Percent complete. The percent complete is given by $\frac{EV}{BAC} = \frac{13{,}650}{31{,}500} = 0.43$.

Interpretation: As of today, 43% of the project is completed.

Actual cost, The actual cost, $AC = 10,000$.

Interpretation: As of today, the work that the team members performed cost $\$10,000$.

Schedule variance. The schedule variance, SV, is given by $EV - PV = 13,650 - 15,400 = -1,750$.

Interpretation: As of today, the project is behind schedule.

Schedule performance index. The schedule performance index, SPI, is given by $\frac{EV}{PV} = \frac{13{,}650}{15{,}400} = 0.88 < 1$.

Interpretation: As of today, the project is behind schedule. The team has completed 88% of what it should have completed.

Cost variance. The cost variance, CV, is given by $EV - AC = 13,650 - 10,000 = +3,650$.

Interpretation: As of today, the project is under budget.

Cost performance index. The cost performance index, CPI, is given by $\frac{EV}{AC} = \frac{13{,}650}{10{,}000} = 1.36$.

Interpretation: As of today, we are getting $\$1.36$ for every dollar that we spend.

Estimate at completion. The estimate at completion, EAC, is given by $\frac{BAC}{CPI} = \frac{31{,}500}{1.36} = 23161.76$.

Interpretation: This value is the estimation of cost at the end of project (Quarter4), based on Cost Performance Index.

Estimate to complete. The estimate to complete, ETC, is given by $EAC - AC = 23161.76 - 10,000 = 13161.76$.

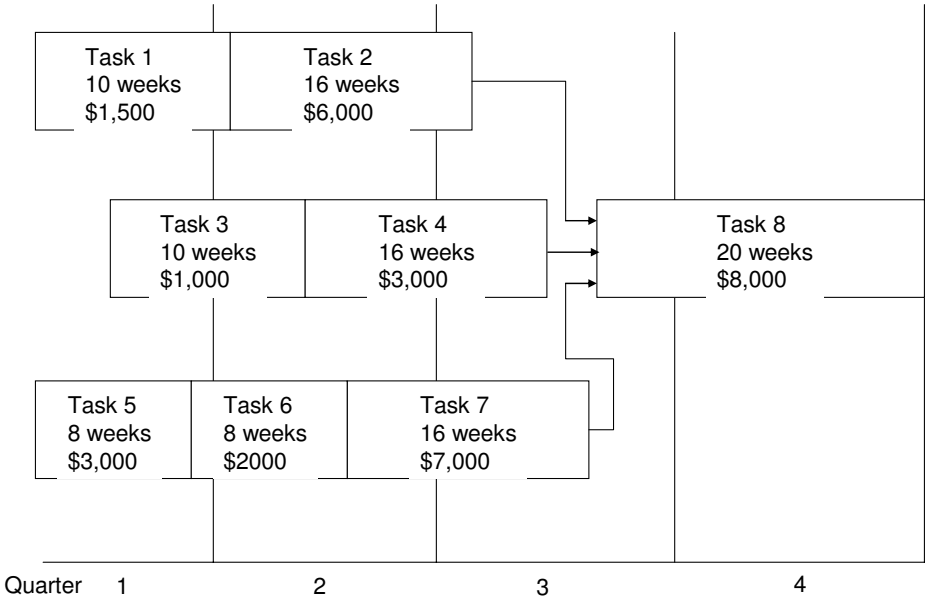

Figure 8.4: Monitoring and control: Example 3.

Interpretation: This value represents the amount of money still to be spent.

variance at completion. The variance at completion, VAC, is given by $BAC - EAC = 31,500 - 23161.76 = 8338.235$.

Interpretation: This value represents the difference between the planed cost at the beginning of the project and the estimation of cost based on today's CPI.

Example: Consider the activity diagram of Figure 8.4 and assume that the project is being prepared for a progress review at the end of the second quarter (today). According to the budget plan, at the end of second quarter, tasks 1, 3, 5, 6 must have been completed. Task 2 should have been completed by 2/3, task 4 should have been completed by 40%, and task 7 should have been completed by 60%. The project manager reports that tasks 1-6 have been completed as planned, except task 7, which is 15% complete. The project manager also reports that the amount of money already spent to date is $15,000.

	Planned				Actual		
Task	Cost	Duration (weeks)	Status	Planned Value (PV)	Cost (AC)	Status	Earned Value (EV)
Task 1	$1,500.00	10	100.00%	$1,500.00		100.00%	$1,500.00
Task 2	$6,000.00	16	66.67%	$4,000.00		66.67%	$4,000.00
Task 3	$1,000.00	10	100.00%	$1,000.00		100.00%	$1,000.00
Task 4	$3,000.00	16	40.00%	$1,200.00		40.00%	$1,200.00
Task 5	$3,000.00	8	100.00%	$3,000.00		100.00%	$3,000.00
Task 6	$2,000.00	8	100.00%	$2,000.00		100.00%	$2,000.00
Task 7	$7,000.00	16	60.00%	$4,200.00		15.00%	$1,050.00
Task 8	$8,000.00	20	0.00%	$0.00		0.00%	$0.00
Total	**$31,500.00**	**104**		**$16,900.00**	**$15,000.00**		**$13,750.00**

Figure 8.5: Monitoring and control: Example 3.

The analysis is tabulated in Figure 8.4 and Figure 8.5

Bringing the project back on track: To bring this project back on track, the manager can recommend the following:

- To correct the schedule deficiencies, renegotiate contract with stakeholders to reduce the scope of the project or extend the delivery date; assign more skilled developers to critical tasks; use reusable components to minimize the time of development (COTS/Open Source)

- To correct the budget deficiencies, perform a forecasting assessment of the project's financial status

  - $EAC = \frac{BAC}{CPI} = \$34,239$
  - $ETC = EAC - AC = \$19,239$
  - $VAC = BAC - EAC = -\$2739$

Since $VAC < 0$, this indicates that the whole project will be over budget based on the estimates of the remaining work. It is therefore incorrect to recommend any actions that would result in more funding being needed unless the stakeholders/development organization are willing to invest more. For

Indicator	Calculation	Interpretation
SV	SV = EV - PV = $13,750 - $16,900 SV = -$3,150.00	Behind schedule
SPI	SPI = EV/PV = $13,750 / $16,900 SPI = 0.81	Current status is 81% of what was planned
CV	CV = EV - AC = $13,750 - $15,000 CV = -$1,250.00	Over budget
CPI	CPI = EV / AC = $13,750 - $15,000 CPI = 0.92	Earning $0.92 for each $1.00 spent

Figure 8.6: Monitoring and control: Example 3.

example, asking the current developers to work overtime or hire new developers would actually worsen the budget situation.

Chapter 9

Software economics

9.1 Introduction

Economics is the the study of how we make decisions in resource-limited situations. During software development, we are faced with numerous considerations, such as make-or-buy, how many options to build, which architecture to use, how much testing (prototyping, specifying) is enough, how much software to re-use, which requirements to add first, and how much time/staff are available.

We define *software cost* to refer to a function of the following five parameters:

1. The size of the end product (in terms of human-generated components), quantified in terms of source instructions as source lines of code (SLOC).

2. The process used to produce the end product.

3. Personnel. Technical expertise of personnel as well as their experience in the problem (applications) domain.

4. Environment. Tools and techniques available.

5. Required quality. Features of the product, performance, reliability and adaptability.

The above parameters cannot be viewed in isolation and in many cases they are interdependent. For example, the deployment of tools enable the reduction in the size of the software. As another example, we can view the adoption of UML which provides a common vocabulary, a tool for thought and a formal way to prove a concept, as a procedure to reduce the overall size of the collection of artifacts, and to improve teamwork.

We need to consider several dimensions around the five parameters of the software cost model (discussed in the subsequent sections).

9.2 Reducing the size or complexity of development

One of the goals during development is to produce a product which meets the specifications with the minimum amount of human-generated SLOC. The reduction in SLOC has been one of the driving factors behind improvements the evolution of higher-order languages (such as object-oriented languages), computer-aided software engineering (CASE) tools (such as automatic code generators), and reuse of commercial components (such as commercial-off-the-shelf components). The reduction in size usually increases several quality attributes such as understandability, changeability and reliability. One typical negative side effect is that as abstraction increases, performance tends to decrease, even though this can be (to some extend) overcome by an improvement in hardware performance.

9.2.1 Reuse

As software products need to satisfy both technical and non-technical criteria, developers find it essential to combine theory and experience in order to reuse proven designs. The importance of reuse lies on the fact that it can speed up the development process, cut down costs, increase productivity and improve

the quality of software. Design- level reuse is viewed as the attempt to share certain aspects of an approach across various projects. Object and component-based systems offer a wide spectrum of techniques to reuse designs on different levels, ranging from frameworks and patterns that constitute approaches on how to best program "in-the-large" to libraries and programming languages that constitute approaches on how to best program "in-the-small." On the higher level of the reuse spectrum we have frameworks and (object-oriented) design patterns. On the lower level of the reuse spectrum, the importance of the mechanism of inheritance and the contribution of Object-Oriented Programming (OOP) towards reusability has been extensively discussed in the literature. Inheritance allows non-invasive (incremental) modification of class definitions. Subclasses may introduce new attributes and methods whose definitions may be based on those of the super-classes. Furthermore, methods may be overloaded (providing different semantics for the same method names) or overridden (providing new semantics for the same method signatures). A key metric in identifying whether a component is truly reusable is to see whether it can have a positive impact on productivity, i.e. money can be made (or saved) on it.

9.2.2 Software components

Component-Based Software Development (CBSD) is based on the idea that there are many software components already built, and that building new systems by selecting the appropriate existing components and assembling them following a well defined software architecture can be more rapid and cost effective approach than building a system from scratch. CBSD has a potential for:

1. Reducing the cost and time of the software development and maintenance, thus increasing productivity.

2. Improving the system maintainability and flexibility by allowing replace-

ment of old components.

3. Enhancing the system quality by allowing components to be built by experts in the domain.

CBSD consists of the following activities:

1. Requirement analysis.
2. Architecture selection and creation.
3. Component selection.
4. Component integration.
5. System testing.

In the literature there is no standardized definition for the term component. Szyperski (2002) defines a component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be deployed independently and is subject to composition by third parties" (Szyperski, 2002)[1].

Brown and Wallnau (1998) discuss four established definitions in order to determine the characteristics of a component. Cai *et al.* (2000) summarize those characteristics to be as follows: A component...

1. is an independent and replaceable part of the system that fulfills a clear function.
2. works in the context of a well defined architecture.
3. communicates with other components via their interfaces.

[1]The definition initially appeared as an outcome of the ECOOP'96 Workshop on Component-Oriented Programming

Components are grouped into two main categories: Commercial Off-The-Shelf (COTS), usually closed source, and Open-Source Software (OSS), usually non-commercial. Even though both COTS and OSS components have similar main advantages from a software maintenance point of view, each has its own advantages and disadvantages. Li *et al.* (2006) investigate the decision making while choosing components. COTS components are usually delivered as binaries rather than as source code in order to protect the rights of developers. OSS components are provided as source code, which is more often the only available documentation. Some of the disadvantages of OSS components discussed in the literature are the lack of a face-to-face development process, code quality, and lack of maintainability.

Ideally we would like to reduce custom development and maximize integration of commercial components and off-the-shelf products. Decisions on custom-built vs. off-the-shelf pose a number of trade-offs which affect quality and cost.

9.2.3 Commercial components vs. custom development

We can identify a number of advantages of commercial components: First, they are available now as opposed to an estimated time for deployment of a custom made component and the costs are known (costs for their license), as opposed to cost estimations for custom built components which may be off. Second, the technology deployed for commercial components would normally be mature and broadly used, so the associated risks are normally rather low. Third, commercial components have a dedicated support. Fourth, they tend to be rich in functionality (sometimes offering more than what we need). The disadvantages of commercial components can be high (and recurring) license fees and the frequent upgrades (which can be also costly), and the dependency on their vendor which usually imposes incompatibilities to other tools. A rich functionality seen as an advantage may also be a disadvantage as unnecessary features may consume extra resources. Further, by deploying commercial

components we have no control over upgrades or maintenance (including bug fixes, which can sometimes be a nuisance) especially in cases where there are problems with the reliability and stability of these components.

On the other hand, we can identify a number of advantages of custom development: First, we have complete control over development and maintenance. Second, we can build our components to fully utilize our environment, thus achieving better performance. The disadvantages of custom development include (sometimes underestimated) cost, and sometimes unpredictable delivery dates. The draining on expert resources can also be seen as a disadvantage.

9.3 Process improvement

Iterative processes help to reduce (or even to avoid) rework.

9.4 Personnel management

It is important to note that teamwork is more important than the sum of the expertise of the individuals. Boehm (1981) offers the following five staffing principles:

1. Use better and fewer people.
2. Fit the tasks to the skills and motivation of the people available.
3. Help people self-actualize.
4. Select people who will complement and harmonize with one another.
5. Keep misfits out.

9.5 Improvements in environment

Tools have a positive effect on the productivity of the process. Tools are deployed for forward engineering (such as GUI builders, code generators), reverse

engineering (abstracting to higher-level, e.g. from code to UML), and round-trip engineering (after a change, need to maintain consistency among entire set of artifacts).

9.6 Quality

Software quality is defined as the degree to which a system, component, or process meets specified requirements.

9.7 References

1. Brown A. W., and Wallnau, K. C. The current state of CBSE. IEEE Software, 15(5):37 46, September/October 1998.

2. Cai, X., Lyu, M. R., Wong, K-F., and Ko, R. Component-based software engineering: Technologies, development frameworks, and quality assurance schemes. In Proceedings of the 7th Asia-Pacific Software Engineering Conference (APSEC00), pages 372 379, Washington, DC, USA, 2000. IEEE Computer Society.

3. Li, J., Conradi, R., Slyngstad, O. P. N., Bunse, C., Torchiano, M., and Morisio, M. An empirical study on decision making in off-the-shelf component-based development. In Proceedings of the 28th International Conference on Software Engineering (ICSE06), pages 897 900, New York, NY, USA, 2006. ACM Press.

4. Szyperski, C. Component software: Beyond object-oriented programming (2nd ed.). Harlow, UK: Pearson Education Limited, 2002.