DIPLOMARBEIT

Software Project Management

Prototype of a Software Project Management Tool

Ausgeführt am Institut für

Gestaltungs- und Wirkungsforschung

der Technischen Universität Wien

Unter der Anleitung von

Univ.Doz. Dipl.-Ing. Dr.techn. Johannes Gärtner

durch

Nader Tawil

Kahlenbergerstrasse 66/B3, A-1190 Wien

Wien, Dezember 2003

Abstract

In spite of years of experience in and research on software development, software development still is a troublesome and risky endeavor. In recent years managerial aspects of software development received more attention, while technical aspects remain important.

This thesis deals with some important aspects of managing software development projects and introduces a new software-tool to ease corresponding management. Specifically, it builds upon earlier research results that identified poor planning (i.e. estimation and scheduling) and failure to monitor and react to changes as belonging to the main causes of project failures.

The software intends to ease a number of tasks in this project planning, monitoring and adaptation process:

- Workload estimation and scheduling: A team sets up a project by estimating the workload (not duration) of individual tasks; Tasks can be grouped to define goals and team members can be assigned to tasks. Based on estimates, priorities and team member availability, a scheduling engine creates project schedules; Only then are task durations and probable completion dates visible.

- Automatic estimate and schedule refinement: During a project, progress information entered by team members is used to automatically refine estimates and schedules.

- Monitoring: A metrics collection engine and a graphical interface were developed, allowing data analysis at all project levels and between different projects. The monitoring system can be used to determine and compare a projects status at different times, identify trends and implement an early warning system.

To asses the models usability, as test-cases, two complex real life projects were re-managed and the results compared to actual values. The same projects were then re-managed using commercial tools (MS-Project, Teamflow). The ease and support for project management by these three tools was then compared and showed that the new software delivered better results in these cases.

Abstrakt

Trotz Jahrelanger Erfahrung und Forschung auf dem Gebiet der Softwareentwicklung, bleibt Softwareentwicklung eine beschwerliche und riskante Aufgabe. Forschungsarbeiten der letzten Jahre widmeten den Managementaufgaben viel Aufmerksamkeit, wobei technische Aspekte wichtig blieben.

Diese Diplomarbeit behandelt einige wichtige Aspekte des Management von Softwareentwicklungsprojekte und stellt ein Softwarewerkzeug vor, um die Ausführung solcher Aufgaben zu erleichtern. Diese Arbeit baut auf frühere Forschungsergebnisse auf die, unzureichende Planung (d.h. Aufwandsschätzung und Terminplanung), unzureichende Überwachung und fehlende Reaktion auf Änderungen im Projekt, als einige der Hauptursachen für Projektmisserfolge identifizierten.

Ziel der entwickelten Software ist die Erleichterung einiger dieser Aufgaben in Projektplanung, Überwachung und bei Änderungen im Entwicklungsprozess zu gewährleisten:

- Aufwand Schätzung und Planung: Das Projektteam schätzt den Aufwand (nicht die Dauer) der einzelnen Aufgaben; Aufgaben können zu Ziele gruppiert werden und Teammitglieder können dieser Aufgaben zugeordnet werden. Basierend auf Schätzwerten, Prioritäten und Teammitgliedverfügbarkeit, wird ein Projektterminplan erstellt. Nur dann sind Dauer und vorrausichtlicher Fertigstellungsdatum der einzelnen Aufgaben sichtbar.

- Automatische Schätzung und Zeitplanverfeinerung: Im Laufe eines Projektes werden, durch Teammitglieder eingetragene Fortschrittsinformationen benützt, um die Schätzungen und Terminplanung zu aktualisieren.

- Überwachung: Das entwickelte Metrik-Modul und dazugehörige graphische Schnittstelle ermöglichen Datenanalysen auf allen Projektebenen und zwischen verschiedenen Projekten. Mit dem Überwachungssystem kann ein Projektstatus zu verschiedenen Zeitpunkten bestimmt und mit anderen Projekten verglichen werden, Trends können identifiziert und ein Frühwarnsystem kann implementiert werden.

Um die Brauchbarkeit des Tools zu prüfen, wurden zwei komplexe Projekte mit dem Tool durchsimuliert und die Ergebnisse mit tatsächlichen Werten

verglichen. Weiters, wurden beide Projekte mit kommerzielle Tools simuliert (MS-Project, Teamflow). Die Brauchbarkeit und gebotene Managementaufgabenunterstützung der drei Tools wurden verglichen und es zeigte sich, dass die neue Software bessere Ergebnisse liefert.

Introduction

There is ample evidence that software development still is a risky and cumbersome process. A 1994 survey by the Standish Group found that about two-thirds of all projects substantially overrun their estimates. In his book "Extreme Programming", Kent Beck draws a similar line "*Software development fails to deliver, and fails to deliver value. This failure has huge economic and human impact. We need to find a new way to develop software*".

Project management and software engineering literature are full of examples like these two above. Software development is a complex process, and there are many reasons for failures. Some are dictated by company politics (e.g. "Be the first on the market."), others are related to poor management or technical error.

In order to improve on software development, over the years, a great deal of work was focused on the technical aspects of software development (design, testing, validation…). This led to great technological advances such as structured programming, structured design, formal verification, object-oriented design, …. With a few notable exceptions (e.g., MYTHICAL MAN MONTH), the managerial aspects of software development attracted much less interest. ([McConnell]) provides a possible explanation for this: "*Perhaps this is so because computer scientists believe that management per se is not their business, and the management professionals assume that it is the computer scientists responsibility."*

But this is changing, more attention focuses on project management tasks. This is evident in the new lifecycle models and methodologies being adopted. These models are moving away from traditional, rigid, waterfall like models where a project is divided into different phases and executed in a linear manner. New models focus on handling emerging complexity and surprises while offering maximum flexibility.

[Gärtner] proposes just one of several such new approaches. "The Snake of Project work" Project work does not commence in separated phases but moves in several phases at the same time; the focus shifts, it is always a mix between project design, problem analysis, requirements analysis and implementation (planning). The idea behind this model is to avoid trying to fit ones work to an unrealistic model; it tries to find models that reflects the realities (surprises and constant changes) of modern project work.

Another methodology that focuses on the needs of modern software development is Extreme Programming. "*XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software."* [Beck]. XP assumes that requirements will change, so it makes no attempt to create and follow an initial design. Instead, the application is designed and developed incrementally in a series of brief design-coding-testing iterations. At the end of each iteration, the customer has a working (though not full-featured) product to use.

"*Management fundamentals have at least as large an influence on development schedules as technical fundamentals do… organisations that attempt to put software engineering discipline in place before putting project management discipline in place are doomed to fail.*" [McConnell]. Another very important aspect of software development is speed. Fast advances in hardware development, competition and short product life spans have made the time to market a very important variable in software development. Many authors (e.g. [McConnell], [Beck]) agree that the first step in increasing development speed is by not allowing a single group (managers, customers, developers) to control all three corners of the classic trade of triangle "schedule-cost-product". Dividing control among all groups increases visibility and, since none of the groups have all the information required to decide on all corners, the results are better.

Different authors see different aspects of software project management as most crucial. [Gärtner] sets a strong focus on problem definition. Problems are man made, change when viewed from different perspectives or at different times and should not be left to the first definition by the customer alone or to be considered static. Setting goals after having all concerned parties think about the problem and consider different perspectives makes the project requirements less susceptible to change; But the team has to still expect changes and be ready to change project goals.

[McConnell] sets the main focus on estimation. *"Without an accurate estimate, there is no foundation for effective planning and no support for rapid development".* McConnell found that at the beginning of a project, high and low estimates can differ by a factor of 16. Even after requirements have been completed, estimates can be off by about 50 percent. Estimates have to be refined regularly and project plans updated accordingly.

A 1991 study by Michiel van Genuchten showed that the majority of causes of schedule slips and cost overruns are either related to project planning and monitoring or can be anticipated by management. [Kulik] suggests

implementing an early warning system that will aid in identifying circumstances that could delay a project.

In addition to estimation, [McConnell] concludes "*Peopleware issues have more impact on software productivity and software quality than any other factor.*" He backs this statement with studies that show a 10 to 1 difference in productivity between different developers and 5 to 1 between different teams. The reasons being, skill levels, motivations, good working environment, …

Although different authors have the main focus set in different areas, they all agree on the importance of allowing and reacting to changes. Agile methodologies are becoming more and more popular. A web based survey (November 2002 to January 2003) conducted by Shine Technologies to investigate market interest in Agile methodologies concluded that adopting an agile method increased productivity for 88% of the participants and improved quality for 84%.

But expecting and allowing change alone is not enough to be successful. The impact and consequences of changes has to be evaluated carefully and all approved changes have to be planned accurately.

The goal of this thesis is to discuss some of the problems and requirements facing modern software development and to develop a model and a corresponding tool prototype to aid in developing and refining plans. The focus was set on estimation, automatic estimate and schedule refinement and monitoring.

The development of the software and the research were conducted in parallel. A first prototype fulfilling minimal requirements (task list with progress information) was developed using MS Excel and gradually refined and expanded with the research conducted. The final prototype was developed in MS Visual Basic and used a MS Access database for data storage.

The research part of the thesis started by discussing "Software engineering", "Project" and "Project management"; and then went on to look at the different lifecycle models available, starting with classical models like "Waterfall" and also looking at modern agile models and methodologies like "XP.

After discussing requirements of modern software development, several important issues were identified:

Workload estimation: A team sets up a project by estimating the workload (not duration) of individual tasks; Tasks can be grouped to define goals and team

members can be assigned to tasks. Based on the estimates, priorities and team member availability, a scheduling engine creates project schedules; Only then are task durations and probable completion dates visible.

Automatic estimate and schedule refinement: During a project, progress information entered by the project team is used to automatically refine estimates and schedules.

Monitoring: A metrics collection engine and a graphical interface were developed. Data analysis is possible at all project levels and between different projects. The monitoring system can be used to determine and compare a projects status at different times, identify trends and implement an early warning system.

To asses the usability of this approach, two real life projects were re-managed with the tool and the results compared to actual values. Then the same projects were re-managed using commercial tools and the results compared again. Using the tools features, the process of management was eased and delays could be identified more easily. Furthermore estimations of delays were much more accurate.

The thesis is divided into seven chapters:

Chapter 1: discusses different definitions of the terms “Software engineering”, “Project”, “Project management”, sets the definitions that will be used in the rest of this thesis and looks at different categories of project management tools available on the market.

Chapter 2: looks at the different life cycle models. The chapter starts of by looking at classical models (e.g. Waterfall) and then goes to modern agile models and methodologies like XP. The different models are discussed and compared to determine their advantages, disadvantages and which project types they may support.

Chapter 3: reviews project management literature to put together a list of the main requirements facing modern software project management. Requirements were grouped in the following categories:

- Project management fundamentals
- Estimation and scheduling requirements
- Monitoring and forecasting requirements

- Controlling requirement
- Lifecycle requirements
- People requirements

Secondary requirements

Chapter 4: Based on the requirements set in chapter 3, a software project management approach is developed.

Chapter 5: describes the tool prototype developed to support the project management approach.

Chapter 6: tests the approach and prototype using two real life projects and compares the results to actual project values and results obtained using commercial systems.

Chapter 7: reviews test results to determine whether the requirements set in chapter 3 were fulfilled, presents recommendations for improvement and looks at further research areas.

Chapter 1 – Working Definitions

Some are of the opinion that almost any non-repetitive task can be classified as a project, making the discipline of project management as old as mankind itself, while others argue that project management as a discipline has only existed since the Apollo projects. However, there is a big difference between carrying out a very simple project which involves a few people, and projects that involve a large mix of people from different organisations. Before going into the details of software project management, it is important to understand the terms “project”, “project management” and “software engineering”. To better understand these terms, we will be looking at and discussing different definitions. At the end of the chapter we will present working definitions that will be valid throughout this thesis.

Some views on software engineering

[IEEE Standard Glossary]

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Similar definitions are presented by [Dumke],

> *The software-development process is the process that comprises defining, planning, implementing and evaluating a software-/hardware application including the applied methods / tools and the required personnel.*

and [Mahnke]

> ***Software engineering*** *emerged at the end of the 60s beginning of the 70s and means nothing more than applying engineering procedures to the software development process.*
>
> ***Engineering procedures*** *means, dividing up a task into many single steps. Each of these single steps is solved by a specialist who uses his own methods and tools.*

[Kimm] narrows down the definition by adding two conditions:

- *more than one person is involved with the creation and/or use of the program and*

- *more than one version of the program will be developed*

The first condition limits software engineering to large projects; creating smaller systems that can be produced by a single person is regarded as programming and not engineering. The second condition adds more confusion to the definition and raises many questions; what about systems whose first version fulfils all requirements and functions without errors? What about projects that are cancelled before their first release?

[Boehm] provides a definition of software engineering based on the definitions of "*software"* and "*engineering" as* given in *Webster's New Intercollegiate Dictionary [Webster, 1979]*

> ***Software****: is the entire set of programs, procedures, and related documentation associated with a system and especially a computer system.*
>
> ***Engineering****: is the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to man in structures, machines, products, systems and process.*
>
> *Since the properties of matter and sources of energy over which software has control are embodied in the capabilities of computer equipment, we can combine the two definitions above as follows:*
>
> ***Software engineering:*** *is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation.*

[Boehm]'s definition includes a key point not found in other definitions, one encapsulated by his phrase "useful to man". This places the responsibility on software engineers to apply a software engineering methodology that addresses the social implications of computer systems and not simply to accept any set of requirements without questioning their usefulness.

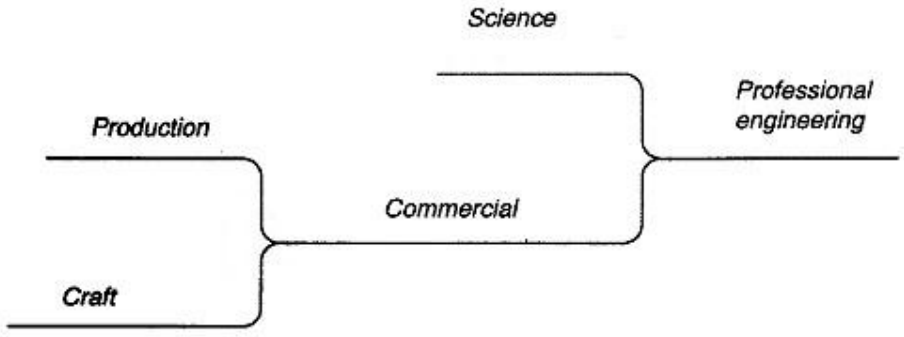
[Shaw] summarises the common clauses of different definitions of software engineering:

Creating cost effective solutions…	*Engineering isn't just about solving problems; it's about solving problems with economical use of all resources, including money.*

… to practical problems…	*Engineering deals with practical problems whose solutions matter to people outside the engineering domain: the customer*
… by applying scientific knowledge…	*Engineering solves problems in a particular way: by applying science, mathematics, and design analysis*
… building things…	*Engineering emphasizes the solutions, which are usually tangible artefacts.*
… in the service of mankind.	*Engineering not only serves the immediate customer, but also develops technology and expertise that will support the society.*

Evolution of an engineering discipline

Before an activity can be referred to as a discipline, it undergoes an evolution as described by [Shaw].

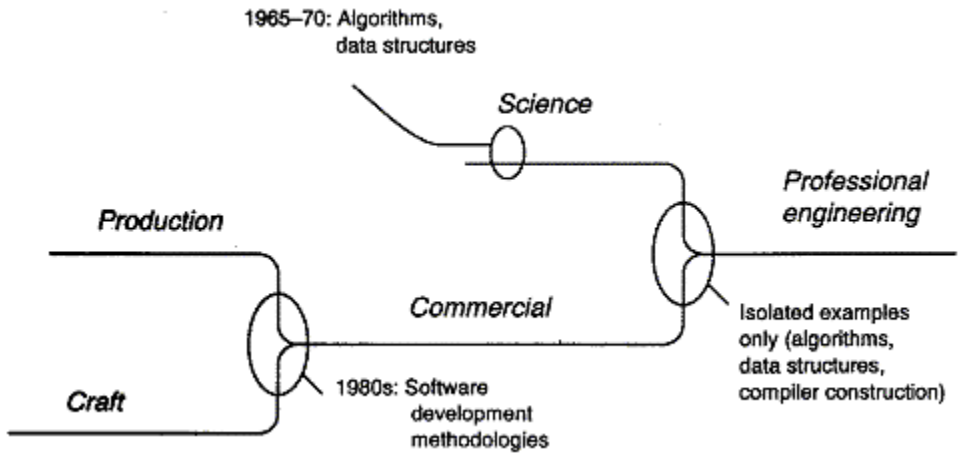


Historically, engineering has emerged from ad hoc practice in two stages. First, management and production techniques enable routine production. Later, the problems of routine production stimulate the development of a supporting science; the mature science eventually merges with established practice to yield professional engineering practices. The lower lines track the technology, and the upper lines show how the entry of production skills and scientific knowledge contribute new capability to the engineering practice.

Exploitation of a technology begins with talented craftsmen inventing (and reinventing) solutions to a set of problems. Progress at this stage is disorganized, transmission of knowledge is slow, resources are used extravagantly and manufacture is for personal use.

At some point, the technology becomes widely accepted, and demand exceeds supply. At this point, attempts are made to define standard procedures, train practitioners to use these procedures, and define the necessary resources and capital for systematic commercial manufacture.

In the final step of evolution, the development of a corresponding science is stimulated by problems of current practice. At some point the science becomes sufficiently mature to turn into a significant contributor to the commercial practice. This marks the emergence of engineering practice.



Example 1: Algorithms and data structures

Early 1960's: Algorithms and data structures were created as part of each program; information about good solutions was transmitted informally.

Mid 1960's: good programmers shared the intuition that if you get the data structures right, subsequently, the rest of the program becomes much simpler.

Late 1960's: Algorithms and data structures began to be abstracted from individual programs, and their essential properties were described and analysed.

The 1970's: progress in supporting theories, performance and correctness analysis.

Early 1980's: sound theory and language support became available.

Example 2: Compilers

Mid 1960's: Formal syntax was first used systematically for Algol 60, first parser generators developed.

1970's: Semantics and types theories developed.

1980's: significant progress toward the automation of compiler construction.

In these examples it took a good twenty years from the time that work started on a theory to the time that it provided serious assistance to routine practice. Considering that the whole field of computing spans about 45 years, many theories are gradually evolving and we can eventually expect the emergence of a software engineering discipline.

Example 3: Software Methodologies [Trauring]

1950's: First "high level" languages appeared (Fortran, Cobol)

03/1968: Dr. Dijkstra published a letter to the editor of the *Communications of the Association for Computing Machinery (CACM)* called *Go To Statement Considered Harmful*. Dijkstra's ideas formed the basis of structured programming (top-down development, using formal programming constructs, formal steps to decompose a large program), which was the first software development methodology.

1971: Professor Nicklaus Wirth released the programming language Pascal, which did not provide a "goto" statement and had control structures to support the Dijkstra structured programming paradigm.

1970: A standard development "life cycle" model for software systems, called "waterfall model", was first publicly documented.

Early 1970's: Structured programming lead to structured design and analysis, seen by [Trauring] as the start of the software engineering discipline

1975: Fred Brooks published his book “The Mythical Man-Month” wherein he described software development as a human centric process, not an engineering discipline.

1980’s: two interesting ideas in software development methods emerge

1. Computer aided software tools: complex notations and processes for modelling software systems were developed. These methods were used in combination with computer aided software engineering (CASE) tools.

2. Object Oriented software development: OO origins were in the 70’s (Smaltalk); the goal was to make the world of software much closer to the real world. Objects of various sorts communicate with each other by sending and receiving messages without knowing anything about the internal workings of each other.

Around 1983: US Department of Defence developed the next generation Pascal with OO aspects called Ada and Bjarne Stroustrup of Bell Labs created a next generation C with OO aspects called C++.

Late 1990’s Brooks’ ideas about managing the people process were taken up by the proponents of agile programming methods.

1997: The Object Management Group (OMG) released the first Unified Modelling Language (UML) standard; a standardized notation for modelling software systems. UML is a notation, not a method, that can be used as a universal language to communicate designs and ideas about software systems, no matter which method is used.

Some views on project

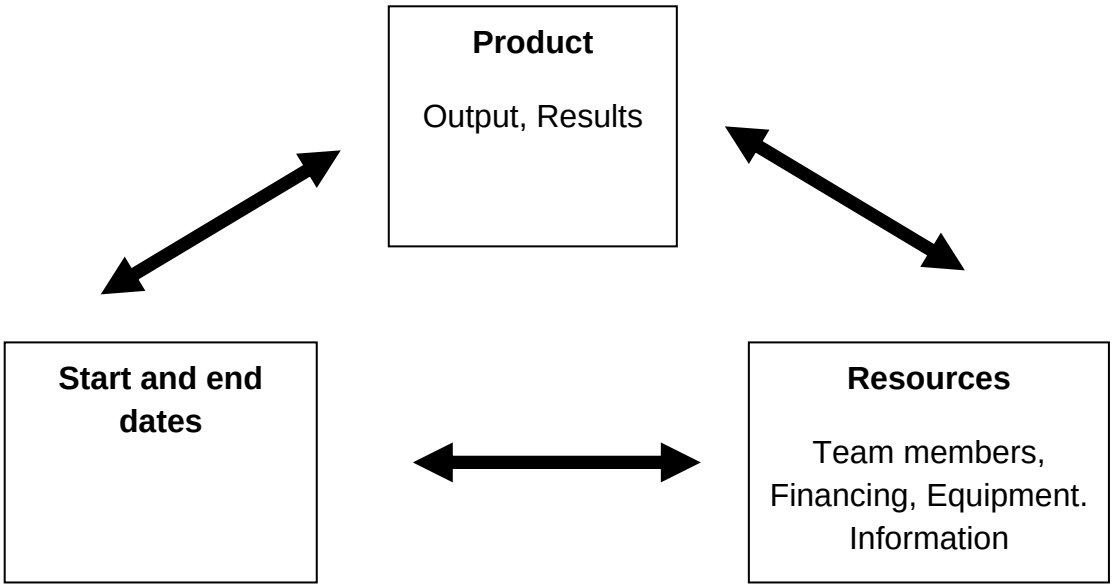
There are numerous definitions for the term “project”; most of them share a common base but differ in terms of detail. The simplest definition is presented by [Portney]

What is a project? A project has the following characteristics:

- *A predetermined output: products or results*
- *A predetermined start and end date*
- *Predetermined resources: budget, team size, equipment…*

[Portny] also presents a graphical view of projects. The diagram clearly shows the dependencies among the three characteristics. Each characteristic is

constrained by the other two. Product size and quality are dependent on the project's timeframe and available resources (budget, number and experience of developers). In addition to being dependent on each other, the characteristics have to be balanced out; e.g. one cannot raise the quality or feature set of a product by increasing the number of team members and keeping the timeframe short; 9 programmers working 1 week produce less output than 3 programmers in 3 weeks; the increased communication overheads, required equipment, waiting times due to dependencies between tasks, etc. have to be considered.



Another simple definition of project is given by [Project management today]:

> *In essence a project can be captured on paper with a few simple elements: a start date, an end date, the tasks that have to be carried out and when they should be finished, and some idea of the resources (people, machines etc) that will be needed during the course of the project.*

This definition builds on the previous one by adding a little more detail in terms of tasks and resource assignment. Although these definitions appear simple, they only apply to small-scaled projects executed by small teams.

Other definitions describe the term "project" in more detail, incorporating the three basic characteristics of output (goals), timeframe and resources.

E.g. [Dumke]:

> *Software project: is a process to develop a specific software using the necessary resources; a project has the following characteristics:*
>
> 1. *unique endeavour*
> 2. *limited by a deadline*
> 3. *has clear goals and validation criteria*
> 4. *Requires cooperation of experts of different fields*
> 5. *Usually requires finding solutions to new and unexplored problems*
> 6. *Limited by a budget*
> 7. *Subject to high risk*

In addition to having clear goals, [Dumke] adds validation criteria to the definition. Since a project team can consist of people from different fields, defining validation criteria helps make the goal's definitions clearer and leaves little room for misinterpretation by the different team members.

Dumke's opinion that a project "usually requires finding solutions to new and unexplored problems." requires further consideration. The term "*unexplored problems"* should not be included in a general project definition. Finding solutions to unexplored problems requires a lot of research; the results (goals) cannot be clearly defined and making size, time and cost estimations is very hard, and in some cases, even impossible.

Another definition is presented in [Tjoa]. In addition to the 7 characteristics above, [Tjoa] includes the following 8th characteristic:

> 8. *A project utilises different methods to break down the complexity, better understand and to present the problem*

Projects can become very complex; using different (project specific) standardised methods and procedures can greatly increase the odds of success of a project.

In his lecture notes to "Project management" [Merkl] adds a 9th characteristic:

> 9. *The project manager and team members are under high pressure.*

This characteristic is, unfortunately, true for many projects, but should not be included in the definition of a project. Being under high pressure is, in most cases, due to a lack of planning, estimation, monitoring and control. One of the

goals of these activities is to have a predictable and balanced schedule that avoids overworking and “burning out” team members.

The [project management institute] uses the following definition:

> *A project is an endeavour undertaken to achieve a particular aim. Every project has a definite beginning and a definite end. They are performed by people, constrained by limited resources, have to be planned, executed and controlled, are temporary and unique. They are created at all levels of an organization, may involve a single person or thousands. Their time spans vary greatly. They may involve a single department of one organisation or cross organisational boundaries.*

The focus here, due to limited resources, is on the need to plan and control. This requirement becomes even more important as the team size increases and the team members come from different departments or organisations. Some may argue that *planning and control* are activities that should be included in the definition of project management, but I think the uniqueness of projects and the variety of people involved in different projects make planning and control crucial to the success of a project and should be included in the definition “project”.

[Lutz] provides a general project definition that focuses on the uniqueness and limitations of other projects structures.

> A project is an intention that is unique in one or more of the following characteristics:
>
> - *Goal specification*
> - *Temporal, financial, personnel or other limitations*
> - *Limitations with respect to other projects*
> - *Project specific organization (structure and flow organization).*

[Lutz] also provides a less than satisfactory definition of Informatics projects.

> *The purpose of Informatics projects is the procurement, adaptation, evaluation, maintenance etc. of Informatics resources to support / enable communication and generating information in organizations in the economic and administration sectors. Informatics projects are characterised by the following features:*

- *Relatively short project running time (ideally few weeks to maximum a year)*
- *Medium to large project groups*
- *Interfaces to other projects or components of the information infrastructure (e.g. existing information systems) to avoid having an isolated view of the project.*
- *Clear and complete definition of project goals dictated by the organisations goals, thus leaving little room for interpretation*
- *High risk, since there is no guarantee that the planned goals can be achieved.*
- *Uncertainties regarding the compliance with deadlines and budget*
- *Competition for resources with other projects*

This definition fails to mention the creation of new systems and only concentrates on generating and passing information, ignoring control, monitoring, automation and many other types of systems.

Looking at the features more closely, we also notice many other omissions:

- *"Relatively short project running time (ideally few weeks to maximum a year)."* Running times of a few weeks to one year apply to small projects; the majority of informatics projects can be classified as medium to large and can have running times of several years.
- *"Clear and complete definition of project goals dictated by the organisations goals, thus leaving little room for interpretation."* It is desirable to have **no** room for interpretation; Different interpretations produce different solutions (systems), of which, in most instances, only one can be correct.
- *"High risk, since there is no guarantee that the planned goals can be achieved."* Informatics projects, like other types of projects, are subject to high risk, but they are not adventures and have to guarantee a certain degree of success.
- *"Uncertainties regarding the compliance with deadlines and budget; Competition for resources with other projects."* The last two aspects,

as with Lutz's point about no guarantees, make Informatics projects sound like great adventures with an unknown outcome.

This definition describes the majority of today's projects. Amongst the many reasons for this definition are vague project goals, bad planning and inexperienced project managers and teams. The reasons and possible solutions will be discussed in more detail later in this thesis.

All the definitions considered until now have one thing in common. They all imply that a project's goals, resources and schedules can, and must, be clearly defined before the start of the project. A different approach to projects is presented in [Gärtner]:

> *The main difference to other approaches is that goals are not fixed. – Nothing is fixed.*
>
> *The underlying model identifies 4 characteristics:*
>
> 1. *Problem*
> 2. *Goals*
> 3. *Project structure*
> 4. *Procedures*
>
> *These 4 characteristics are connected, start of as being blurred and change as the focus moves. The model uses a stepwise approach to better understand the relations, and based on the results, the project characteristics are defined/refined or redefined.*

We will be considering and comparing this approach and the underlying model in more detail in the forthcoming chapters.

Some views on project management

Project management is the process that accompanies a project from start to finish with the goal of ensuring a successful end to the project.

The different definitions encountered in literature ([Biffl], [Lutz]) share three common tasks:

> **Planning:** involves estimating size, cost, resources and deadlines; the result of planning is a list of goals/tasks, their deadlines and the assigned resources. It is the project team's responsibility to ensure that the plan is followed and, when not possible, to have plans adjusted or changed and approved.

Monitoring: assessing the project's progress; this can be done through regular meetings, project journals, reports, reviews and etc.

Controlling: Depending on results produced by monitoring, measures are developed and implemented.

[Lutz's] definition limits the three tasks above to "*cross-organisational multidisciplinary informatics projects*", ignoring smaller projects that are performed by a single person or within a department and larger projects that are carried out across different departments within an organisation.

[Ricketts] sees management as *a continuous cycle consisting of three activities: planning, execution and monitoring.* Execution is described as "*following a recipe"*, i.e. performing actual project implementation. Execution is a task performed by the project's team members and should therefore not be included in a definition of project management.

[Versteegen's] definition adds "Configuration management" to the common definition. This is an important factor to add to the equation when talking about software projects since today's software systems can consist of hundreds of different components, developed by different teams or companies. Trying to manage different versions or releases without proper configuration management could become an impossible task.

Project management spans the following management technologies:

- *Requirement management*
- *Configuration management*
- *Risk management*
- *Change management*

[Portney] identifies *"Organization: Defining the rolls and responsibilities of the team members"* as one of the basic project management activities. However, the roles, (of the project manager, team leader, lead programmer, customer, and etc.) definitions and responsibilities are, and should be, standardised within a company and not redefined for each project. In some cases, adjusting the definitions to certain projects might be necessary, but not a complete redefining of definitions.

[PMToday] sees project management as "not only about planning but also about human attributes like leadership and motivation, and answering 'what if?' questions."

Answering *"what if?"* questions is important when planning or reacting to changes in project plans. However, to be successful, project management must also be able to supply answers to the following five questions at all times:

1. *Where are we now?*
2. *When will we be done?*
3. *Are we taking the right actions now to complete the project successfully?*
4. *How much have we spent so far?*
5. *How much do we need to spend?*

Project management as described by [Gärtner] focuses on the changing nature of software projects and includes problem definition (and not only the finding of solutions) as a management task:

> *A stepwise refinement of the 4 characteristics: Problem, Goals, Project structure and Procedures. The first step is identifying, if one exists, the problem. While doing this, the perspectives of all involved parties must be considered.*
>
> *This is done by continuously shifting the focus of the project work between the actual situation, the planned situation and the implementation.*

Project management tools

Project management tools have the sole purpose of making the project team's life easier. This may seem obvious, but is not true for many systems. Software projects are challenging and fast-changing, a software tool has to aid a project manager in solving the numerous challenges presented by a software project.

Project management tools have to provide support in the following areas:

- Building a project in a structured manner
- Estimating (size / cost / resources) & Scheduling
- Integration of multiple projects
- Monitoring, forecasting and change management
- Support for different life cycle models

What is on the market?

All project management systems claim to excel in most of the points listed above. Some excel in a few, some in many and some in none. If considered from a project structure point of view, the systems on the market can be divided into 3 categories:

- Flowchart systems: Allow you to build projects plans/schedules using flowcharts.
- Hierarchal systems: Use a hierarchal goal – sub goal structure.
- Timeline systems: Allow the user to define project plans along a timeline.

Throughout the thesis we will be analysing two systems representing the three categories. **Microsoft project 2000**, a very well known and widespread system, represents the timeline and hierarchal categories and **TeamFlow7** represents the flowchart, as well as hierarchal, systems.

Working Definitions

Out of the differing views discussed above, we consider the following definitions to be appropriate for our purposes.

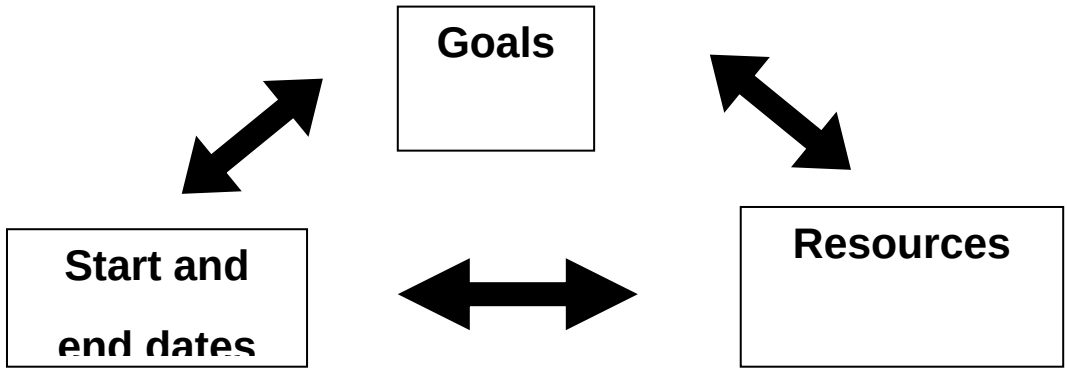
Project

A project can be defined by three primary characteristics:

1. **Goals:** Goals are not fixed; they can change, be cancelled or new goals can be defined. All team members have to understand and agree on what the current project goals are and are not. Goal status must be verifiable throughout all stages of the project. To ease the verification process, team members have to define and agree on validation criteria and procedures.

2. **Start and end dates:** The project team has to complete goals by pre-defined dates. Like goals, start and end dates can be changed during the project, but require the approval of all those concerned.

3. **Resources:** A project has access to a limited amount of resources with which to achieve its goals. Resources include financing (budget), personnel, equipment (hardware, software, offices) and etc. Resources can also, with the consent of all parties involved, be changed.



Together, these three characteristics form the basis of a project. None of them can be defined / set without considering the other two. The customer(s) and project team have to define / set and balance out these three primary characteristics before the start of a project. A project that commences without a balanced basis has little chance of success. However, starting with a balanced basis is not a guarantee to success; these three factors have to be monitored and controlled / adjusted throughout the whole project.

Other project characteristics include:

- Projects are unique and non-repetitive
- Projects are **complex** and **require different methods and tools** to break down the complexity
- **A project requires close co-operation between its team members.** A project can be created at all levels of an organization, may involve a single person or thousands, a single department of one organization or may be carried out across organizational boundaries.
- **Projects are subject to high risks.** Balancing out the three primary factors helps reduce the risks; to be successful, a project has to address and monitor risks in all stages of its development.
- **Projects do not find solutions to unexplored problems.** Unexplored problems are researched in research projects, a project sub-type.

Project management

Project management is the process that accompanies a project through all its phases with the goal of ensuring a successful end. What the individual phases are and how many iterations are performed is different from project to project and is dictated by the lifecycle model chosen.

Before listing the different tasks involved in project management, let us take a closer look at the term “successful end”. When can a project be rated as “successful”? Most people would say “If I achieve the project goals, then I can rate the project as a success.” However, the answers are as numerous as the perspectives are.

- **Project manager**: Stay within budget, deliver by planned date and implement what is in the requirements documentation.
- **Team leader:** Develop a high quality system.

- **Developer:** Complete the project with a minimum of extra hours (nil if possible) and be given a certain degree of freedom in design and implementation.
- **Marketing manager:** Have the newest and fanciest system on the market.
- **End user:** Have a system that will make life easier, automate routine tasks, reduce error frequency.

However, having so many solutions to one question is usually not desirable. So projects end up being rated according to which goals were achieved, and to what degree.

Project management tasks:

- **Identifying and setting goals:** Since the project's success or failure is determined by the goals achieved, it is very important to have the concerns and requirements of all involved parties integrated into the goal identification and setting process. This also has to be done when goals are changed or new goals are defined.
- **Planning**: Involves defining required tasks, estimating size and assigning resources. The results of planning is a schedule which states who does what, by when. Since we defined project goals as not being fixed, so the same goes for the project plan. The plan can and will, with the approval of all involved, be changed as needed. It is the project team's responsibility to ensure that the project runs according to plan.
- **Monitoring**: Assessing project status and identifying risks.
- **Forecasting:** In addition to assessing project status, project management must also forecast completion dates and cost expectations. The data has to be forecast on all levels of a project (goal, sub-goal level, individual tasks and individual resources /team members).
- **Controlling**: Depending on monitoring and forecasting results, re-planning is triggered, measures are developed, contingency plans are implemented, and so on.
- **Reporting**: A project manager must always be able to report a project's status and supply answers to questions such as:
    - What are the current project goals?
    - Where are we now? When will we be done?
    - How much have we spent so far? How much do we need to spend?

  These questions have to be answered at all project levels.

Chapter 2 - Software Lifecycle Models

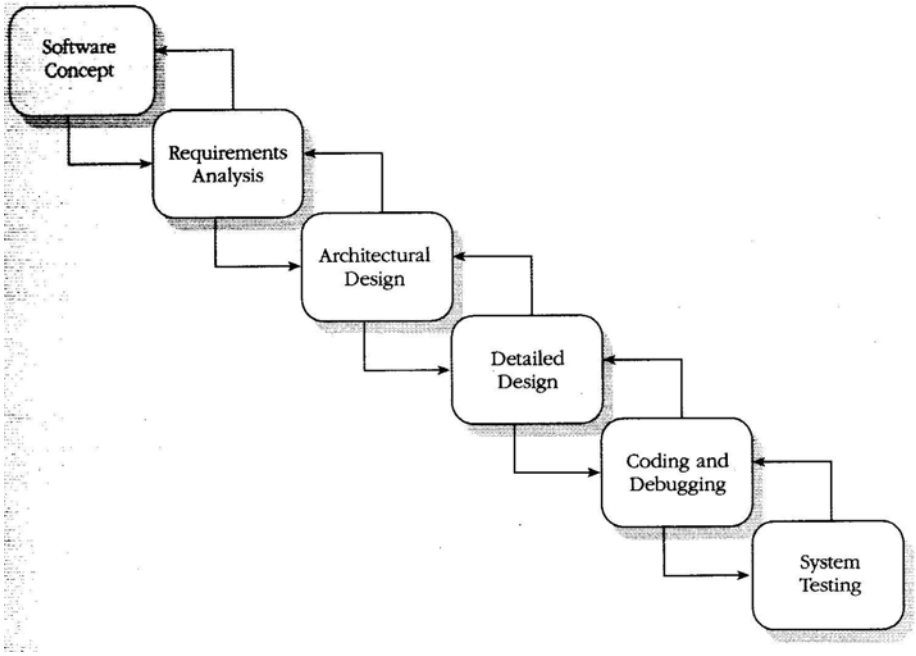
A software lifecycle model defines a framework for project management activities. It establishes the order in which a project goes through different stages (requirements, specification, design, implementation, testing, etc.) and defines criteria used to determine whether to proceed from one stage to the next. Choosing the right lifecycle model can have a substantial impact on the course of a project; the wrong model can be a constant source of slow progress, repeated work and frustration.

Different lifecycle models support different types of projects and place different demands on the project management system used.

In the previous chapter, we defined the terms “engineering”, “project” and “project management”. I will be considering these definitions in my discussion of different lifecycle models in order to establish what types of projects are supported. At the end of the chapter, I shall choose a model which will then be focused on in the rest of the thesis.

Waterfall

The waterfall model is probably the oldest model available. In the waterfall model, a project progresses through a sequence of steps from the initial system concept through to system testing. The results of the phases are summarized in a document. A review of the documentation determines whether or not the project is ready to move on to its next phase. The individual phases of the waterfall model do not overlap; if review determines that the project is not ready to move on, it stays in the current phase.



The waterfall model has been criticized by many as being too rigid. Its linear structure does not support backing up which enables the fixing of problems created in earlier, already completed, stages. This inflexibility means that the team is required to supply a complete requirements specification at the beginning of the project. [McConnell] sees this a bit differently; that is, backing up is allowed but it is very difficult and *"the effort might kill you"*. This is probably why most people agree that backing up is not possible.

Another major criticism of the waterfall model is due to its low visibility. The documents produced at the end of each phase are too technical for most end users and customers. They only get to see the product at a very late stage where subsequent changes in requirements are very expensive to implement.

[Merkl] Another problem, shown by research, is that real world projects do not (can not) adhere to the model.

[William] identifies another problem, "*Analysis Paralysis*", which can result from applying the waterfall mode. Analysis Paralysis occurs when the goal is to achieve perfection and completeness of a phase, resulting in detailed documents and models of little use to downstream processes.

[McConnell] also lists the advantages and supported project types of the waterfall model:

- Doing all planning upfront, reduces the planning overhead.

- Works well for technically weak or inexperienced staff because it provides the project with a structure that helps to minimize wasted effort.
- Performs well for products with a stable definition. E.g. Maintenance release for a product.

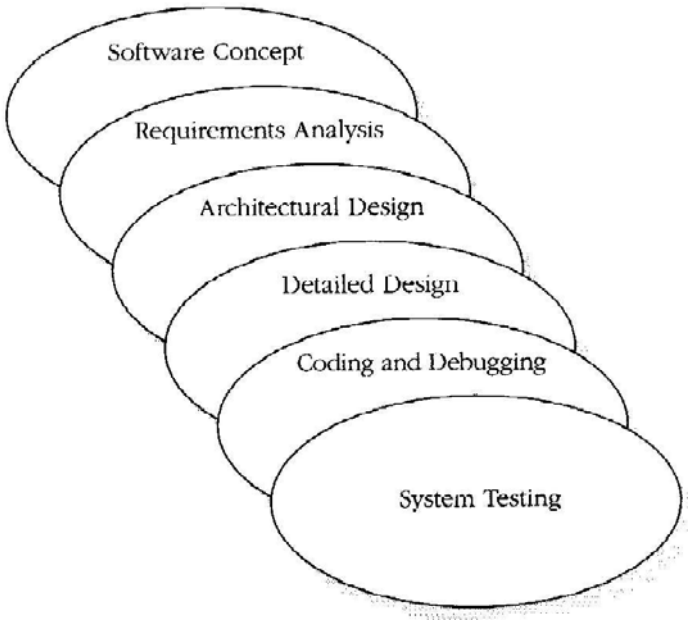
In chapter 1, the term “project” described the characteristic goals, completion dates and resources as being NOT fixed and allowed to change at any stage of the project. The waterfall model, even though it allows backing up, does not supply the needed support for those types of projects.

Modified Waterfall

Most of the weaknesses in the pure waterfall model arise not from problems with the activities but from treating the activities as disjointed, sequential phases [McConnell]. In order to overcome these limitations, modified versions have been developed.

Sashimi

DeGrace tries to overcome the sequential problem by allowing a stronger degree of overlapping between the phases than is usual in the traditional waterfall model. For example, one might be well into architectural design and partially into detailed design before one considers the requirements analysis to be complete.



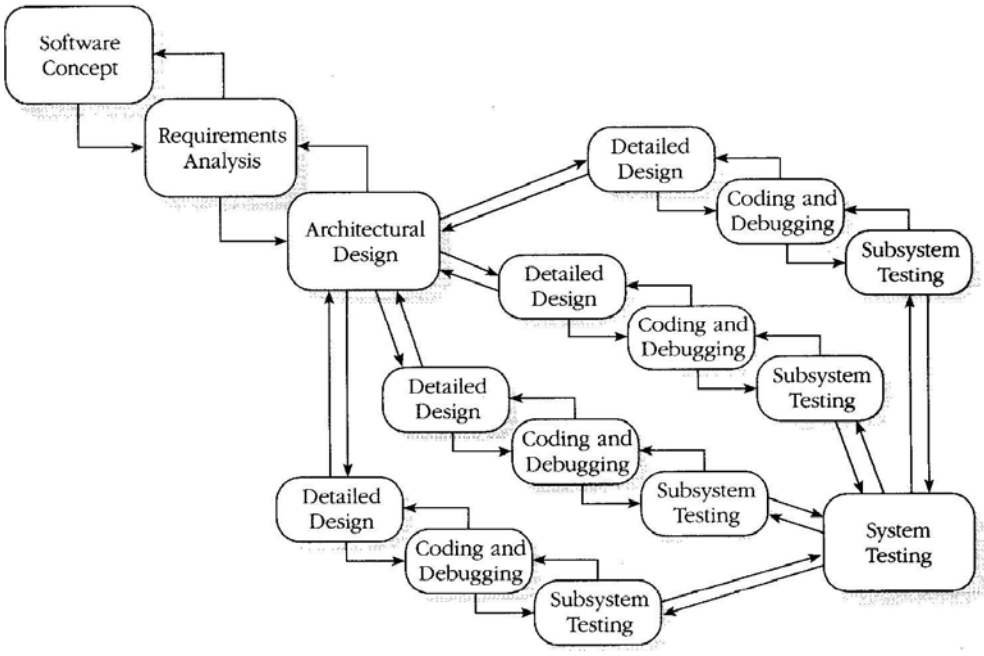
[McConnell] sees the modified approach as reasonable for “*projects which tend to gain important insights into what they’re doing as they move through their*

development cycles" but also warns of the problems that come with overlapping stages: *"milestones are more ambiguous, and its harder to track progress accurately. Miscommunication, mistaken assumptions and inefficiency can also result from performing activities in parallel."*

The Sashimi model provides more support to projects as defined in Chapter 1 than the pure waterfall model does, but it does not solve the linearity problem. Finding architectural design flaws or having to change certain requirements while the project is well into coding and debugging can lead to problems which might be very hard to solve.

Waterfall with Subprojects

[McConnell] identifies another problem with the classic waterfall model: "*because you are supposed to complete each phase before starting the next, system areas that are easy to implement are delayed*." [McConnell] tries to reduce the delays by breaking up the project into subprojects.



In the architecture phase, the system is broken up into smaller, logically independent, subsystems. Each subproject can be completed at its own pace by an independent team.

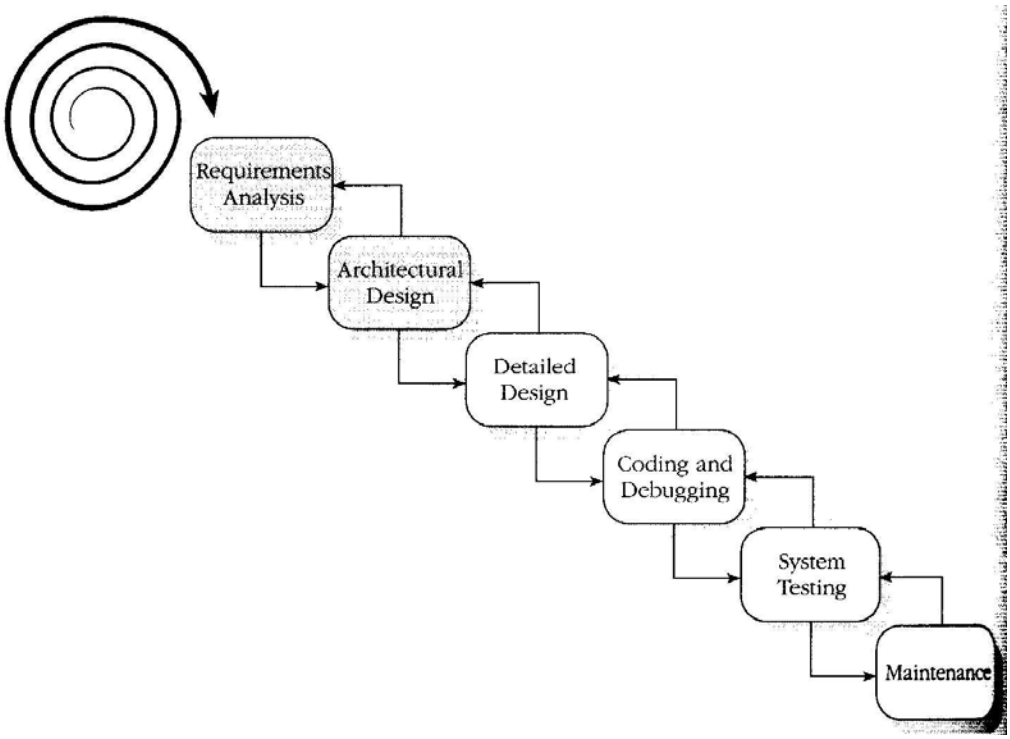
[McConnell] warns of unforeseen interdependencies when using this approach. This can be overcome by eliminating dependencies at the architecture stage / phase. He also suggests defining the subprojects after the detailed design phase. I think, however, due to the high level of effort required by detailed

design, that this would also greatly reduce the desired effect of minimizing delays.

Again, as with Sashimi, the problem of linearity is not solved.

Waterfall with Risk Reduction

The last modified waterfall we will examine tries to minimize the risks which arise from the necessity to fully define requirements before architectural design. No matter how well requirements analysis is performed, architectural design reveals new insights and interdependencies which cannot always be captured in a requirements gathering procedure.



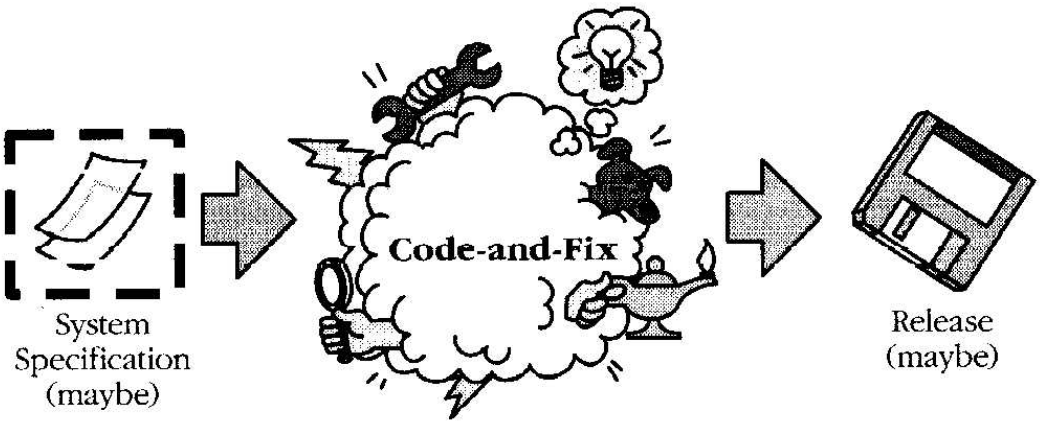
[McConnell] introduced a risk reduction spiral for requirements analysis and architectural design. This spiral can be used to reduce risks in other phases as well.

The risk reduction spiral presented here better suits the changing nature of goals, completion dates and resources as stated in Chapter 1. However, having it only for the first 2 stages is not always sufficient. High risk projects require such a spiral in all stages of development.

Code and Fix

[McConnell] “If you haven’t explicitly chosen another lifecycle model, you’re probably using code & fix by default”. Even though the latter is not really a model, it is used by many teams today.

The Code and Fix model eliminates phases such as requirements, specification, design, testing. Team members usually have an idea of what is needed and go straight into coding and debugging and hope to have a system before the project is officially cancelled.



[McConnell], who strongly discourages adopting the Code and Fix approach, identifies three main advantages:

1. **No overhead**: Since most phases, documents, quality assurance and other activities are eliminated, the overhead is kept at a minimum.
2. **Requires little expertise**: Team members do not have to understand the different stages in software development. All they have to do is write code and it does not have to be efficient, bugfree (i.e. bug-free) or fulfill any requirements or standards.
3. **Visibility**: Since the team jumps straight into coding, they have some concrete / tangible results to show soon after starting the project.

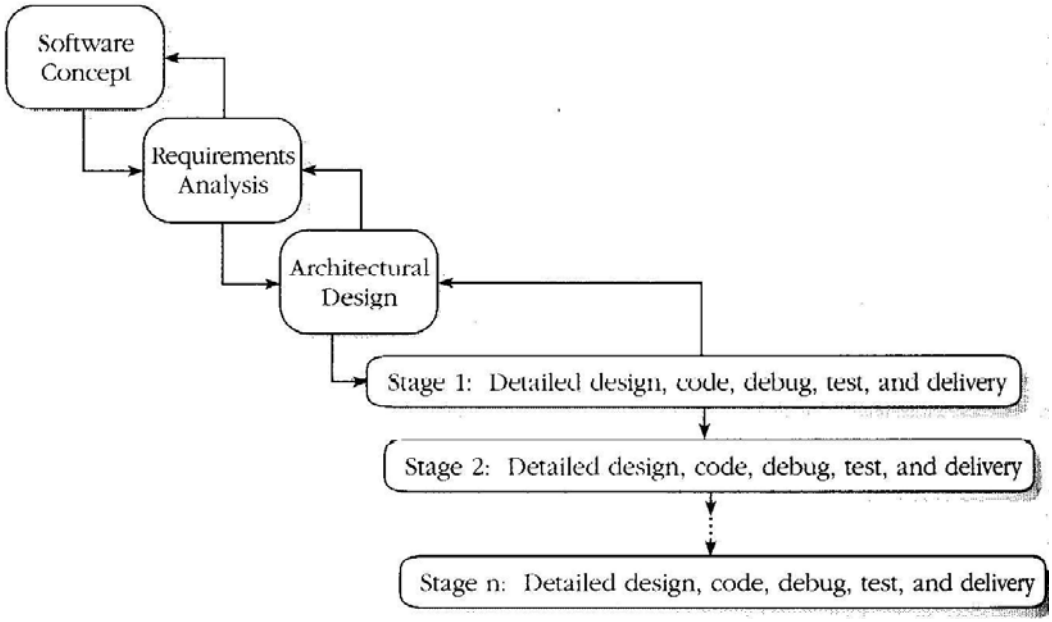
I would only agree with the second advantage as cited by McConnell, that of “requir[ing] little expertise”. “No overhead” may seem an advantage at the beginning, but the overhead incurred by mistakes discovered later on will far outweigh the overhead induced by a structured approach. The third advantage, visibility, is in most cases only an illusion.

Code and Fix supplies no mechanism for assessing quality, cost, schedule or other important variables. Correcting mistakes is very costly. This approach is

dangerous and should not be used, with the exception of for small-scaled projects such as throwaway prototyping or concept demos.

Staged delivery

The staged delivery model starts off as the waterfall model does. The project goes through requirements analysis and architectural design for the whole product. After completing the architectural design, the system is developed in successive stages. At each stage, the system undergoes detailed design, coding, debugging, testing and is completed with the delivery of a working product. The decision, whether or not to go on to the next stage is made at the end of each consecutive stage.



In his book *Software Project Survival Guide*, [McConnell] recommends staged delivery as one of the best ways of organizing a project. The benefits provided are:

- Critical functionality is available earlier: The stages are designed to deliver critical functionality first, which can be a valuable approach in a project with a tight schedule.
- Risks are reduced early: Delivering in stages forces integration to occur more often, thus reducing technical risks. Requirements risk is reduced by putting software early into the users' hands. Management risk is reduced by presenting tangible signs of progress. Planning risk is reduced by revising plans at the end of each stage
- Problems become evident early: Short stages and frequent releases make problem areas visible early.

- Status reporting overhead is reduced: Since the stage release itself is a status report, developers do not have to spend time creating lengthy progress reports.
- Staged delivery makes more options available: Planning to release the software at the end of each stage does not mean that the software definitely has to be released then.
- Possibility of estimation error is reduced: Estimating smaller systems is easier than one large system. Errors in early stages can provide vital information used in successive estimations.

[Merkl] describes staged delivery as "*growing, not building, software*" and argues that staged delivery produces leaner software that is not cluttered with too much functionality, and that is not required by end users.

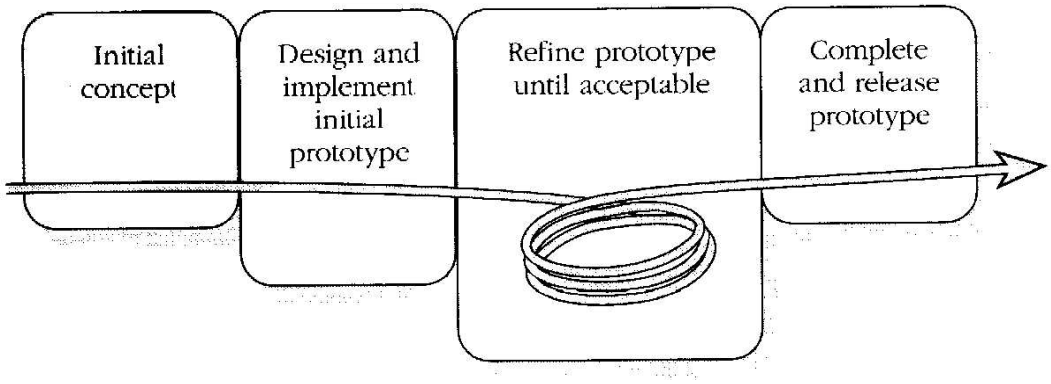
While promoting staged delivery, [McConnell] also warns of increased overhead and forecasts the costs of that result from releasing a product multiple times.

To work well, staged delivery requires careful planning at both management and technical levels. At management level, one has to make sure that the planned stages are meaningful to the customer and keep the size of each stage manageable. At the technical level, dependencies among different components at different stages have to be considered.

Compared to the models discussed previously, staged delivery is the first model that can support the changing nature of projects as defined in Chapter 1. The increased overhead and costs induced by the model are acceptable considering the flexibility, risk reduction, visibility and complexity reduction achieved.

Evolutionary prototyping

Evolutionary prototyping allows the team and customer to develop a system concept as they move through the project. In the first iteration, the most visible parts of the system are developed and presented to the customer. The next iteration is defined by the customer's feedback and results in a refined prototype. This process is repeated until the customer decides that the system is good enough.



[McConnell] recommends evolutionary prototyping when:

- Requirements are rapidly changing.
- The customer is reluctant to commit to a set of requirements.
- Neither the development team nor the customer understands the application area well.

The rewards of this approach are visibility and flexibility. The drawback is that it is very hard to estimate how many iterations are needed and there is the danger of falling into Code & Fix mode.

[Merkl] cautions that this approach, due to changing requirements, could be frustrating to designers and developers.

Evolutionary prototyping shares many similarities and advantages of staged delivery. The major difference is that the former requires much more participation from the customer than the latter method.

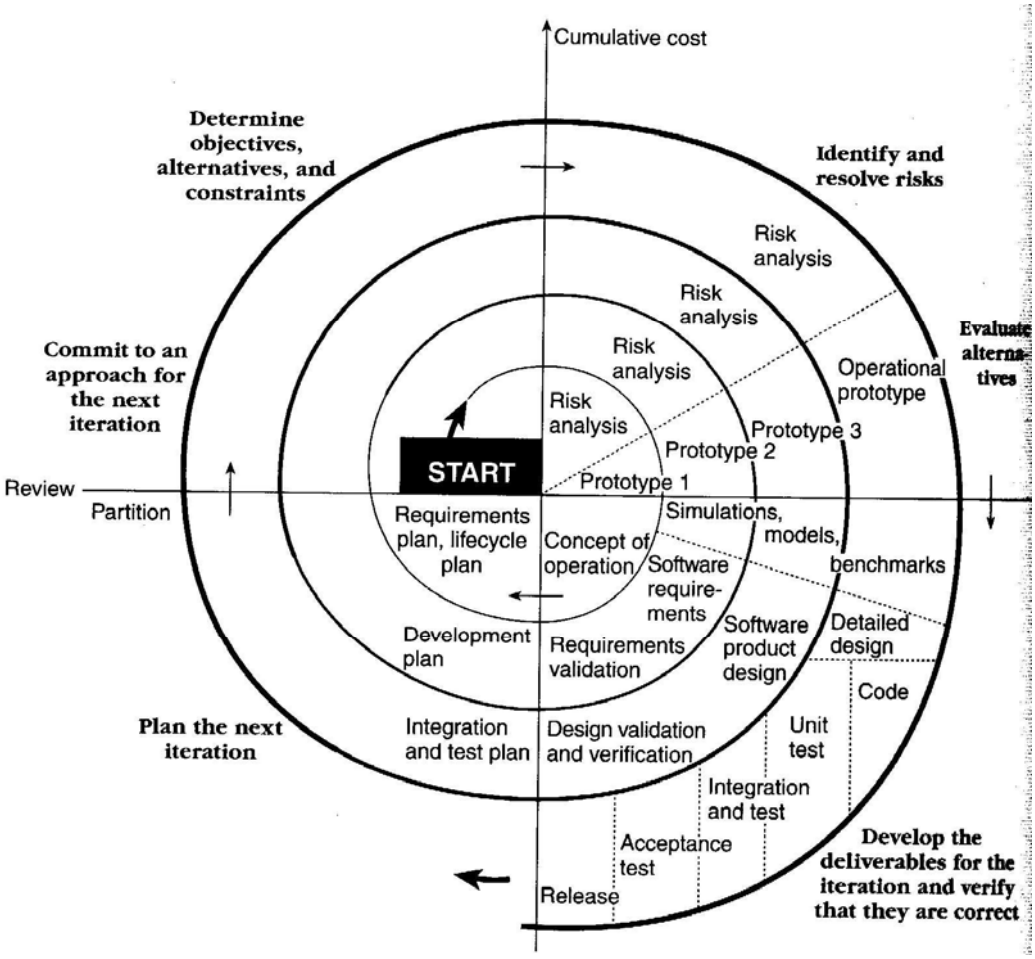
The danger of falling into Code & Fix mode and the resulting frustration felt on the part of designers and developers can happen in any model if the project's changing requirements are not managed. These dangers have to be addressed and monitored in all models.

Spiral

The spiral model is a riskoriented (i.e. risk-oriented) lifecycle model. The project is divided up into smaller projects, each addressing one or more major risks. The spiral model starts on a small scale, explores risks and develops a plan to handle risks. The decision to move on to the next iteration is, as in the waterfall model, considered at the end of each iteration. Each iteration moves the project on to a larger scale.

[McConnell] identifies six steps in the spiral model:

1. Determine objectives, alternatives and constraints.
2. Identify and resolve risks.
3. Evaluate alternatives.
4. Develop the deliverables for that iteration, and verify that they are correct.
5. Plan the next iteration.
6. Commit to an approach for the next iteration (if you decide to have one).



[Merkl] sees the spiral model as a reaction to the weaknesses, visibility and risks, of the waterfall model.

[McConnell] could only identify one disadvantage in the spiral model*: "The spiral model is complicated. It requires conscientious, attentive, and knowledgeable management. It can be difficult to define objective, verifiable milestones that indicate whether you're ready to add the next layer."* To overcome this disadvantage, [McConnell] suggests using the spiral model until all risks have been reduced to an acceptable level and then adopting a less complicated model.

In addition to the points about increased visibility and reduced risks, [McConnell] also mentions the cost effectiveness of the approach: ”*The early iterations are the cheapest. As costs increase, risks decrease.*”

Even though the spiral model is more efficient in addressing risks than the staged delivery and evolutionary prototyping ones, its complexity makes it unattractive as a method. If executed conscientiously, staged delivery and evolutionary prototyping can effectively deal with risks without the complexity of the spiral.

The Snake of Project work

[Gärtner] The underlying model assumes that (almost) everything is interconnected. The project starts with a blurry picture and the focus is constantly changing. As the project progresses, connections are better understood; based on this better understanding, project goals can be defined and solved. The effects of this approach are that goals can be refined or completely re-defined.

Project work does not commence in separated phases. The difference between phases is in the focus, it is always a mix between problem analysis, requirements analysis and implementation (planning).

The idea behind this model is to avoid trying to fit one’s work to an unrealistic model; it tries to find models that reflect the realities (surprises and constant changes) of modern project work.

The main danger of this approach is its complexity. The model tries to reflect the real world thus demanding very experienced personnel to manage the project.

Extreme Programming

[Beck] *“XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software.”*

XP assumes that requirements will change, so it makes no attempt to create and follow an initial design. Instead, the application is designed and developed incrementally in a series of brief design-coding-testing iterations.

Each iteration begins with a team of ten or fewer developers and at least one user representative (the "customer") determining what features the upcoming one- to two-week iteration will implement. During each iteration’s design phase, the user provides a list of "stories" they would like implemented, the developer

estimates the time required to implement each story, and based on that, the customer then decides what to implement in the current iteration.

During the implementation phase, developers often work in pairs. Each pair writes unit tests before they code the information. Unit tests are written with one developer typing and one developer watching for design flaws, algorithmic errors, and general coding problems. The tests are later used to verify the unit's correctness.

Code is integrated, after being tested thoroughly, on a daily basis. At the end of each iteration, the customer has a working (though not full-length) version of the product to use. The customer provides feedback on the current iteration as the team begins designing the next one. The new iteration might implement outstanding features requested in previous iterations, incorporate features that satisfy new business requirements, or refine existing features. After this iteration is planned, the process cycles through another round of implementation, testing, and feedback; additional iterations are performed until the customer is satisfied with the product.

Studies have shown that code quality improves when XP is implemented. [Kolawa] proposes that this improvement occurs because XP promotes error prevention in the following ways:

- By requiring developers to create unit test cases for each piece of code even before it is written, XP ensures that testing occurs throughout the development process, instead of merely at the end.

- By requiring developers to build a large number of test cases and rerun the entire suite religiously, XP encourages test automation. Test automation reduces the risk of human error leading to false positives or overlooked errors. If one can quickly and easily replay the test suite, one will be able to pinpoint and remove errors quickly, before other team members begin building upon the problematic code.

- By using pair programming, XP prompts developers to engage in perpetual code reviews. By requiring flexible and shareable code, it encourages developers to follow coding standards that steer them away from confusing and dangerous coding constructs. When code heeds/ adheres to proven coding guidelines, modifications are much less likely to introduce errors.

[Kolawa] also sees two disadvantages in using XP:

- To work well, XP requires excellent programmers, otherwise the code will lack a real internal structure. *"If the process is not controlled, the developers might use XP as a cover for happy hacking and the program can end up looking like Frankenstein's monster."*

- *"XP can lead to extra work if it is applied in inappropriate situations."* If the project's general scope is known upfront, choosing a more traditional process, one which contains an initial design phase, would give more accurate estimations of resources required and would require less code rework.

XP is one of a new breed of software development models that claim to emphasize development speed and still be able to deliver high quality software. These new models are in reaction to the "software crisis". If there really is a crises and if the answer is XP or maybe just better planning and monitoring will be looked at in Chapter 3.

Summary

The models presented above can be divided into three different categories:

Linear models

These types of models have strictly separated phases. The project is always in one phase only at any given time; at the end of each phase a decision is made as to whether or not to continue to the next phase. Backing up to correct mistakes or implement changes is extremely difficult.

Subproject models

The second type of model breaks down project complexity by defining subprojects that are independent of each other. Subprojects can be completed at their own pace by different teams. Each subproject delivers a functioning system. Backing up to fix problems is easier in this model than in linear models.

Iterations

The final type uses iterations as the main mechanism to refine the system, implement changes and reduce risks.

Choosing the right model

The models discussed above are not bibles to different religions. Different projects have different needs and very often a single project might require

features from different models. The models can be combined or modified to suit the current projects needs. For example, one can reduce the complexity by defining subprojects and in each subproject use a different model.

Choosing a model is one of the most important decisions in a project. However, understanding a model does not necessarily guarantee one's ability to use it. Sometimes it is best to specialize in one model that best suits one's needs.

When deciding which model to use, literature recommends:

Whenever possible, break down the project into independent subprojects. This reduces complexity and allows you to use different types of models in different subprojects.

For straightforward simple systems where the developers and customers are clear about requirements, use a linear model.

For systems with high risk, use a model that includes iterations

When the customer is unclear about requirements, use prototyping to have visible results early and refine them

When constrained by budget and/or schedule, adopt an incremental delivery model and start with highest priority features.

Chapter 3 - Requirements

Why another software project management tool?

The following chapter will review project management literature with the intention of putting together a list of the main requirements facing project management.

A 1994 survey by the Standish Group found that about two-thirds of all projects substantially overrun their estimates.

Six years after the Standish Group study, Kent Beck wrote in his book *Extreme Programming*, *"Software development fails to deliver, and fails to deliver value. This failure has huge economic and human impact. We need to find a new way to develop software."*

Over the years, a great deal of work has focused on the technical aspects of software development (design, testing, validation, and so on.) This led to great technological advances (see Chapter 1- Evolution of an engineering discipline) such as structured programming, structured design, formal verification, and so on. The managerial aspects of software development have attracted much less interest. A possible explanation for this could be ([McConnell]): "*Perhaps this is so because computer scientists believe that management per se is not their business, and the management professionals assume that it is the computer scientists responsibility."*

Project management and software engineering literature are full of examples like the two above. Software development is a complex process, and the reasons for so many failures are many and varied. A project management tool for estimating scheduling and monitoring software projects will not necessarily guarantee success, but this is an area for expansion and research, and where a lot can be achieved in the long run.

The goal of this chapter is to understand the problems and requirements of modern software development. In Chapter 1 we defined the terms "project" and "project management". We will start by grouping the requirements according to the project management tasks defined in Chapter 1. Thereafter, we will consider other important project management issues.

Project management fundamentals

"*Management fundamentals have at least as large an influence on development schedules as technical fundamentals do… organisations that attempt to put software engineering discipline in place before putting project management*

discipline in place are doomed to fail." [McConnell]. Management often controls all three corners of the classic triangle of trade of "schedule-cost-product"; marketing sometimes controls the product and development sometimes controls the schedule.

In XP, there are four variables to control "cost-time-quality-scope". Unlike McConnell, control of the four variables does not lie in the hands of one group. Customers and managers are allowed to pick and control three of the four variables and the development team gets to control the fourth.

Not giving control of all variables to one group makes them more visible. The second advantage is, since none of the groups have all the information required to decide on all variables, the result is better. And finally, people are more likely to commit to a project if their voices and viewpoints are heard.

The four variables XP tries to control are related to each other and cannot be controlled individually. We will be focusing on supporting the variables *time* and *scope*.

The time variable will be supported by calculating the completion date of the project or parts of the project. During execution, forecasting will play an important role. Depending on how fast or slow the project is progressing, the tool will try to forecast completion dates.

Beck sees scope as the most powerful variable. "*If you actively manage scope, you can provide managers and customers with control over cost, quality and time.*" In software development, requirements are never clear at first. Seeing the softness of requirements as an opportunity, and not as a problem, allows the team to shape the system any way the customer needs.

The main requirement which the tool has to support will be to:

Provide the project team (customer, management, developers and all concerned) with information to effectively control the variables *time* and *scope*.

Project management tasks

Identifying and setting goals

Since a project's success or failure is determined by the goals achieved, it is very important to have the concerns and requirements of all involved parties flow into the goal identification and setting process. This also has to be done when goals are changed or new goals are defined.

Most project management models start off by identifying and setting goals by asking customers "What do you need?" Setting goals without thoroughly analyzing the problem is a problem. "*The problem is a problem, is a problem*" [Gärtner]. Problems are manmade (i.e. "man-made") and change when viewed from different perspectives or at different times. Projects are initiated to solve problems or improve circumstances. So defining a good problem lays a foundation for success. But a good problem definition is not enough by itself. Problems change over time so definitions have to be adapted accordingly. However, getting all concerned parties to think about the problem and consider different perspectives makes project requirements less susceptible to change.

The process of problem and goal definition is accomplished by holding interviews, meetings, brainstorming, looking at different technologies, and so on. These tasks cannot be automated or supported by tools and therefore pose no requirements on the tool.

Planning

Planning involves defining tasks, estimating size and assigning resources. The result of planning is a schedule that states who does what by when. Since we defined project goals as not being fixed, so are project plans. Plans can and will, with the approval of all involved, be changed as needed. It is the project team's responsibility to ensure that the project runs according to the current plan.

Software estimation is very difficult, but an accurate estimate builds a strong foundation to work with. You cannot tell how big a software product is or how much it is going to cost until you know exactly what it is.

A 1991 study by Michiel van Genuchten showed that the majority of causes of schedule slips and cost overruns are either related to project planning and

monitoring, or can be anticipated by management. Some surprises which cause project plans to become out-of-date can include:

- Tasks completed late.
- Activities not started on time.
- Test systems in use by another project.
- A technology with less functionality than promised.

In his book *The Mythical Man-Month*, [Brooks] lists five reasons for project delays/failure; all five are related to poor planning and monitoring:

1. Techniques of estimating are poorly developed. They reflect an unvoiced assumption which is quite untrue, i.e., that all will go well.
2. Estimating techniques fallaciously confuse effort with progress, hiding the assumption that men and month are interchangeable.
3. Because we are uncertain of our estimates, software managers often lack the stubbornness to disagree with customers with regard to defined completion dates.
4. Schedule progress is poorly monitored.
5. When schedule slippage is recognized, the natural response is to add manpower.

[Kulik]'s early warning system for software projects suggests that task granularity be one week or less; a schedule slip in a one month long task may not become visible until the end of the month. However, if the task is one week long, a slip will certainly be visible at the week's end.

Software products are usually estimated by looking at the features and objects (Tables, forms, libraries, and so on) and summing up individual estimates. This can be done by applying a top-down, bottom-up or object-oriented approach. Independent of whichever approach is chosen, the tool has to supply an estimation structure that breaks down complexity.

Besides these traditional approaches, there are more and more agile approaches (such as XP) emerging. Agile models require more flexibility; a project is developed as a series of very small releases. The first release contains minimum functionality and each further release builds on that.

So in order to estimate well, we will need an estimation model that can be extended easily.

After estimation has been completed, the project is scheduled. [Kulik] chooses GANTT schedules because they provide a view in the context of time. But Gantt schedules have many problems:

- Gantt schedules are linear but, as Chapter 2 showed, software projects and effective lifecycle models are not linear.
- Gantt schedules are not easily changed; software projects change constantly.
- Gantt schedules warn if the project is late, but do not specify how late. Knowing that the project is late by itself is useless.
- Gantt schedules allow overbooking resources to reach a certain, predefined date. Schedules should be dictated by the amount of work estimated, dependencies between objects and resource availability, and not by a predefined date.

Estimation and Scheduling Requirements:

- **Estimation model that allows breaking down the complexity.**
- **Estimation model that can be extended easily.**
- **Dynamic scheduling based on estimation, dependencies and resource availability.**

Monitoring & Forecasting

Monitoring: Assessing the project status and identifying risks.

Forecasting: In addition to assessing the project status, project management must also forecast completion dates and cost expectations. The data has to be forecast on all project levels (goal, sub-goal level, individual tasks and individual resources /team members) and for different sets of goals.

"Companies that have active measurement programs tend to dominate their industry" [McConnell].

Having an accurate estimate and a realistic schedule puts a project on the right track. To keep the project on track, the team has to make many small adjustments; the process is "*like driving a car*" [Beck].

Monitoring which tasks have been completed, which are open, which have been completed as scheduled, and which early or late tells the team where the

project is standing. Based on these results, the team and /or system can forecast completion dates. Together, they can monitor and forecast the supply information needed to make the many small adjustments.

Another part of monitoring is collecting historical data (metrics). Metrics can help in identifying problems early on and provide a basis for estimating further projects. For [Kulik], collecting metrics is crucial for a project's success and he includes it as part of an early warning system.

McConnell lists measurement as a "Best Practice" that has the following effects:

- Potential reduction from nominal schedule: Very Good
- Improvement in project visibility: Good
- Effects on schedule risk: Decreased risk
- Chance of first-time success: Good
- Chance of long-term success: Excellent

Measurement has dangers as well; measurements can be misused to evaluate employees. Measurements can also be misleading; for example, developers might adopt a coding style that generates large amounts of code to improve the "lines of code" metric, thus focusing on quantity and not quality of code.

Granularity of measurement is also an important factor. Knowing the time spent on the whole project is not as useful as the time spent in each phase (requirements, design, and so on) or on each feature. The time between measurements is also important. Taking measurements on a daily or weekly basis can identify trends within a quarter. Such trends cannot be identified if the measurements are taken monthly or once every quarter.

Two kinds of metrics can be distinguished those that are collected manually and those that can be collected automatically. Automatically collected metrics are harder to manipulate and easier to collect.

Monitoring and forecasting requirements:

- **Monitor estimation and actual progress values on all levels of the project**
- **Forecast the remaining amount of work and completion dates on all levels of the project.**
- **Automatically collect all possible estimation and scheduling metrics.**

Controlling

Depending on monitoring and forecasting results, re-planning is triggered, measures are developed, contingency plans are implemented, and so on.

Controlling a project means changing a project; for example, eliminating problem team members, adding more developers, cutting, adding or redesigning features, and so on. Change management always involves repeating the estimation and scheduling process for all or parts of the system. Furthermore, in many cases, the team has to present different scenarios to choose from.

One of the universal assumptions of software engineering is that the cost of changing a program rises exponentially over time. This assumption is increasingly becoming questioned: "*Under certain circumstances, the exponential rise in the cost of changing software over time can be flattened*." [Beck]

Agile methodologies are becoming more and more popular. From November 2002 to January 2003, Shine Technologies ran a web-based survey to investigate the market interest in Agile methodologies. Here are some of the results of this survey:

- Extreme Programming (XP) is the most popular approach, being used by 59% of participants in the survey. Scrum and Feature Driven taking the second and third position.
- Adoption of an Agile method increases productivity for 88% of participants.
- Quality is improved for 84% of the participants.
- The more positive features are the ability to respond to change and the importance given to people over processes.
- The negative features are the low documentation and the lack of planning and project structure.
- 66% of the participants think that Agile processes are applicable to more than 75% of software development projects.

Agile methods emphasize the need to change. "*Lack of continuous care allows a good design to degrade over time, whereas a dedication to continuous care will correct a poor initial design.*" [Martin] Designs degrade because requirements change in ways that the initial design did not anticipate. Those

changes introduce new and unplanned dependencies between the modules of a system. As the dependency structure degrades, the system becomes ever more rigid, fragile, and immobile. Updating the design is more important than developing an initial good design. The goal is not to create a comprehensive design that cannot degrade, but rather, to create a small but capable initial design, and then to maintain and evolve that design over the life of the system.

By placing the emphasis on continuous care rather than initial design, agile teams are able to keep the designs quality at a very high level, while also being able to migrate the system's functionality to respond to business needs.

The importance of change management is also stressed in the life cycle model "The Snake of Project work" by Gärtner. A project is viewed as something that is constantly changing, and to be successful, project management has to be flexible enough to adjust to those changes as often as needed.

An agile project is developed as a series of very small releases, typically over less than a month. The functionality of the next release is negotiated at the end of each release. Immense attention is paid to migrating the design of the whole system so that it is conducive to the new features.

While allowing maximum flexibility, agile approaches can be very dangerous and require an experienced team. An inexperienced project team could be tempted to start with a poor initial design and hope it will evolve as the project progresses. Every change, no matter how minor, requires more work. Allowing too much change can drag a project on indefinitely. This danger is elaborated upon in "The snake of project work". An important task facing project management is to create better problems by constantly considering all perspectives. This leads to project work that is not separated into distinct phases; it is more a shift of focus. Project work is a constant mix of:

- Analyzing the current situation.
- Analyzing the planned situation.
- Planning and implementation.

Insisting on creating better problems gives the project two advantages:

1. All concerned parties are required to think about the problem and view all perspectives. This leads to a more stable problem that is not so susceptible to change.

2. The change process requires analyzing the current situation as well as the demanded changes. This re-evaluation helps identify new or changed problems.

Controlling requirement:
- **Project structure has to allow changes.**
- **Implementing changes has to be quick and easy.**
- **The consequences of changes have to be visible.**

Other important issues

Software Lifecycle Models

In Chapter 2 we saw that lifecycle models distinguish three structures:

- Linear structures like the waterfall model.
- Hierarchical models that break down the complexity (staged delivery).
- Iterative models like the spiral model.

Extreme programming adds a fourth demand on lifecycle models. The first iteration of XP produces a minimal functioning system which is then extended in further iterations. Parallel to that, bugs found in releases of previous stages are collected and fixed. So the system is developed and maintained simultaneously.

Agile programming is becoming more and more important, but simple lifecycles, such as the waterfall model, still have their uses (e.g. in maintenance projects). Moreover, as mentioned in Chapter 2, there is no universal lifecycle model that can be used for all software projects. So the tool will not be limited to one type of development.

Lifecycle requirements:
- **Support linear, hierarchical and iterative lifecycle.**
- **Support multiple projects.**

Support projects using a mix of different lifecycle structures.

People

Even though projects are very technical and rely on various technologies, tools and processes, they are performed by people and an unmotivated or unqualified team member can do a lot of damage to the project. Therefore, peopleware issues should get as much attention as any other issue in project management.

For [Merkl], the main danger facing a project comes from the people involved. The reasons he cites for this include: incapable employees or project managers, too much or too little control by management, conflicts within a team or between departments, customers disagreeing with goals and stealing of resources between rival projects.

[McConnell] concludes, "*Peopleware issues have more impact on software productivity and software quality than any other factor.*" He backs this statement with studies that show a 10 to 1 difference in productivity between different developers and 5 to 1 between different teams. The reasons being: skill levels, motivations, good working environment, and so on.

The importance of people is also discussed by [Trauring], [Boehm], [Kulik], [Brown] and others; here are some points to consider:

- Strong leadership and teamwork: Strong leadership can overcome – or even substitute for – many organizational and process weaknesses. Today's software environments are so complex that lack of teamwork is an obstacle to the successful completion of software projects.
- Management sponsorship: Helps ensure that projects receive the resources they need. Management sponsorship comes with a cost, usually more hands-on involvement in the project by management.
- Highly productive programmers: Productivity of individual programmers can vary by a factor of 10. Highly productive individuals are set on the project's critical path.
- Top talent: Use better and fewer people.
- Job matching: Fit the tasks to the motivations, not only to the skills of the people available.
- Career progression: Help people advance in their career rather than forcing them to work where they have the most experience or where they are needed most.
- Team balance: Select people who will complement and harmonize with each other.
- Misfit elimination: Eliminate and replace "problem" team members as quickly as possible.

- Have programmers and customers talk: Communicate, communicate, communicate, and communicate.

We have now seen many issues that show the importance of people, but none of them can be controlled, resolved or automated by project management tools. However, one important issue is still missing; that of overtime. One of the rules of Extreme programming is, "*Work no more than 40 hours a week. Never work overtime a second week in a row*".

None, or little, overtime keeps the team members fit and motivated. They can enjoy their free time and concentrate better when at work. Too much overtime can undermine motivation and burnout team members, resulting in: lower quality product, high defect rates, more schedule slips, people leaving the project (according to McConnell, the best developers leave first) and even project cancellation.

Overtime has become the norm in software development today; there are even project management seminars that simulate real life projects by holding workshops late into the night. Overtime is sometimes necessary but it should not become routine and should not be planned at the beginning of a project. The majority of project management tools allow for the overbooking team members.

People requirements:

- **Resources cannot be overbooked; Overtime cannot be planned.**

Risk

"*The basic problem of software development is risk.*" [Beck] In his book *Extreme Programming Explained*, Kent Beck sets out on a mission to find a solution to risk. Not paying attention to risk results in budget and schedule overruns; in the worst case, the project is cancelled. Risk cannot be attributed to a certain stage or activity during a project; they can be found in all areas of a project.

[McConnell] compares projects without risk management to gambling: *"they are more like the purchase of a lottery ticket than a calculated business decision."*

Risk is the final issue we will be looking at, not because it is less important than the others, but because it is part of all issues we have discussed so far.

Actually, we have been discussing and resolving risk all along. Each requirement stated addresses one or more risks facing software development and management.

An example of risk:

Feature creep: Adding new features to the system without evaluating the impact and changing the schedule and/or design can have the following consequences: the system design degrades, new bugs are introduced, bugs are detected faster than they are fixed, schedule slips have to be taken, team moral deflates, team members start leaving the project and in the worst case the project is cancelled. The consequences do not have to be as dramatic; this example is merely to illustrate the snowball effect that could occur if no risk management is practiced.

The job of software risk management is to identify, address, and eliminate sources of risk before they become threats to successful completion of a software project. [McConnell] identifies 5 different levels of risk management:

1. Crises management: Address risks only after they have become problems.
2. Fix on failure: Detect and react to risks quickly, but only after they have occurred.
3. Risk mitigation: Plan ahead in time to provide resources to cover risks if they occur, but do nothing to eliminate them in the first place.
4. Prevention: Implement and execute a plan as part of the software project to identify risks and prevent them from becoming problems.
5. Elimination of root causes: Identify and eliminate factors that make it possible for risks to exist at all.

Efficient risk management addresses risks at levels 4 and 5. Levels 1 through 3 handle risks only after they have occurred, which is too late.

Software Risk Management is an iterative process comprising of three steps:

> **Identify:** Risk identification can be accomplished through brain storming, formal risk assessment or using risk assessment tools.
> **Act:** Acting means implementing strategies that significantly reduce risks or taking steps to avoid them altogether.
> **Monitor & Evaluate:** Since projects change as they progress, risks identification and the effectiveness of the implemented strategies have to

be re-evaluated in regular intervals. A prioritized risk list can be very helpful.

Risk identification can be supported by risk assessment tools, but a risk assessment tool is not within scope of this thesis. Of the three steps involved in risk management, monitoring is most suited for automation.

As the tool will be concentrating on estimation and scheduling, so will the risk management issues. The following is a list of risks that can be reduced if the tool is used properly:

- Feature creep: Dynamic scheduling will automatically update the completion dates as soon as features are added to the project plan.
- Overly optimistic schedule: By automatically generating schedules based on estimation, dependencies and resource availability, a schedule cannot be made to converge early.
- Acceptance and team moral: Not allowing planned overtime increases the acceptance of the schedule and motivates the team members.
- Underestimating a project: Breaking down the complexity leads to estimation of smaller features and tasks. The estimation errors are also smaller, so the danger of under- or overestimating a project is reduced.
- Visibility: Monitoring actual progress values at all levels increases visibility.
- Schedule slips: Monitoring also provides an early warning system; monitoring on all project levels allows pinpointing problem areas before they become real problems.

One more requirement

Besides the requirements identified above, we will be trying to fulfill one more, secondary, requirement:

Keep the tool as simple as possible

This requirement is important as long as none of the other requirements are impaired. The reasons for simplicity being:

- Easy models are easy to understand.
- Easy tools are easy to learn and use.
- Project teams are challenged enough and don’t need a further challenge.
- A tool should aid, and not hinder, a project.

Summary

The following is a summary of all the requirements discussed in this chapter.

Project management fundamentals:

- Provide the project team (customer, management, developers and all concerned) with information to effectively control the variables *time* and *scope*.

Estimation and Scheduling Requirements:

- Estimation model that allows breaking down the complexity.
- Estimation model that can be extended easily.
- Dynamic scheduling based on the estimation, dependencies and resource availability.

Monitoring and forecasting requirements:

- Monitor estimation and actual progress values on all levels of the project.
- Forecast the remaining amount of work and completion dates on all levels of the project.
- Automatically collect all possible estimation and scheduling metrics.

Controlling requirement:

- Project structure has to allow changes.
- Implementing changes has to be quick and easy.
- The consequences of changes have to be visible.

Lifecycle requirements:

- Support linear, hierarchical and iterative lifecycles.
- Support multiple projects.
- Support projects using a mix of different lifecycle structures.

People requirements:

- Resources cannot be overbooked; overtime cannot be planned.

Secondary requirements:

- Keep the tool as simple as possible.

The next chapter will concentrate on designing the tool.

Chapter 4 - Designing the tool

Now that we have worked out the tools requirements, it is time to work on the design. The focus will be on the following three areas:

- Estimation
- Scheduling
- Monitoring and forecasting

The challenge facing us in this chapter is to find a design that is simple and useful. Making the design too simple will result in a useless tool; on the other hand, having a very complex design can result in a tool that is too hard to use and ends up being rejected by users.

Estimation

"*Most projects overshoot their estimated schedules by anywhere from 25 to 100 percent. Without an accurate estimate there is no foundation for effective planning and no support for rapid development*." [McConnell] But as we have seen in previous chapters, it is hard to estimate accurately. McConnell found that at the beginning of a project, high and low estimates can differ by a factor of 16. Even after requirements have been completed, estimates can be off the mark by about 50 percent.

McConnell presents a three step process to estimating projects:

1. Estimate the size of the product: This is the hardest step.
2. Estimate the effort (man-months): This is done by adding up the estimates of step 1. If estimates were not a duration (e.g. lines of code), they are converted to a time format before summing up.
3. Estimate the schedule (calendar months):

$$\text{schedule in months} = 3 * \text{man-month}^{1/3}$$

Estimation indicators

There are several established ways to estimate a product size. The following are most commonly used indicators:

Lines of code: The project team estimates the product size by estimating how many lines of code are required. However, different tools, coding styles and different levels of experience all play a role in how much code is produced / generated. Then there are also activities (e.g. analyzing data from a database) that do not generate any code at all. Some projects, even though they are

software projects, do not require much coding; most of the development work is supported by tools.

Number of function points [Vijay]: Function points is a scientific method that:

- Measures functionality that the user requests and receives.
- Measures software development, maintenance rates and size independently of the technology used for implementation.
- Provides a normalizing measure across projects.

The basis of function points uses “Data functions” and “Transactional Functions” for estimating effort. Data functions refer to data storage and transactional functions refer to processing complexity.

Function points produce accurate estimates, much better than lines of code. But it does not fit the requirements defined in Chapter 3:

- Break down complexity.
- Can be extended easily.
- Dynamic scheduling based on estimation, dependencies and resource availability.

Duration: Instead of estimating size and then estimating the required development time, this approach goes straight to time (task) estimations. The approach may not be as scientific as Function Points, but the model will use durations for the following reasons:

- Easy to understand and implement.
- Allows estimation of all other tasks that are not function points and do not produce code, but are still part of the project.
- The estimation can be used for scheduling without being converted.
- Better suited to modern software development where the emphasis is more on modeling and designing than on “hard-core” programming.
- Better suited to our definition of “project" which requires that the team defines “goals”, so the next logical step would be to define the tasks required to achieve the goals.

Estimation methods

McConnell advises using range estimates because they provide a more accurate picture of reality. A similar estimation to ranges is PERT estimation. An optimistic, realistic and pessimistic estimate is made for every task and the weighed average of all three is the tasks PERT estimate.

Another factor that influences estimation accuracy is task granularity. Setting the granularity to one week or less results in more tasks to estimate, but the individual estimation errors are also smaller, so errors are more likely to cancel out than they are in larger tasks.

A side effect of decreased task granularity is an increasing number of tasks that have to be managed. This task list can become very long very quickly, making it very hard to manage. To avoid such huge lists, we can use a hierarchal estimation model similar to Work Breakdown Structures found in literature (e.g.[Merkl]). Complex (large) goals are simplified by defining sub-goals and the same can be done to simplify sub-goals. The actual estimation is done at task level and aggregated up the hierarchy to get sub-goal and goal estimates.

In Chapter 1 we identified goals as one of the basic characteristics of a project, so the model will use hierarchal estimation to support that characteristic. Other reasons for this decision are:

- Hierarchies reduce complexity.
- Hierarchies increase flexibility:
    - Allow bottom-up and top-down planning.
    - Allow modeling subsystems independent of each other.
    - Allow modeling goals, sub-goals.
    - Allow modeling by stages (requirements, design, etc.)
    - Allow modeling by functionality (core functionality, top priority functionality, etc.)
    - Allow modeling by releases (release 1, 2, etc.)

Refining the estimate

Our requirements discussion in Chapter 3 illustrated that software development is a process of gradual refinement. Newer lifecycle models discussed in Chapter 2 (e.g. “The snake of project work”) stress the need to continually refine project plans. McConnell’s three step process to estimate projects is also intended to be performed several times during the life of a project. While the results produced by this approach are very good, it is only useful when the project is small or re-estimation is only required in certain project areas; otherwise it is a very time intensive process.

An alternative to having team members re-estimate the project is having the project re-estimate itself. This can be achieved by using the following formula:

$$NewEstimate = \left(\frac{100}{PercentOfTaskCompleted} \right) * NumberOfDaysWorked$$

The formula requires that team members enter and update two values for every task that is being worked on:

1. *PercentOfTaskCompleted*: Percentage of total task workload completed.
2. *NumberOfDaysWorked*: The number of days spent so far.

To get a project re-estimate, the individual task re-estimations are summed to get sub-goal re-estimates, sub-goals are again added up to get goal re-estimates and finally a project re-estimate.

Compared to manual re-estimation, automatic re-estimation offers the following advantages:

- Requires less time: Team members have to only update tasks they are currently working on and not the whole project which could consist of several hundred tasks.

- Less room for manipulation: Projects are very often (see Chapter 3) underestimated. When allowed to re-estimate the whole project, team members can manipulate the estimates to fit certain needs. However, allowing team members to only update *PercentOfTaskCompleted* and *NumberOfDaysWorked* for tasks currently being worked on reduces the effect of manipulation.

To re-estimate sub-goals and goals, tasks that will be started in the future have to be considered as well. The current approach uses the original estimates for those tasks; however, consider the following:

> A sub-goal consists of ten tasks. Each task requires developing a report, and the complexity of all reports is the same. The original estimate was 1 day per report. At the end of day 1, the developers update the information for task 1 and enter the following: 50% completed, 1 day needed. The current re-estimation model would re-estimate task 1 as 2 days and leave the other 9 tasks unchanged; the resulting sub-goal re-estimation is 11 days. But since all reports have similar complexity, a more realistic estimate would be 20 days.

To achieve this accuracy, the model will be extended as follows:

For every task completed or in progress, a slip/gain factor is calculated using the following formula:

$$SlipGain = \left(\frac{NewEstimate}{OriginalEstimate} \right)$$

The slip/gain factors of all tasks (completed or in progress) are averaged to get a new factor which will be used to re-estimate tasks that are planned to start in the future. For more accuracy, the factor can be weighed differently in different sections of the hierarchy, allowing local tasks to have a larger influence on their immediate goals / sub-goals.

Scheduling

"*Excessive schedule pressure is the most common of all serious software engineering problems*." [McConnell] To underline his statement, McConnell quotes research results from the 60's to the 90's:

1967, Bylinsky: "All significant programming problems turn out to be emergencies."

1975, Brooks: "More software projects have gone awry for lack of calendar time than all other causes combined."

1984, Costello: "Deadline pressure is the single greatest enemy of software engineering."

1991, 1994, Jones: "Excessive or irrational schedules are probably the single most destructive influence in all of software."

But why is scheduling such a problem? A schedule should state WHO does WHAT by WHEN while considering DEPENDENCIES between tasks. Still, many projects are scheduled unrealistically. Some of the causes are [McConnell]:

- External, immovable deadlines such as the date of a computer trade show.
- Managers or customers refuse to accept a range of estimates and make plans based on a single-point "best case" estimate.
- Managers and developers deliberately underestimate the project because they want a challenge or like working under pressure.
- The project is deliberately underestimated by management or sales in order to submit a winning bid.

- Developers underestimate an interesting project in order to get funding to work on it.
- The project manager believes that developers will work harder if the schedule is ambitious and therefore creates the schedule accordingly.
- Top management, marketing, or an external customer want a particular deadline, and the project manager cannot talk them out of it.
- The project begins with a realistic schedule, but new features are piled on to the project, and before long the project is running under an overly optimistic schedule.
- The project is simply estimated poorly.

Types of schedules

Before we discuss scheduling in detail, let us take a look at different types of schedules and what issues one needs to consider when deciding which schedule to commit to.

[McConnell] identifies three types of schedules that, under certain conditions, can be achieved by the project team.

Shortest possible schedule:

The shortest possible schedule uses many schedule-reducing practices, allows no room for mistakes and requires a very experienced software development organization. Shortest possible schedules contain many optimistic assumptions, such as:

- Team members come from the top ten percent of the talent pool. They have all had several years of experience working with the programming language and environment. They all have detailed knowledge of the application area, are motivated, share the same vision and the expected employee turnover, during the project, is zero.
- Ideal project management.
- Advanced software tools are available, developers have unlimited access to computer resources. Communication technologies such as network, video conferencing, phones, email, and so on, are integrated into the work environment.
- The most time efficient development methods and tools are used. Requirements are completely known at the time design work starts and do not change.

Efficient schedule:

Efficient schedules assume that one does most things right, but stop short of making the ideal-case assumptions that underlie the shortest possible schedule.

- Team members come from the top 25% of the talent pool. Everyone has worked with the programming language and environment for a few years. Turnover is less than six percent per year.
- Efficient use of programming tools. Modern programming practices. Rapid development practices.
- Active risk management.
- Integrated use of communication tools.

Nominal schedule:

Nominal schedules are intended for use on average projects and use less optimistic assumptions than shortest possible or efficient schedules.

- Team members come from the top 50% of the talent pool and have some familiarity with the programming language and environment. The team has some, but not extensive, experience in the application area and expected turnover is between 10 to12 percent per year.
- Programming tools and efficient programming practices are used to some extent, but not as much as would be in efficient schedules.
- Risks are managed less actively than is ideal.
- Communication tools are available, but not integrated in the workflow.

Given that the shortest possible schedules are out of reach for nearly all organizations [McConnell], Efficient or Nominal schedules are a more realistic alternative. Most projects will find that the efficient schedule is also the best case schedule; these schedules are achievable if nearly everything goes right.

In Chapter three we identified the need to be flexible during a project as a requirement. This is best fulfilled by nominal schedules. They may not be as fast as efficient schedules, but they are not the worst, and they allow for minor delays and changes to take place.

Schedules shorter than the shortest possible schedule are considered to be too optimistic or too aggressive. Project teams that adopt such schedules can find themselves getting caught in a vicious circle of: Schedule slips → more schedule pressure → more stress → more mistakes → more schedule slips →

In the worst case, the project is cancelled. Other effects of overly optimistic schedules are quality loss, higher employee turnover and reduced loyalty.

In most cases, (see causes of unrealistic scheduling) the responsible people know that they are scheduling aggressively. But can a tool stop aggressive scheduling? The final decision is not taken by a tool, so the answer is definitely NO. However, a tool can make visible or help detect aggressive schedules.

Scheduling methods

The basis for scheduling are goals – tasks and estimates of individual tasks. Scheduling then assigns team members to tasks, defines dependencies between tasks and finally puts the whole project within a timeframe.

We will look at three commonly used scheduling methods- milestones, bar charts (GANTT) and network plans- and then present a fourth method, priority scheduling.

Milestones:

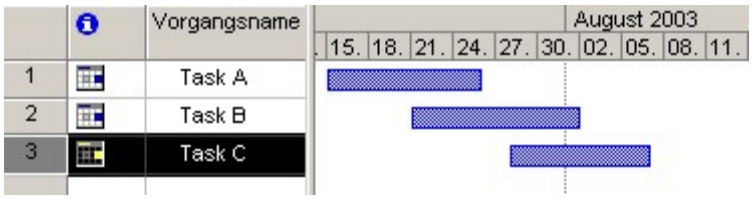
Milestones mark significant or critical events in the life of a project. The events define an end of a phase, a goal or a critical activity. One way of creating a schedule would be to define the milestones and schedule tasks to meet these milestones.

Another approach to milestone scheduling is setting the milestones depending on when the final task, belonging to a milestone, is completed.

Both methods can be used to create all kinds of schedules. However, the second approach sets milestones according to estimated workload, so the project team will probably by less tempted to compress tasks to meet certain milestones. Real life projects can use both approaches simultaneously, or adopt other approaches not discussed here.

Gantt Charts:
A Gantt chart is a horizontal bar chart which includes the following features:

- Activities identified on the left hand side.
- A time scale drawn on the top (or bottom) of the chart.
- A horizontal rectangle drawn against each activity indicating estimated duration.



Gantt charts are very popular and are supported by most scheduling tools (e.g. Microsoft Project). There are no widely accepted standards, different tools incorporate different features such as:

- Dependencies between activities.
- Display of original and latest time for task.
- Display of person(s) allocated to tasks.
- Integration with other planning techniques (i.e. networks and milestones).
- Display of critical path.

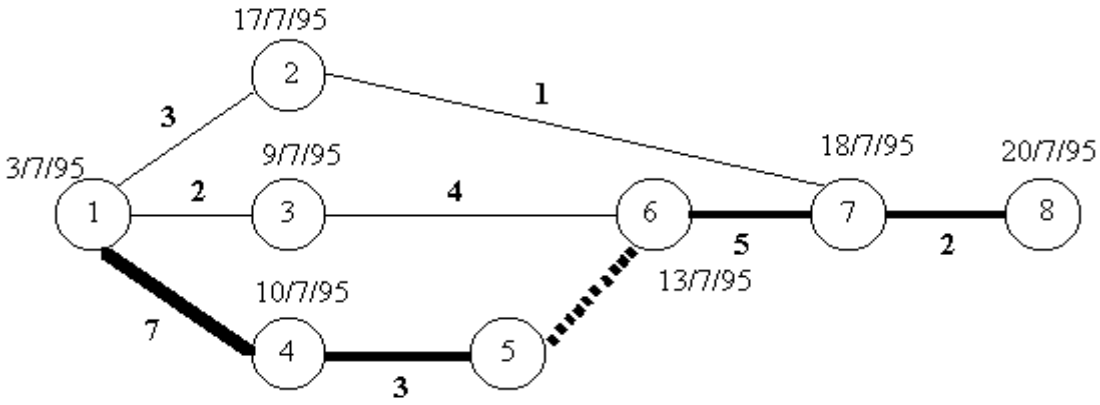
Gantt charts are graphical, easy to read and small changes can be done quickly. However, big changes that involve rescheduling large parts of a project can be difficult and time consuming. Other disadvantages of Gantt charts (in its original form):

- It does not show precedence relationships between activities.
- Although the status of individual activities can be ascertained, the overall project status is hard to determine.

Activity Networks:

The foundation of this approach came from the Special Projects Office of the US Navy in 1958. It developed a technique for evaluating performance in large development projects, which became known as PERT - Project Evaluation and Review Technique.

The heart of any PERT chart is a network of tasks needed to complete a project, showing the order in which to complete and the dependencies between them. This is represented graphically:



The diagram consists of a number of circles, representing events within the development lifecycle, such as start or completion of a task, and lines which represent the tasks themselves. Each task is additionally labeled by its time duration. Thus the task between events 4 & 5 is planned to take 3 time units. The advantages of PERT are:

- Identification of the critical path.
- Definition of dependencies between tasks (not possible in the original GANTT).

Priority scheduling

All three methods, milestones, bar charts and networks are very popular. The models can be combined (e.g. GANTT with milestones or precedence) or extended (e.g. hierarchal GANTT to allow more detail in the lower levels) to form different models.

However, none of the models consider priorities. Priority scheduling places higher priority goals (or tasks) ahead of those that have a lower priority. Objects that have the same priority are scheduled together.

Priority scheduling offers advantages that are difficult to implement in the models discussed above:

- Many people tend to prioritise their work when planning. In such cases, using priority scheduling is more coherent with the planning process already being used and does not require any transformations; e.g. to dependencies

- Setting dependencies by itself is not always enough. Different goals can depend on each other, e.g. “Build Database Tables” and “Develop Input Forms”, or they can be independent but have different priorities; e.g.

"Building Interface to Finance System" could be more important than "Showing a preview before printing".

- Flexibility: Priorities offer great flexibility in change management. For example, the scheduling of two or more goals, each consisting of several sub-goals and many tasks, can be scheduled differently by only changing priorities at top level. Setting the priority equal would then schedule the goals together and resolve the priority issues at lower levels in the hierarchy. The flexibility is also apparent when adding goals or tasks to the project. By choosing the right priority, a goal or task can be scheduled at the beginning, end or between two other goals/tasks without having to manipulate existing objects.

Even though priorities are very effective in scheduling, they do not fulfil all of one's needs. One big drawback is that one cannot define dependencies. A workaround would be to assign higher priorities to tasks/goals that depend on others, however, a person who is not familiar with the project will not be able to interpret the dependencies. So, if we want to have a complete, useful model, we should not neglect dependencies.

Scheduling Model

In the previous section we defined an estimation model. This model will now be extended to incorporate scheduling. To be effective, the model will support priority and dependence scheduling.

Team members

Knowing the priorities and dependencies between tasks and goals is not enough to create schedules. The actual work on a project is completed by the project team. Working hours, project availability (holidays, working on other projects) and tasks assigned all have to be considered when scheduling. So we will introduce team members to our model and define the following properties:

- Work hours: Monday to Sunday.
- Intervals (begin and end dates) assigned to project.
- Percentage of work time assigned to project.

Team members can only be assigned to tasks. The assignment to sub-goals and goals is inherited from tasks. Since more than one person can be working on a single task, the percentage of workload assigned to a team member also has to be defined.

Scheduling using priorities

So far, we know WHO does WHAT, but we still do not have an order in which to complete the goals / tasks. This is done using either priorities (0..highest, 99..lowest) or dependencies. The project will be stored in a hierarchal structure. The details of the project are hidden in the lower levels of the hierarchy.

The projects schedule will be generated top down. The algorithm will start at the root node and work its way down the hierarchy using the following rules:

- When deciding which task/goal to schedule first, the objects are compared at the same level.
- Schedules that cannot be resolved (same priority) are delegated to lower levels of the hierarchy.
- Dependency relationships are stronger than priorities and, if present, are used to create the schedule.

To better understand the model, we will look at an example:

The following table defines our sample project (X). It consists of a requirements and design goal and two goals to develop two features. The goals 1.2 & 1.3 both depend on goal 1.1.

	Priority	Predecessor	Days Planned
1 Project X	10		16
1.1 Requirements & Design	10		4
1.1.1Requirements analysis	10		1
Customer			0.5
Project Manager			0.5
1.1.2 Design	20	1.1.1	3
Project Manager			1.5
Developer1			1.5
1.2 Feature A	30	1.1	4

1.2.1 Detailed design	10		1
Developer 1			1
1.2.2 Code & Test	20	1.2.1	2
Developer 1			2
1.2.3 Integration & System Test	30	1.2.2	1
Developer 3			1
1.3 Feature B	30	1.1	8
1.3.1 Detailed design	15		2
Developer 1			2
1.3.2 Code & Test	20	1.3.1	4
Developer 1			2
Developer 2			2
1.3.3 Integration & System Test	30	1.3.2	2
Developer 3			2

The model starts at the root and works its way down the hierarchy.

Step 1:

Since we only have one goal at the highest level, we can go one level deeper into the hierarchy. Here we find 3 sub-goals (1.1, 1.2 and 1.3). Both sub-goals, 1.2 and 1.3, depend on sub-goal 1.1, leaving only one starting point, sub-goal 1.1.

After scheduling sub-goal 1.1 we have to decide which sub-goal follows. Since no dependency is defined between the sub-goals, we resolve the schedule using the priorities. But both goals have priority 30, so the schedule has to be resolved at lower levels in the hierarchy; only then can we say whether or not the sub-goals are equally important, and if work is to start simultaneously or if the priorities differ at lower levels.

So after Step 1 we have the following schedule:

1	Requirements and Design
2	Feature A Feature B

Step 2:

Sub-goal “Requirements and Design” has two tasks, “Requirements Analysis” (1.1.1) and “Design” (1.1.2). Task 1.1.2 depends on task 1.1.1, so our schedule now looks like this:

1	Requirements and Design \ Requirements Analysis
2	Requirements and Design \ Design
3	Feature A Feature B

Sub-Goals “Feature A” and “Feature B” both have the same priority and do not depend on each other. In such a case, we resolve the schedule by comparing priorities at a lower level. Within each sub-goal, the schedule is defined by dependencies (1.2.1 → 1.2.2 → 1.2.3) and (1.3.1 → 1.3.2 → 1.3.3). To schedule the sub-goals together, we start by comparing the priorities of the starting points; Task 1.2.1 has priority 10 and 1.3.1 has priority 15, so 1.2.1 is scheduled first. Next, we compare the priorities of tasks 1.3.1 and 1.2.2 and schedule 1.3.1. Next on our list are tasks 1.2.2 and 1.3.2; both have priority 20, so they are scheduled together and, as with sub-goals 1.2 and 1.3, refined at a lower level. The same applies to tasks 1.2.3 and 1.3.3.

1	Requirements and Design \ Requirements Analysis
2	Requirements and Design \ Design
3	Feature A \ Detailed Design
4	Feature B \ Detailed Design
5	Feature A \ Code & Test

	Feature B \ Code & Test
6	Feature A \ Integration & System Test Feature A \ Integration & System Test

Step 3:

Our sample project has three levels (excluding root), so the algorithm will also go through three steps and schedule the project as follows:

1	Requirements and Design \ Requirements Analysis \ Customer Requirements and Design \ Requirements Analysis \ Project Manager
2	Requirements and Design \ Design \ Project Manager Requirements and Design \ Design \ Developer 1
3	Feature A \ Detailed Design \ Developer 1
4	Feature B \ Detailed Design \ Developer 1
5	Feature A \ Code & Test \ Developer 1 Feature B \ Code & Test \ Developer 1 Feature B \ Code & Test \ Developer 2
6	Feature A \ Integration & System Test \ Developer 3 Feature A \ Integration & System Test \ Developer 3

Calculating completion dates

Now that we have an execution order, we can use the “days planned” and team member properties to calculate the completion dates of individual tasks, sub-goals and the project itself. During the project we can use refined estimates (see “Estimation”) to dynamically update the schedule.

If we go back to our example, we see some tasks have 2 team members assigned to them. We know how many days are planned, we know the working hours of the team members and what percentage of their work time is reserved for the project. However, is that enough? Consider the following situation:

2 team members are assigned to a task for which 4 days have been planned. The working hours of the team members are as follows:

- TM1: Only Monday
- TM2: Monday to Friday

If the work is divided up equally, then the task will take 2 weeks to complete (2 days for TM2 and 2 weeks for TM1). How long will the task take to complete if the work was not divided up equally? The following table shows how those values could look:

% of workload assigned		Number of work days		Time needed to complete	
TM1	TM2	TM1	TM2	TM1	TM2
10%	90%	0.4 Days	3.6 Days	0.4 Days	3.6 Days
50%	50%	2 Days	2 Days	2 Weeks	2 Days
90%	10%	3.6 Days	0.4 Days	4 Weeks	0.4 Days

The time needed by team member 2 equals the number of days planned; but team member 2 only works 1 day per week; So, depending on how much of the total workload is assigned, the time needed to complete the project varies between 0.4 days and 4 weeks. That is a huge difference and should not be neglected when scheduling. So we will introduce another property, **% of Workload**, to our model. The property will be defined when assigning team members to tasks and is only relevant when more than 1 person is assigned to a task.

Monitoring and forecasting

"Companies that have active measurement programs tend to dominate their industries." [McConnell] Measurement has both short term motivational benefits and long term cost, quality and schedule benefits. Some of the advantages (as found in literature) of measurement are:

- Improved planning of future projects: Analysis of historical data can make strengths and weaknesses of individuals, teams or organizations visible. The results can then be used as a basis for future planning.
- Provides status visibility: Measurement during a project greatly improves project visibility. Questions such as “How far are we?”, “Which tasks/Goals are on / behind schedule and by how much?” can be answered and used to steer the projects.
- Focuses people’s activities: People will change (improve) the way they work to optimize the characteristics being measured.
- Helps set realistic expectations: The project team will be in a much stronger position to make and defend schedule estimates if they are supported by accurate measurement.
- Lays the groundwork for long-term process improvement: Good measurement, performed consistently over several years, can help identify weaknesses and strengths in the company’s strategies, practices, technologies, and so on. *“Measurement is the cornerstone of any long-term process-improvement program.”* [McConnell]

What to measure

Collecting metrics is both time-consuming and costly. So it is very important to only measure metrics that are useful. (Basili and Weiss) defined a Goals, Questions, Metrics process to help identify relevant metrics:

- Set goals: The first step is to determine areas that need to be improved.
- Ask questions: Formulate questions that have to be asked in order to meet your goals.
- Establish metrics: Set up metrics that will answer the questions.

The focus of our tool is estimation, scheduling and forecasting; the same applies to metrics. We will follow the Basili and Weiss approach to determine which metrics can be collected and what they can be used for.

In Chapter 3 we identified the following requirement: “Automatically collect all possible estimation and scheduling metrics”. We have to keep this in mind while defining metrics.

Set Goals

As stated above, the tool’s focus is estimation, scheduling and forecasting. In previous sections, we designed an estimation and scheduling model that would hopefully provide the required improvement. The model also incorporates

automatic re-estimation and completion date forecasting. However, it provides little support in creating the initial estimate; that is left to the project team. An experienced team relies on its experience to create accurate estimates; an inexperienced team will do its best to get to accurate estimates and in some cases, guess instead of estimate. So the first goal is:

Goal 1: Provide support for the initial project estimate.

Another challenge facing estimation is getting approval. A good estimate by itself is not always enough; the estimate has to be approved, very often by people who want high quality, low cost and short schedules and, in some cases, by people who do not have an insight into how difficult software estimation and scheduling can be. These people have to be convinced that the project team's estimate is accurate and realistic.

Goal 2: Provide support in getting approval for the estimate.

During the project, the model re-estimates and forecasts probable completion dates; but that does not relieve the team from monitoring and controlling activities.

So a third goal is:

Goal 3: Provide support for monitoring and controlling.

Ask Questions

Goal 1: Provide support for the initial project estimate: An experienced project team can rely on its experience from previous projects to get to an accurate estimate. The inexperienced team will need support; the amount required increases as the level of experience decreases. This support is realized by supplying the team with data from past projects to compare with. To arrive at an accurate estimate, the team would try to answer questions such as "How long did it take to complete a similar object (object being a project, goal, task, feature, stage, document, and etc.?"

Having the estimates and actual values by themselves is not enough. The team still needs to know which objects can be compared to each other. So additional information is required:

- How complex are the objects?
- What is the size (e.g. lines of code)?

- What tools were/will be used? And how much support is provided by the tools?
- How much experience do the team members have in the application area, tools, programming language, duties assigned…?

Goal 2: Provide support in getting approval for the estimate: The answers to the questions listed under Goal 1 provide strong support when trying to get approval. This support can be increased if the team can also show what the consequences of underestimating a project can be. The following questions, concerning projects that were not completed as planned, can help in achieving goal 2:

- How much delay did the project experience?
- How often was the project rescheduled?
- What overhead resulted from poor estimation?
- Was the project ever completed? Were parts completed?
- What was the impact on the project team? What was the staff turnover? Did members leave the team / company after the project completion or cancellation?

Goal 3: Provide support for monitoring and controlling: The support required by monitoring and controlling can be achieved by supplying answers to the following questions:

- How far are we?
- Are we ahead or behind schedule?
- How far ahead or behind schedule are we?
- When will we be done?
- How well are the different goals / sub-goals progressing?
- Are there any problem goals / sub-goals?
- How well are the individual team members working?
- Are there any “problem members”?

Establish metrics

The requirement was “automatically collect”, so metrics such as lines of code, function points, open/closed defects, even though they answer the questions above, will not be part of the tool.
The first 4 questions of goal 3 can be answered by the model without being collected explicitly. The answers are current project values that are always visible; however, we will collect the data in order to be able to compare the values at different points in time and detect trends in the projects.

An important issue in metrics is granularity. Data collected at too large a granularity (e.g. at top goal level) is not very useful. The right granularity depends on the project and on what is being measured / analyzed. Our model is hierarchal so we will collect data at all levels (goal, sub-goal, task and team member assignment) of the project and we will be collecting aggregated data for the individual team members.
Another issue to consider is the time span between two measurements. Daily? Weekly? Yearly? Again, the requirements vary from project to project. To be flexible, the tool will allow data collection to take place at any point in time.
The following metrics will be collected:

Days Planned: The number of days planned (estimated) by the project team.
% Complete: Percentage of work completed as entered by the project team.

Days Worked: The number of days worked as entered by the project team.

New Estimate: The new estimate calculated by the tool.

Days Ahead/Behind Plan: Using the value estimated by the project team as the planned value and subtracting from that the automatically refined **NewEstimate** delivers the difference to plan. A negative value indicates that work is behind plan, a positive value means ahead of plan and 0 indicates work is progressing according as planned.

DiffPlan = Plan – NewEstimate

Completion Date: As with the estimates, the completion date is also constantly changing.

Efficiency: Comparing the **DiffPlan** of a goal estimated at 10 days to a goal estimated at 100 days is not meaningful. To allow for such comparisons, we will calculate an efficiency factor. As with **DiffPlan,** a negative value indicates that work is behind plan, a positive value means ahead of plan and 0 indicates work is progressing according to plan. But the values of different goals, sub-goals, tasks or team members can be compared with each other to see where work is progressing more efficiently.

Efficiency = (Days Planned - New Estimate) / Days Planned

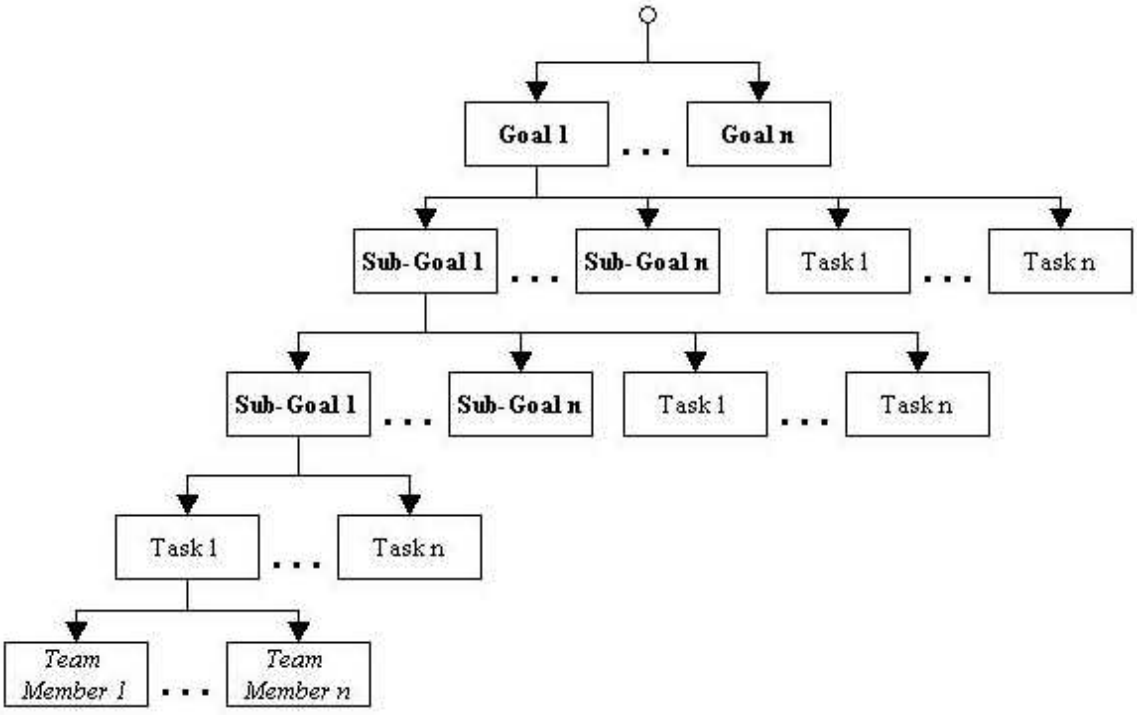
Summary

The hierarchal estimation and scheduling model described above consists of three building blocks:

Goals / Sub-goals: Used to group sub-goals and tasks. Every Goal/Sub-goal can have 0..n sub-goals and 0..n Tasks. Relationships between different goals, sub-goals and tasks are defined using either priorities or dependencies.

Tasks: The individual steps that need to be executed to achieve the goals and sub-goals. Every task must belong to a goal or sub-goal and has 1..n Team members assigned to it. The actual project estimation is done at task level by estimating the number of days needed to complete the task.

Team Members: The individuals who complete the tasks. To be able to calculate the completion dates, the team members' working hours (Monday - Sunday) and assigned workload have to be supplied.



Now that we have designed our model, it is time to look at the requirements once again and see if they have been fulfilled.

Provide the project team (customer, management, developers and all concerned) with information to effectively control the variables time and scope.

If the project data is updated regularly, the model supplies the following information instantly and at all levels (Goal, sub-goal, task and team member):

- Days Planned
- % of work completed
- Number of days worked
- Refined estimate (as soon as work starts)
- Number of days Ahead or behind plan (as soon as work starts)
- Number of work days remaining to complete goal
- Probable Completion date

Estimation and Scheduling Requirements:

- *Estimation model that allows breaking down the complexity.*
  Complexity is reduced by allowing sub-projects.

- *Estimation model that can be extended easily.*
  Using priorities as well as dependencies allows adding goals and tasks anywhere in the project structure easily.

- *Dynamic scheduling based on the estimation, dependencies and resource availability.*
  Scheduling is automatic and, if the estimates are realistic, the schedule produced will also be realistic.

Monitoring and forecasting requirements:

- *Monitor estimation and actual progress values on all levels of the project*
- *Forecast the remaining amount of work and completion dates on all levels of the project.*
- *Automatically collect all possible estimation and scheduling metrics.*

As soon as work starts, estimates are refined, progress is calculated and remaining work and completion dates are forecast.

Controlling requirement:

- *Project structure has to allow changes.*
- *Implementing changes has to be quick and easy.*
- *The consequences of changes have to be visible.*

The hierarchal structure and collected metrics support all three controlling requirements. The model will be tested in the next chapter to see if the provided support is enough.

Lifecycle requirements:

- *Support linear, hierarchical and iterative lifecycles.*
- *Support multiple projects.*
- *Support projects using a mix of different lifecycle structures.*

Different lifecycles and multi-projects are also supported. The next chapter will show how to implement the different lifecycles.

People requirements:

- *Resources cannot be overbooked; overtime cannot be planned.*

The people requirement is supported in two ways. First, a team members working hours cannot exceed 8 hours per day and 40 hours per week. Second, dynamic scheduling will not overbook a team member, even if the person is assigned several tasks that have to be completed simultaneously.

The model designed in this chapter has fulfilled all the requirements we set. But is it also useful? How does it compare to other tools? What are the advantages? What are the disadvantages? To answer these questions, we will develop a prototype to test. The prototype and the tests will be the focus of chapter 5.

Chapter 5- The tool and its components

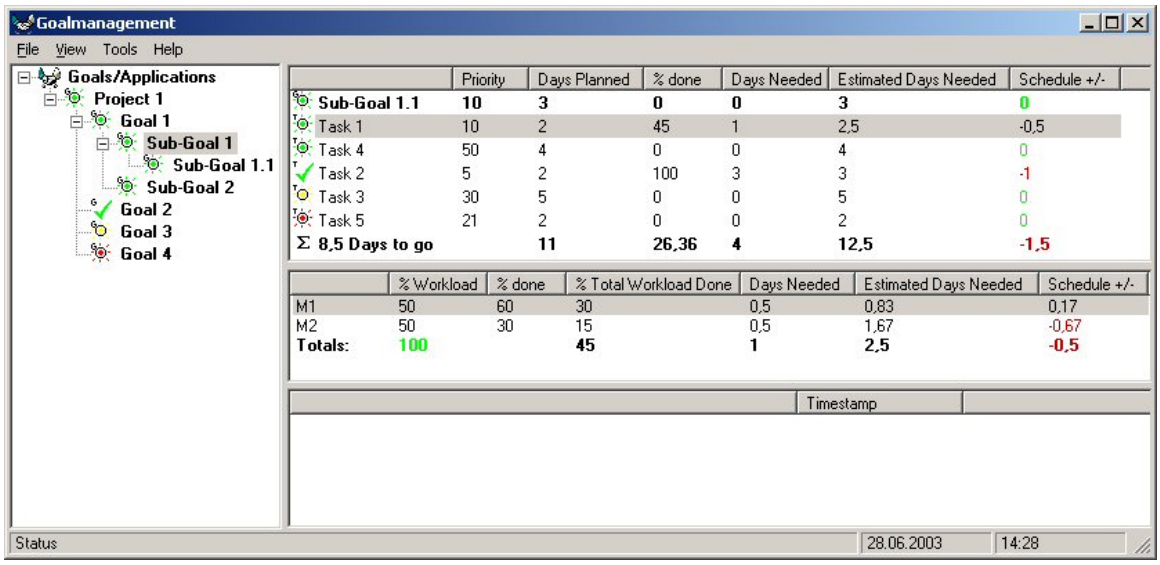
In Chapter 4 we designed a model to support the requirements defined in Chapter 3. To find out how well it implements these requirements, a prototype will be developed and tested.

Due to time constraints, the following restrictions will be imposed on the prototype:

- Only team member working hours (Monday to Sunday) will be stored and used for scheduling. Team member holidays, % of work time assigned to project and official holidays will not be considered.
- One work day is equivalent to eight working hours.
- Schedules will be created using priorities. Dependencies between tasks and goals have to be modeled implicitly using priorities.

Prototype

Overview



Goalmanagement's main window is divided into four panes. The left pane shows the hierarchies and status of different projects.

The top-right pane contains detailed information on the selected project, goal or sub-goal. Goals and Sub-goals are printed in bold and tasks in normal letters. The six columns to the right of the name display the objects:

- priority
- number of days planned

- percentage of workload completed
- number of days worked so far
- new refined estimate
- difference to plan in days

The last row sums up the values of the individual steps and displays the remaining number of workdays left.

The middle and bottom pane on the right are only active when a task is selected in the top-right pane.

The middle pane shows which team members are assigned to a task, how the workload is divided up and progress information similar to that above.

The bottom-right pane can be used to add time-stamped notes to individual tasks.

Icons

The following icons are used to illustrate an object's status:

 Goal or sub-goal status open.

 Goal or sub-goal status completed.

 Goal or sub-goal status on hold.

 Goal or sub-goal status closed.

 Task status open.

 Task status completed.

 Task status on hold.

 Task status closed.

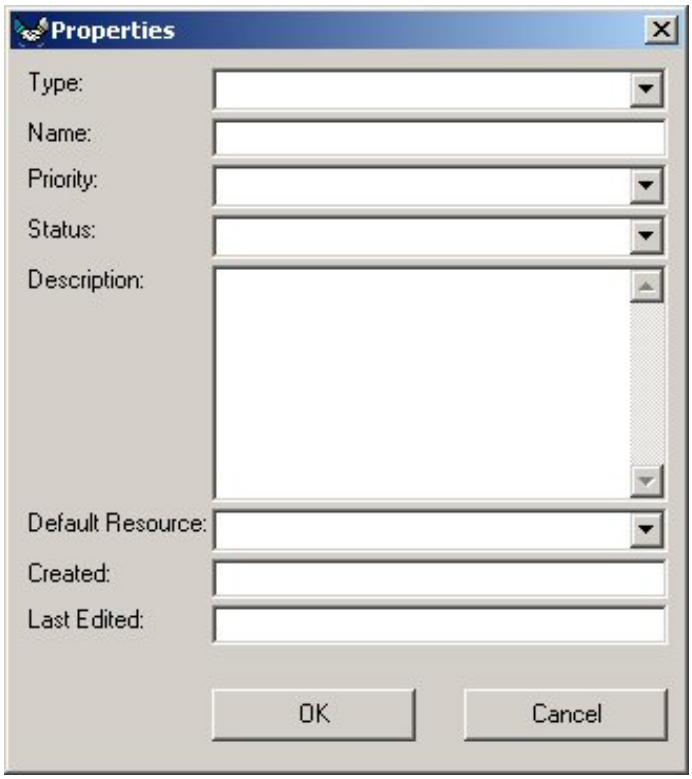
How do I…?

The following "How do I…?" is intended to help users familiarize themselves with Goalmanagement's basic concepts and operations.

… add a goal or sub-goal?

Projects, goals and sub-goals are all treated by Goalmanagement equally. Objects that are directly under the root node are defined as projects. Objects directly under a project are goals and objects under goals are sub-goals. To create a new object, right click the mouse while the parent object is selected

and from the pop-up menu choose **Add Goal**. The following dialog, goal properties, will be opened:



Type: Defines whether the goals result is an application or not.

Name: Goal name.

Priority: Goal priority between 0 (highest priority) and 99 (lowest priority).

Status: A goal can have any of the following status:

- Open: Work on the goal is either in progress or has not yet begun. To have status Open, at least one sub-goal/task has to have status open. Goals with status open are considered when generating schedules and refining estimates.
- Completed: A goal's gains status is completed when all sub-goals and tasks are completed, on hold or closed. Goals with status completed are considered when generating schedules and refining estimates.
- On Hold: Status on hold is used when the project team is unsure whether the goal will be completed or cancelled. Goals with status on hold, as well as all their sub-goals and tasks, are not considered when generating schedules and refining estimates.

- Closed: Status closed is used for goals that are no longer part of the project, but have to remain in the project structure for documentation reasons. As with status on hold, goals with status closed, as well as all their sub-goals and tasks, are not considered when generating schedules and refining estimates.

Description: Detailed description of goal.

Default Resource: The team member who, by default, will be assigned to tasks belonging to the goal.

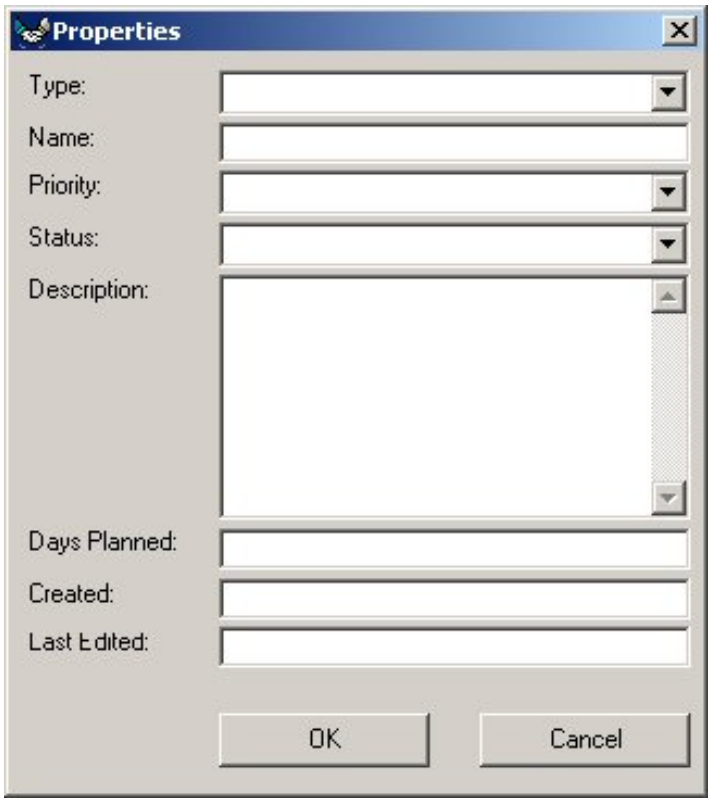
Created: Timestamp indicating the goals creation date. This property is read only.

Last Edited: Timestamp indicating when the goal was last edited. This property is read only.

… add a task?

Tasks cannot be created under the root node, every task must belong to a goal or a sub-goal.

To create a new task, right click the mouse while a goal or sub-goal is selected and from the pop-up menu choose **Add Task**. The following dialog, task properties, will be opened:



Type: Defines whether the tasks result is a software object (table, form, report, library, etc.) or not.

Name: Task name.

Priority: Task priority between 0 (highest priority) and 99 (lowest priority).

Status: A task can have any of the following status:

- Open: Work on the task is either in progress or has not yet begun. Tasks with status open are considered when generating schedules and refining estimates.
- Completed: Indicated that work has been completed. Tasks with status completed are considered when generating schedules and refining estimates.
- On Hold: Status on hold is used when the project team is unsure whether the task will be completed or cancelled. Tasks with status on hold are not considered when generating schedules and refining estimates.
- Closed: Status closed is used for tasks that are no longer part of the project, but have to remain in the project structure for documentation reasons. As with status on hold, tasks with status closed are not considered when generating schedules and refining estimates.

Description: Detailed description of task.

Days Planned: As described in Chapter 4, project planning is done at task level. Days planned is the team's initial estimate; the values at task level are aggregated to get sub-goal and goal estimates. They are also used to generate schedules and determine a project's progress.

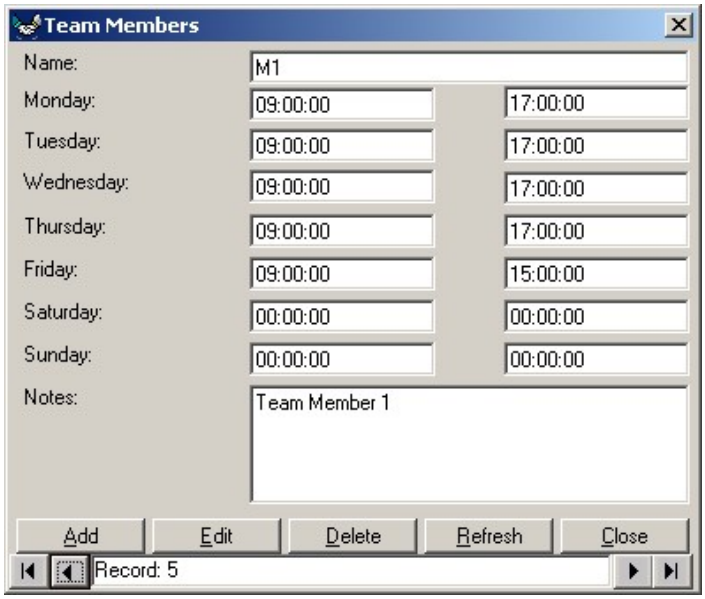
Created: Timestamp indicating the task's creation date. This property is read only.

Last Edited: Timestamp indicating when the task was last edited. This property is read only.

… add a new team member?

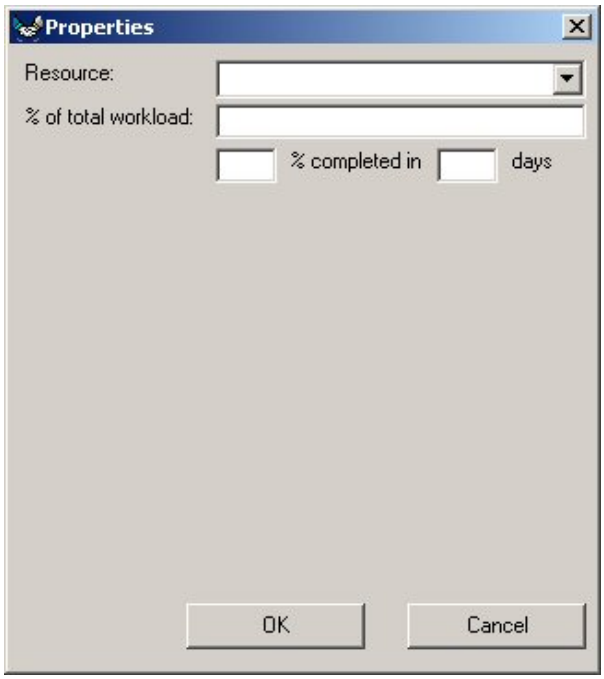
Select **Tools->Team members** to open the team member's properties dialog and click on **Add**. A team members start and end times have to be entered for each day of the week. The difference between start and end time defines the total number of hours a team member will work on a certain day. A work day is defined as 8 working hours, so any value above 8 would mean that the team member will be doing overtime. The standard setting for a team member is 8 hours per day Monday to Friday and none on Saturday and Sunday.

The working hours defined are used to generate task and goal start and completion dates. Changing the working hours of a team member will not automatically update scheduled tasks; the schedule has to be generated again.



… assign team members to tasks?

The final step in setting up a project is assigning the team members to the individual tasks. By default, the team member selected in the goals/sub-goals dialog is automatically assigned to all tasks at lower levels of the hierarchy. But sometimes a task is to be completed by a different team member or by more than one team member. Selecting a task in the top-right pane will display the assigned team members in the middle right pane. By right clicking a team member in the middle right pane and selecting **Properties**, a team member's assignment can be edited. A new team member can be assigned by right clicking in an empty area and selecting **Add Resource**. In either case the following properties dialog will be opened:



Resource: Displays a list of all team members.

% of total workload: Defines what percentage of the task's workload is to be assigned to the team member. The default value is 100%; this property is used when more than one team member is assigned to a task. The sum of all percentages is displayed in the totals line of the main window and has to equal 100%. A value greater or less than 100% is displayed in red.

% completed: Refers to the percentage of the assigned workload completed. This property is to be updated as work progresses.

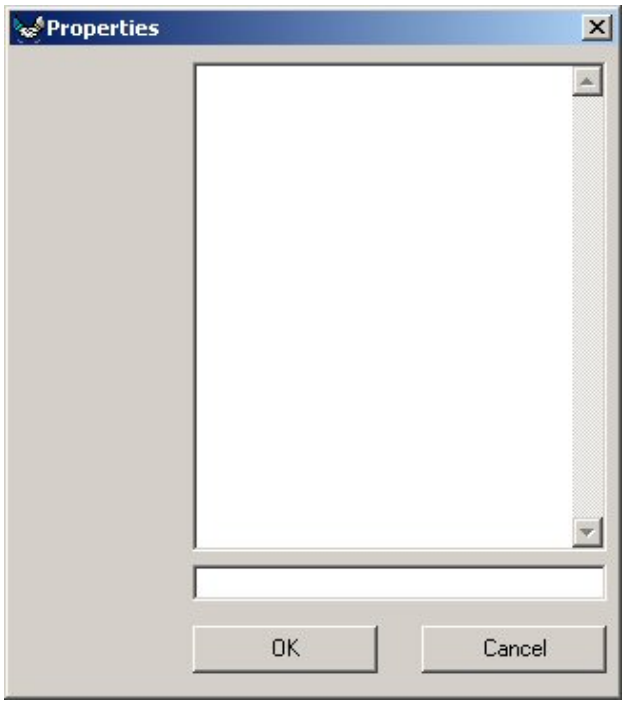
Days: Refers to the total number of days worked so far to complete the percentage in the previous property. As with % completed, this property is updated as work progresses.

The property's **% completed** and **days** play a central role in refining project estimates. They are used to refine team member estimates and compared to initial estimates to determine the team member's progress. The values are aggregated to determine progress at task, sub-goal, goal and project levels.

… add notes to a task?

Every task has a description field that allows defining the task in more detail. However, editing this field results in overwriting old data which is not always desirable. To overcome this shortcoming, team members can attach notes to tasks.

While a task is selected in the upper-right pane, right clicking the lower-right pane and selecting **Add Note** opens the following dialog:



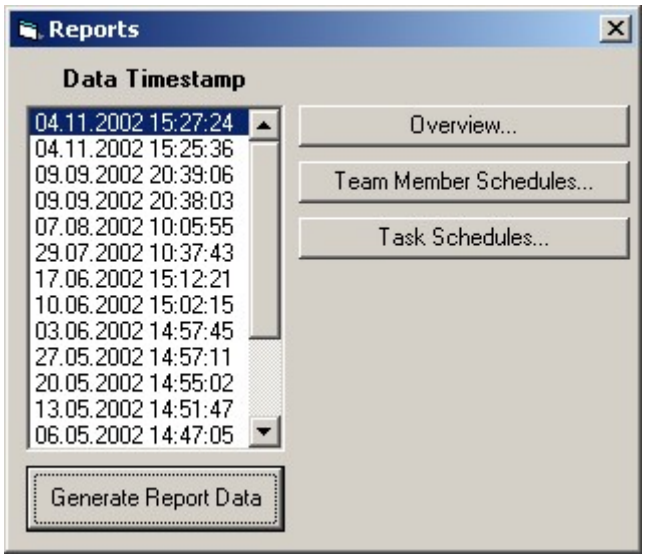
The upper field allows a text of variable length to be entered and the lower field is automatically time-stamped to allow reconstructing the order in which notes were entered.

… generate report data?

A projects current status is always visible in the main window. However, schedules have to be generated explicitly and are marked with a timestamp. In addition to begin and end dates, the system also stores a snapshot of the projects current status. The reasoning behind this is twofold:

1. Having different versions of a schedule allows for a comparing of the project at different phases during its lifecycle.
2. The historical data generated is used as input for project analysis. This is the topic of the next section.

Selecting **tools → reports** from the main will open the following dialog:



By pressing the button **Generate Report Data**, a new set can be created. The data is time-stamped with the current date and time and the timestamp is added to the list. The prototype comes with three predefined reports:

1. Overview: A hierarchal overview of all projects down to team member assignments. The data contained is the same as that in the main window plus probable completion dates for all components.
2. Team Member Schedules: Displays execution order and status information of all tasks assigned to a team member.
3. Task Schedules: Displays execution order and status information of all tasks.

Selecting a timestamp from the list allows viewing reports at different stages during the project.

… analyse data?

Project analysis is realized in a mini data-warehouse-like module. Input data comes from the report data generator described above. Each data set is marked with a timestamp to allow for timeline analysis.

The following metrics are collected when generating report data:

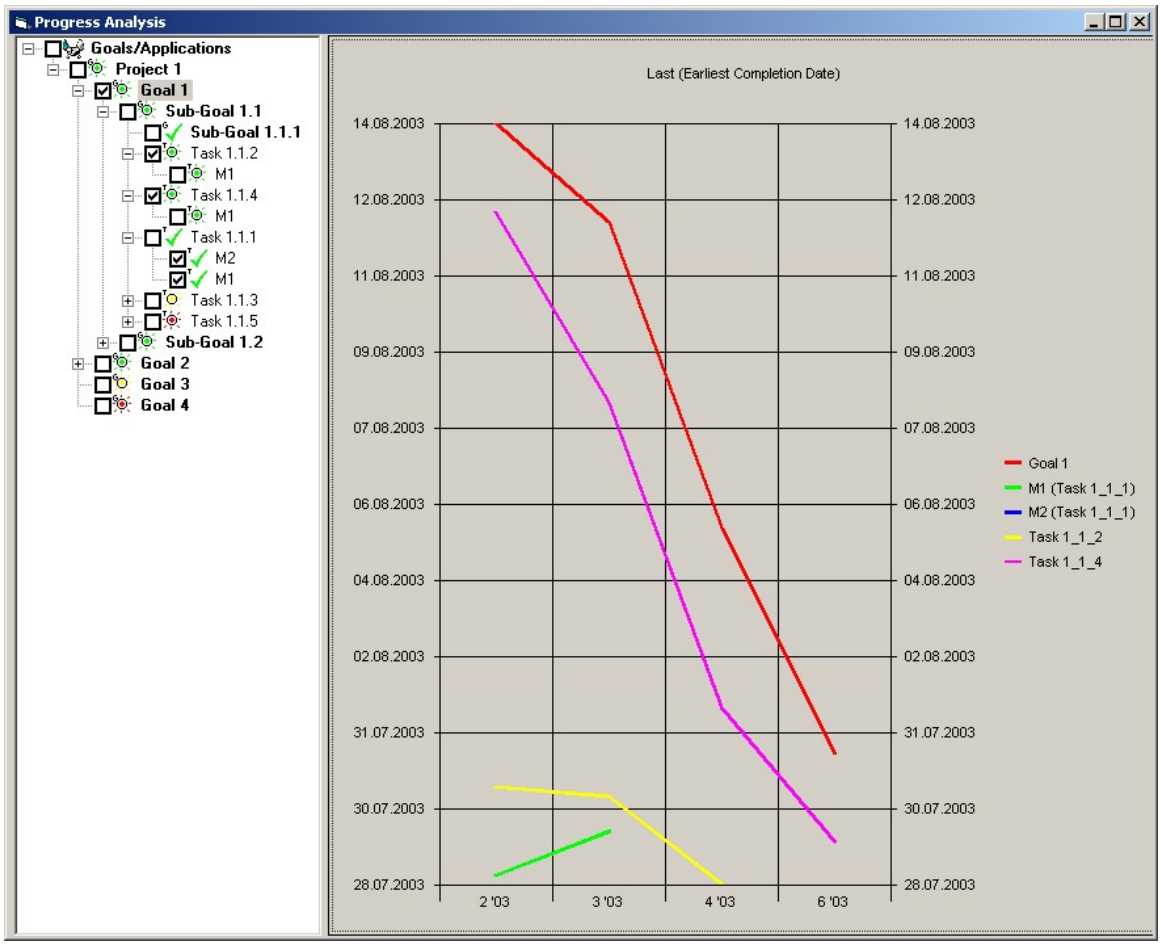
- **Days planned:** Reflects the team's estimate for an object in the project. Objects being Project, Goal, Sub-Goal, Task or Team Member assignment.
- **New Estimate (Days):** Is the new estimate calculated by the system as soon as work starts on an object.

- **% Completed:** The percentage of an object's workload completed as entered by team members.
- **Days worked:** Number of days needed to complete **%Completed**.
- **Days Ahead/Behind Plan:** Calculated by the system based on the project team's initial estimate **Days Planned** and systems **New Estimate**.
- **Completion date:** Calculated by system, uses a combination of the project team's initial estimates and systems calculated new estimates. The team's estimates are used for objects before work starts, and as soon as work progresses on an object, the systems forecast (**New Estimate**) is used.
- **Progress:** Progress is calculated to allow direct comparison between two or more objects. The following formula is used to calculate progress:

  (Days Planned – New Estimate) / Days Planned

  A 0 value indicates that work is progressing as planned or work has still not begun. A positive value indicates work is progressing better than planned, and a negative value indicates work is not progressing as planned.

The data analysis window consists of two panes: tree view on the left and graph on the right. The graph's x-axis represents a time line and can be grouped by year, quarter, month, week or day. The y-axis shows a scale for the selected metric.

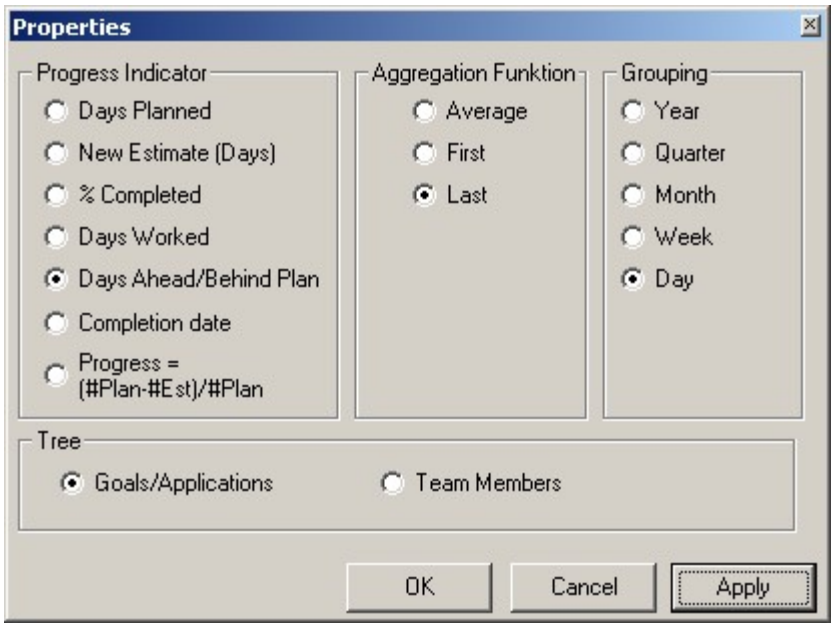


Two analysis modes are distinguished:

1. Goals/Application: Allows analysing individual components of a project. The tree shows the project's hierarchy down to individual team member assignments.
2. Team members: Is used to analyse the work of individual team members. The tree contains a list of all team members and depending on which metric is selected, different aggregations are used to analyse the overall progress for that metric.

The tree views of both modes allow for the selection of multiple objects and each object is represented by its own line in the graph. Only one metric can be viewed at any time.

The analysis module is controlled by the following dialog (right click on the graph and select **properties**):



The **Tree** block allows for a switching between the two modes of Goals/Applications and Team Member. The **Progress Indicator** block is used to select a metric. **Grouping** is used to determine how the time line (x-axis) is presented. Since there is no restriction on how often or in what intervals data can be generated, **Aggregation Function** is used to determine whether the grouped data represents the beginning (First), end (Last) or average of the selected time grouping.

Summary

This chapter provided a brief overview of the prototype and supplied a “How do I …?” guide to ease the process of learning and understanding the software. While designing our model in Chapter 4, we looked at different methods and models used to estimate, schedule and monitor projects. The next chapter will introduce two real life projects that will be managed using the tool and commercial systems. The results will be compared with each other and to the actual values.

Chapter 6 - Tool Evaluation

Chapters four and five discussed a project management approach and developed a tool prototype to aid the approach. The next step is to evaluate the prototype as follows:

- The prototype is used to (re-)manage two real life projects and the results are compared to actual values.
- The same projects will be (re-)managed using commercial systems and the results of all tools will be compared.

Projects

The projects chosen for evaluation both come from the data warehousing domain. However, the larger part of Project one, Management Information System (MIS), was concerned with customising ready systems while the focus of Project 2, Marketing Data Mart (MDM), was more on designing and developing routines to automate data extraction and loading.

Project 1: Management Information System (MIS)

MIS's goal was to implement a data warehouse to allow detailed analysis of production, sales, deliveries and returns of a bakery.

A DOS based order tracking system running on a Novell server and cash registers in the shops served as data providers. The data warehouse was implemented on a Microsoft SQL Server and Proclarity desktop professional was chosen as a front-end analysis tool.

As with many software projects, MIS went through a fair amount of changes, the two main reasons being:

1. Poorly documented systems: The bakery's order tracking system was the main supplier for deliveries and returns data. The systems had been in use for a long time, had gone through various customisations to fulfil the customers needs and the available documentation was incomplete and of low quality.
2. Customer's little experience regarding data warehousing: The customer had, in the beginning stages, difficulty understanding and visualising the possibilities offered by a data warehouse and that affected the ability to define requirements.

The project was divided into 4 sub-goals (stages), specification, implementation, testing and installation and training. The numerous changes to the data warehouse structure and business rules made tackling the goals one after another impossible. Every change required going back to the specification, which, as we have discussed previously, is not unusual for software projects.

Project facts:

	Planned / Estimated	Actual
Start date	03.03.2003	10.03.2003
Completion date	07.08.2003	04.08.2003
Workload in days	50	38.5

Project 2: Marketing Data Mart (MDM)

MDM was a sub-project within a larger marketing project. The marketing department of a telecommunications company bought a marketing tool which promised more effective, more focused marketing campaigns with the possibility to accurately measure and track the response of individual campaigns. To function properly, the tool (Marketing Director) required its own data mart; the scope of MDM was to build this data mart. The data had to be extracted from a data warehouse running on NCR Terradata and loaded into an Oracle system.

The projects sub-goals were:

- Gathering requirements.
- Installing a development and test environment.
- Designing the data mart, data retrievers (from Terradata) and data loaders (to Oracle).
- Implementation.
- Testing.
- Switching to production.

Project facts:

	Planned / Estimated	Actual

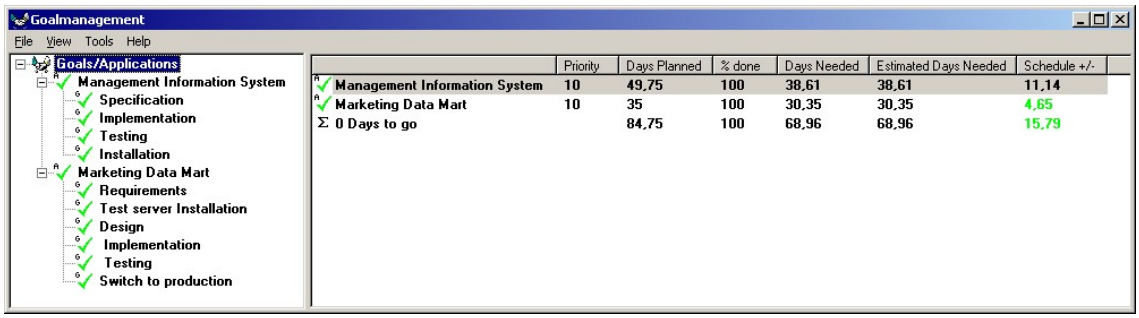
Start date	15.05.2003	12.05.2003
Completion date	14.07.2003	25.08.2003
Workload in days	35	30.5

The changes experienced by MDM, as compared to MIS, were minor, yet MDM still missed the planned completion date by more than a month. However, the project was still a success since it was within budget, and the actual number of days worked was less than the 35 planned days.

Tool Evaluation

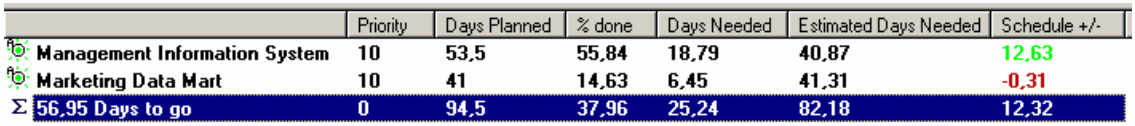
Project status

As we have seen in our discussions throughout, knowing a project's status is vital to success. The tool's main window supplies the project team with such information. The following snapshot was taken after both projects were completed.



At one glance we can see that project MIS was completed 11.14 days ahead of plan and project MDM was 4.65 days ahead of plan; taken together, the total figure is 15.79 days. These values correspond with actual values.

The following snapshot shows the project's status towards the end of May. Back then the teams were almost half a day behind on MDM and 12.6 days ahead on MIS.



The percentage completed on each project and on both can be easily read and we can see that both teams together already worked 25.24 days out of 94.5

planned and had still 57 days of work ahead. The new estimate, based on progress data entered was 82.18 days.

These values we have just read from the snapshots are the answers to questions a project manager has to be able to answer at all times (see Chapter 1):

- How far are we?
- How much work do we still have?
- Are we ahead or behind plan? And by how much?

With a few mouse clicks, a project manager can drill down the project's hierarchy and answer those and similar questions at different detail levels. Or by going to the root level, these questions can be answered for all projects together.

Comparing the two snapshots we see that project plans had been changed from 94.5 to 84.75 days. By drilling down, we will find two tasks whose status have been changed to "Closed", thus taking them out of the planning, but leaving them in for documentation reasons. The same can happen if an object is changed to "On Hold" or if the planned values are increased or decreased. What changes, when and which objects were affected can be read from the metrics collected during the project.

Metrics

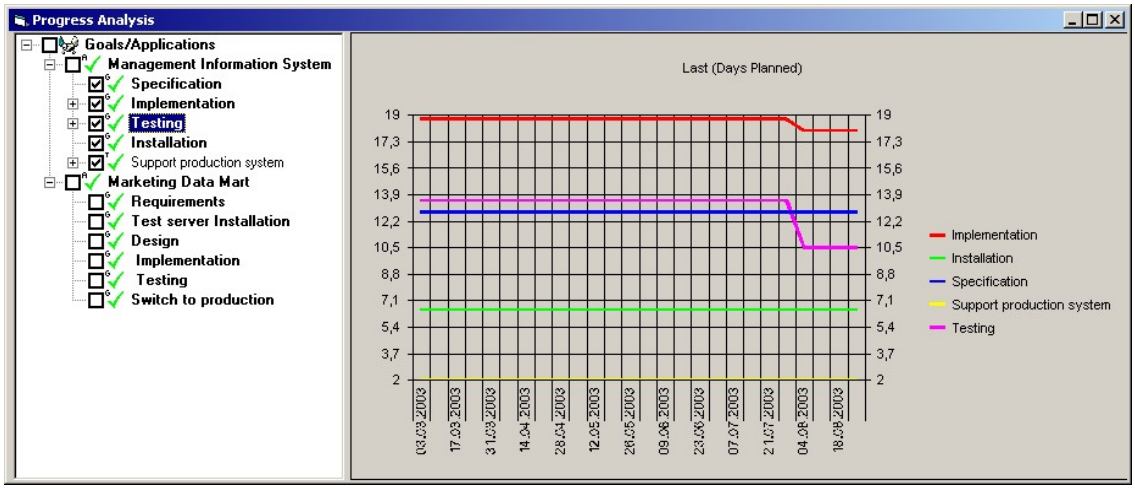
The tool sets no limitation as to how often or in what intervals metrics collection can take place. For projects MIS and MDM, metrics were collected on a weekly basis.

The following sections discuss the different metrics collected and how information is interpreted.

Days Planned:

Days planned graph can be used to track when, where and by how much project planning was changed. From the snapshots above we know that project MIS planned days was changed from 53.5 to 49.75 days; a difference of 3.75 days. By examining the graph we can see that sub goal "Implementation" was reduced by 0.75 days, sub-goal "testing" by 3 days and that the changes took place sometime between 28th July and 4th August (had a smaller metrics collection interval been chosen, then a more accurate statement could have

been made). By drilling down the projects hierarchy we will find out that the plan changes were actually caused by changing individual task status from open to closed (an objects current status is always visible in the tree view).



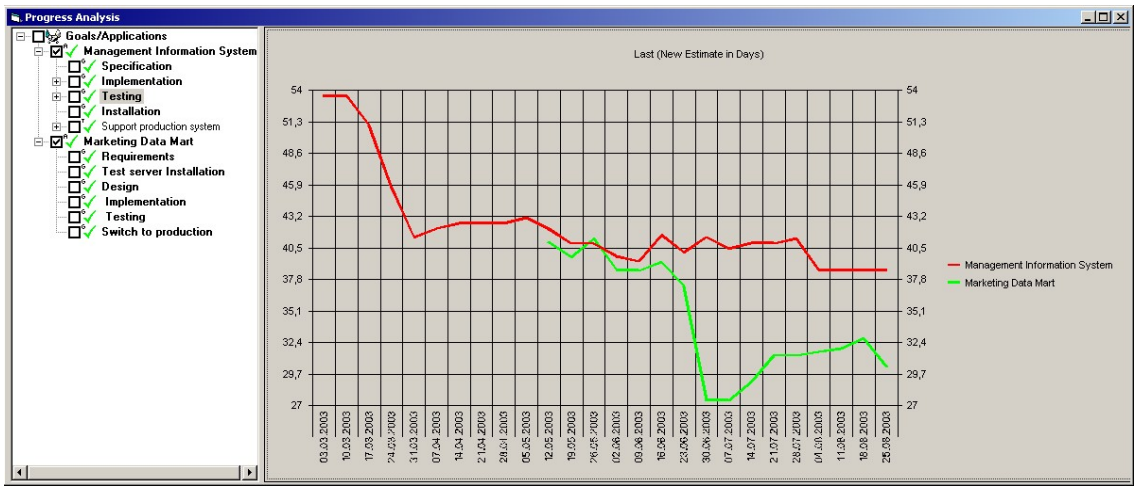
Interpreting the graph:

- Flat line: Project plans unchanged.
- Line pointing up: Project workload increased. Occurs when an object's "Days Planned" is increased or when new tasks are added or when an objects status is changed from "On Hold" to "Open". Drilling down to more detailed levels reveals which actions were taken.
- Line pointing down: Project workload decreased. Occurs when an object's "Days Planned" is decreased or when objects are deleted or an object's status is changed from "Open" to "On Hold" or "Closed". Drilling down to more detailed levels reveals which actions were taken.

Refined estimates:

The next metric we will look at is the tool's automatically refined estimate.

Project MIS took a little over 5 months to complete. It was planned for 53.5 days and completed in 38.6 days. Only 4 weeks after project start, the system's estimate was down to 41.5 days; that is, only 3 days off the actual value. Project MDM was harder to re-estimate; yet, at the half way mark, the refined estimate was less than 3 days off the actual value.



The re-estimation formula, implemented in this version of the tool, only considers tasks that have been started. More accurate results can be achieved by extending the formula to also re-estimate tasks that lie in the future (see Chapter 4).

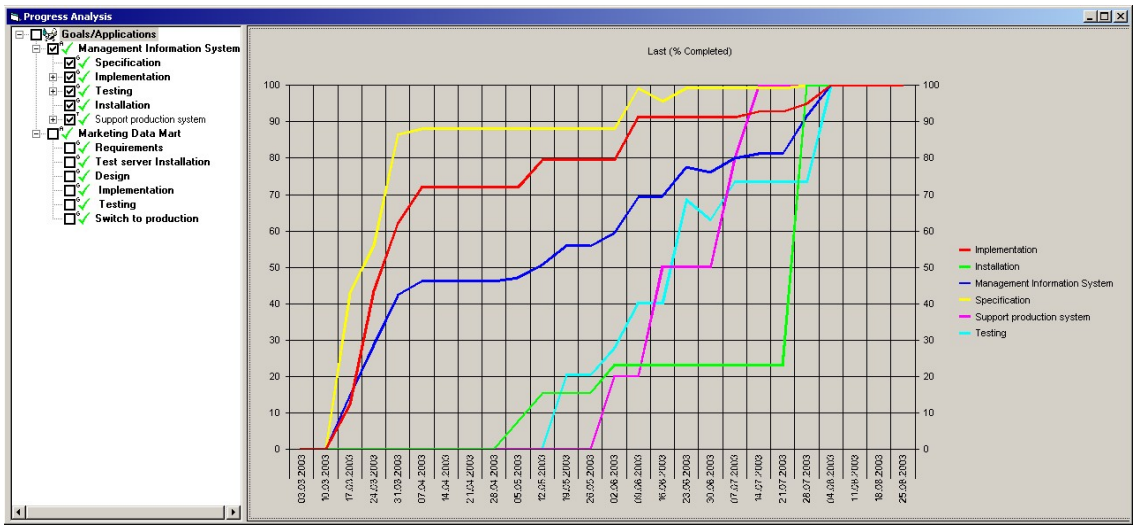
Interpreting the graph:

- Flat line: A flat line is an indication that work is progressing as planned or no work has been performed on the object. Looking at the values in combination with another metric (e.g. “Days Worked”) reveals which of the two cases has occurred.
- Line pointing up: Indicates work is not progressing as planned.
- Line pointing down: Indicates work is progressing better than planned.

However, adding, removing or changing status of tasks will also increase or decrease the “Refined estimate”. Drilling to more detailed levels or looking at other metrics can reveal what actually took place.

% Completed:

Knowing how much work has been completed is one of the questions a project manger has to be able to answer at all times.



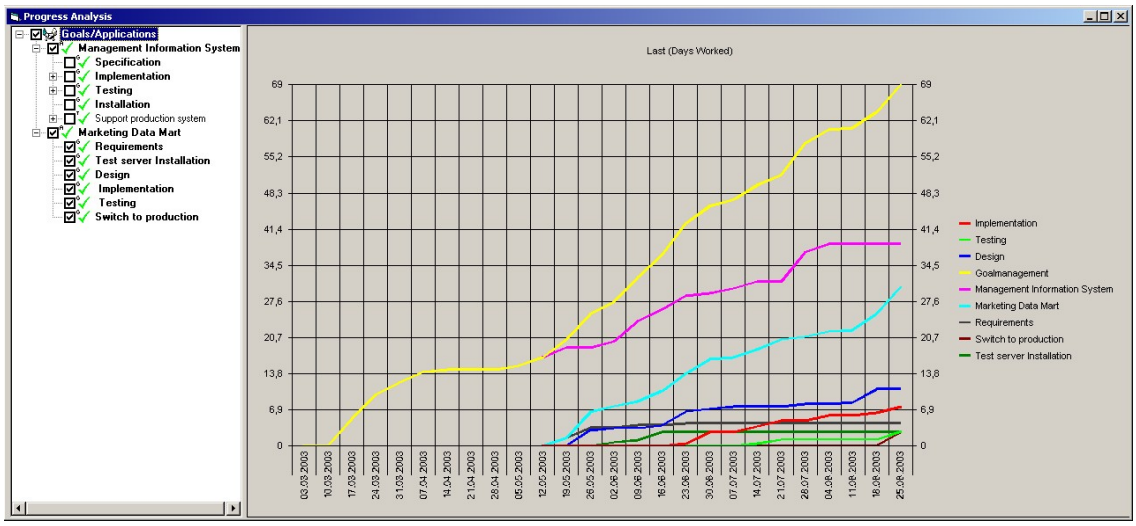
By examining the graph, we can not only tell how much % was completed by what date, but also:

- When work actually started: Line leaves the 0% mark.
- When progress was fast: Line is steep.
- When progress was slow: Line is flat.
- When the project was experiencing problems: Line goes down. Caused when team members lower the percentage completed. Usually happens after finding a bug or requirement changes make it necessary to redo parts of the project, or when team members realize that the project isn't as simple as they anticipated it to be.

A percentage completed line going down is always an indication that something is not as it should be and is worth investigating.

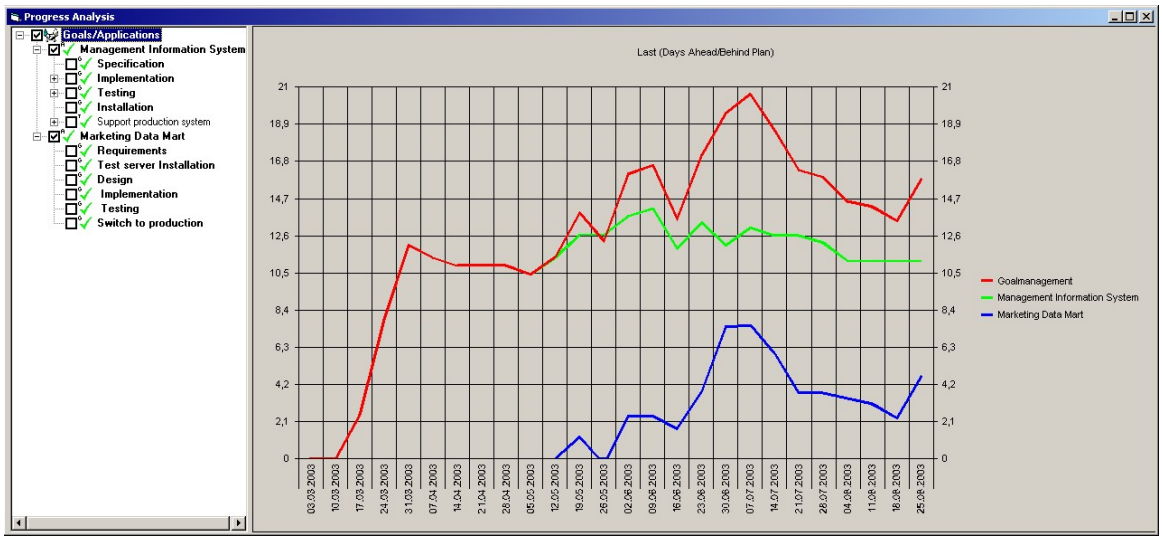
Days worked:

Days worked is another important indicator of how well a project is progressing. Flat lines should be investigated as they indicate that no work is being done. Steep lines mean that a lot of time is invested in an object. However, steep lines do not necessarily mean fast progress; it could simply be that the team has run into problems that are taking a lot of time to solve.



Days ahead/behind schedule:

Another very important question every project manager always has to answer: Are we on schedule? And how much ahead or behind is the project?



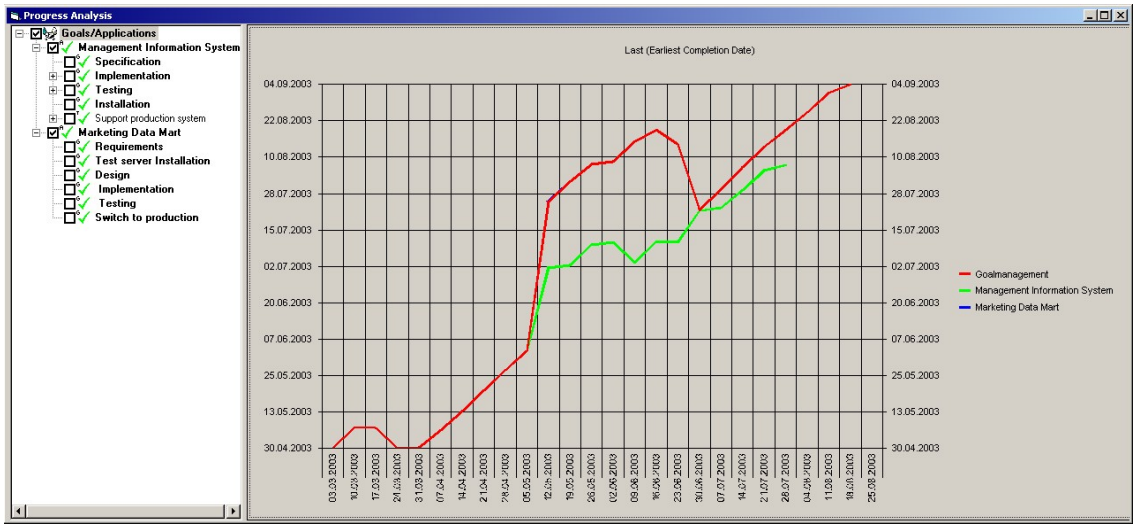
Other questions that can be answered are:

- Where did the team lose time? Line goes down.
- Where did the team gain time? Line goes up.

Care should be taken when interpreting flat lines since they are caused by either steady work or no work at all. Combining the metric with other metrics (e.g. Days Worked) can distinguish between the two possibilities.

Completion date:

Another of the most important questions is, “When will the project be finished?”

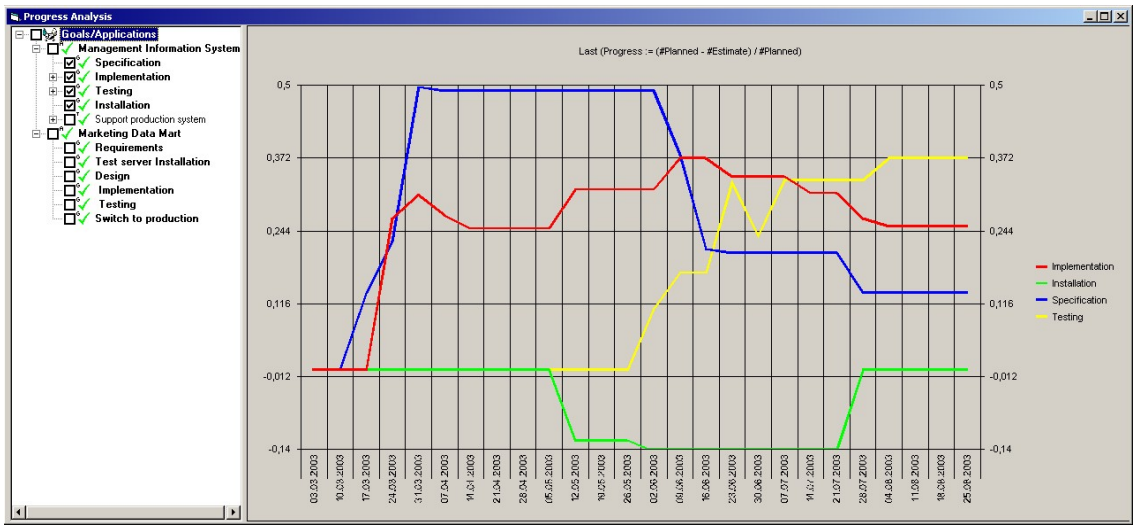


The graph can be interpreted as follows:

- Steady rise, in accordance with metric collection interval, indicates no or little work being done.
- Sharp rise: Work going slow or new tasks have been added or a team member has been assigned to a second project (as was the case between 5th and 12th May).
- Flat curve: Work is progressing as planned.
- Curve pointing down: Indicates work progressing better than planned or that an object’s status has been changed to “On Hold” or ”Closed”.

Progress:

The final metric is useful when comparing objects directly to each other. In Project MIS, sub-goals Implementation, Installation and Specification were all completed better than planned; in comparison with each other, however, testing was completed best.



Commercial systems

In Chapter 1 we identified three types of commercial systems. Hierarchical, timeline and flowchart systems. Almost all systems support hierarchies. We will be looking at two systems, **Microsoft project 2000**, a very well known and widespread system which represents the timeline category, and **TeamFlow7** which represents the flowchart systems. Both systems also support hierarchies.

Microsoft project

Microsoft's recommended approach to setting up a project is as follows:

1. Define the project's start or end date.
2. Define tasks and their durations.
3. Define dependencies between tasks.
4. Assign team members to the tasks.

According to Step 2, the project team is encouraged to start scheduling. Team members are subsequently assigned to tasks. Let us look at the consequences of such an approach by using an example:

> We define Task A as having a duration of 2 days, then we assign a team member to the task. What is the task's workload? If the task fell into a time range whereby the team member was assigned full-time to the project, then the task workload is 2 days. If however, the task fell into a time range whereby the team member was working part-time (50%), then the task workload would be 1 day!! So which is the correct workload? 1

or 2 days? Furthermore, if a second team member is assigned, the workload is doubled!

Isn't a task's workload actually an estimate? So what Microsoft Project is doing, is estimating the project based on the schedule.

Throughout the whole thesis, we have been discussing the importance and advantages (and have also quoted numerous authors, who all agree) of first estimating, and then creating a schedule based on the estimate. An inexperienced team is bound to fail using Microsoft's approach; even a team with great experience is taking a high risk by generating such schedules.

Another drawback is the way team members are handled in Microsoft Project. Assigning a team member to two tasks running parallel would, by default, book the team member 200%.

When monitoring projects, Microsoft's default functionality assumes that all is going according to plan; updating % completed or days worked or time elapsed will update the schedule to keep the project running according to plan. Entering actual start or end dates moves a task forward or backward and as a consequence the whole project's completion date is also moved forward or backwards. A re-estimation (re-scheduling in the case of Microsoft Project) has to be explicitly initiated. Again, as with scheduling without estimation, our discussions throughout have shown that most projects do not progress as planned and it is vital that estimates are refined and schedules are updated regularly.

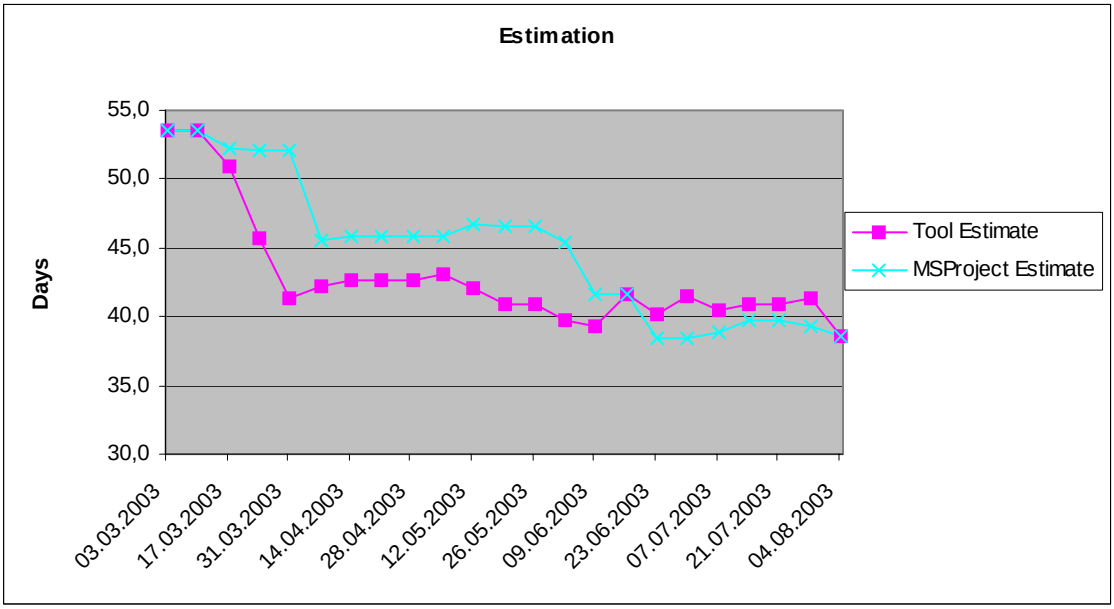
To allow for direct comparisons between MS Project and the tool, MIS was re-managed using MS Project.

Estimation:

As mentioned above, when entering progress information, MS Project assumes work is progressing as planned and updates the plan accordingly. Achieving accurate results while working on a task, is possible, but requires that the team members re-estimate work manually. Such an approach is only feasible when working on smaller projects.

In addition to planned values, MS Project also offers calculated values; after completing work on a task, a user can enter the actual amount of time spent and MS Project will use that value to calculate a project's workload.

Calculated values were used, while managing MIS in MS Project as new estimates. The following graph summarizes the results.



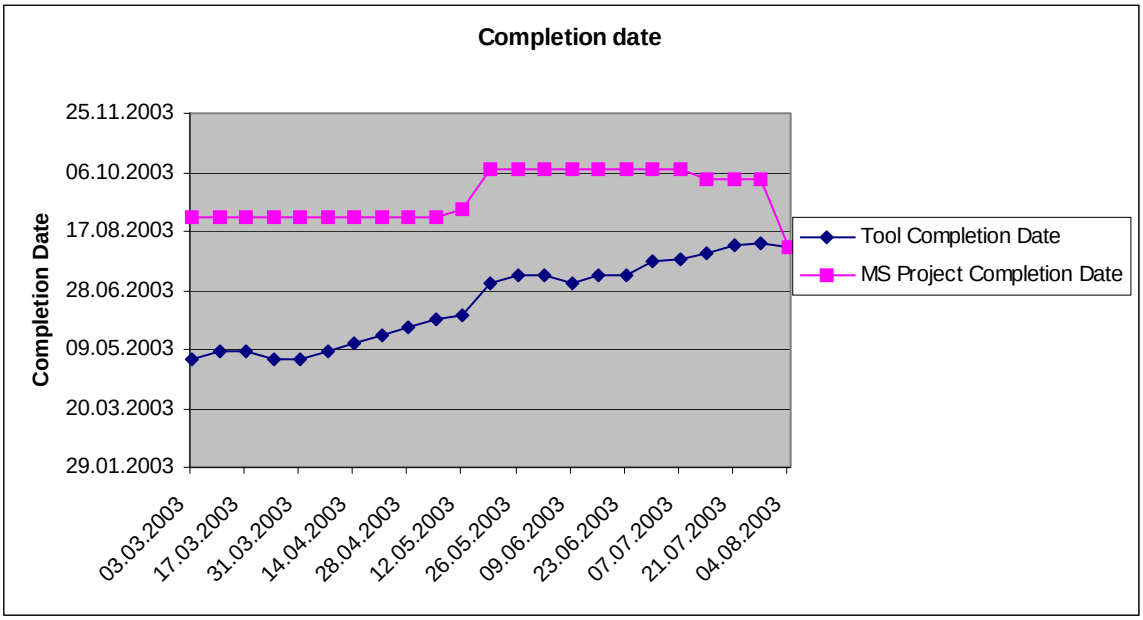
The difference in the two estimation approaches is that MS Project only considers tasks that have been completed; the tool uses the completed tasks as well as all tasks currently being worked on.

Only 4 weeks after project start, the tools estimate was down to 41.5 days, or only 3 days off the actual value. It took MS Project 14 weeks to get to a comparable estimate.

Scheduling:

When comparing calculated completion dates, MS Project produced more stable results. That is because MS Project calculates a critical path and only changes to tasks on the critical path will influence the project's completion date. Furthermore, MS Project offers a variety of scheduling options: differing types of dependencies, earliest or latest start dates, setting durations, etc. which can all help create more accurate schedules, but also increase scheduling overhead.

The tools approach is simpler. It uses priorities to generate a task execution order and combines that with team member working hours to generate start and end dates. Start and end dates cannot be changed and the duration of a task depends on the team members' working hours. The resulting schedule is better described as an earliest (fastest) possible schedule and, unless team members complete tasks as planned, is constantly changing.



Monitoring:

MS Project allows for a comparison of current values to planned values and also offers the possibility of saving up to five different base plans to compare with. These monitoring capabilities may be enough for small projects, but will not provide the required support when working on large projects.

Metric collection and analysis capabilities as offered by the tool (see Tool Evaluation – Metrics) are not possible.

Teamflow

Teamflow prides itself as being “The project manager for the rest of us”. To compare Teamflow to the tool, project MDM was (re-)managed using Teamflow.

Estimation:

Teamflow does not support estimation. The objects (tasks, meetings, etc.) properties dialog offers a duration property, but that is intended for scheduling. Team members’ working hours cannot be defined, so calculating an estimate (duration * %of WorkHoursAssigned) is also not possible.

Scheduling:

Teamflow schedules are generated from flowcharts using dependency and durations as entered by team members. While being simple to learn and use, the scheduling engine is too simple to be useful:

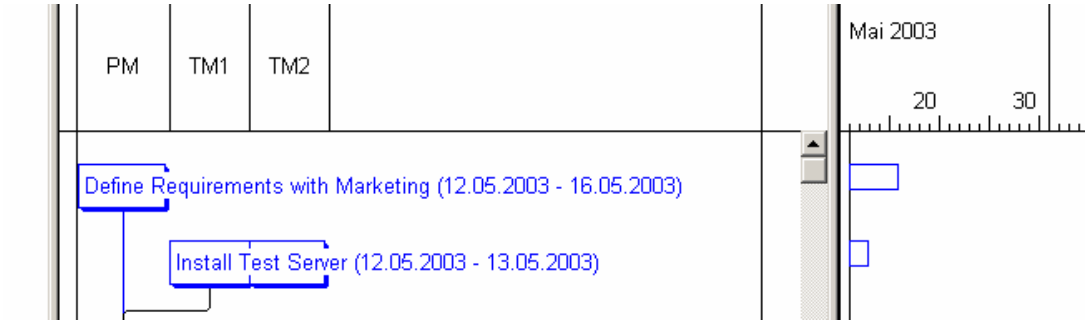
Team members' working hours and project availability cannot be defined and are therefore not considered when scheduling.

Task duration are not adapted when a team member is assigned to several tasks at once or when several team members are assigned a single task. Such adaptations have to be done manually by the project team.

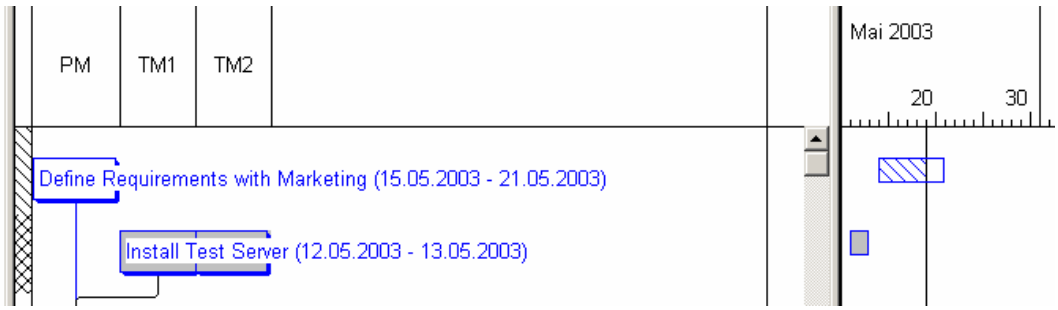
The scheduler only distinguishes between automatically scheduled and actual (manually entered) start dates. Entering a planned start date manually is not possible. This would not be a problem if the scheduler could identify which tasks are being worked on and which are not.

While trying to manage MDM, the following scheduling problem presented itself in the first week:

The first two tasks "Define Requirements with Marketing" and "Install Test Server" had no predecessor, so they both began at the project's start date.



One week into the project, Teamflow presented the following schedule:



"Define Requirements with Marketing"s actual start date was 15.05.2003; when entered, Teamflow calculated the end date properly. However, work had still not

begun on “Install Test Server” and Teamflow had marked the task as completed. Moving the task’s start date forward is only possible by setting the actual start date.

Monitoring:

Teamflow’s project monitoring capabilities are also very weak. Metrics cannot be collected, actual dates overwrite calculated dates and changes to durations are not historicized. Teamflow merely informs the team where they should be.

Teamflow’s advantages lie in its capabilities to visualize processes as flowcharts and in its ease of use. But that alone is not enough to successfully manage a project.

Summary

The two projects managed within the tool illustrated the tool’s flexibility, ease of use, advantages and weaknesses. However, more, and larger, test projects are needed in order to get more comprehensive results.

The re-estimation model produced very good results and can be improved if extended to include the ideas discussed in Chapter 4.

The scheduling engine still needs a fair amount of work before it can be used to generate schedules that can be presented to the customer.

Metrics collection produced very good results, but needs to be refined to avoid misinterpretation.

The next chapter (Conclusions and Outlook) will review test results to determine whether the requirements set in Chapter 3 were fulfilled, and present recommendations for improvement and investigate further possible research areas.

Chapter 7 – Conclusions and Outlook

The goal of this thesis is to discuss some of the problems and requirements facing modern software development and to develop a model and a corresponding tool prototype to aid in developing and refining plans. The focus was on estimation, automatic estimate and schedule refinement and monitoring.

Research and prototype developments were conducted in parallel. A first prototype fulfilling minimal requirements (task list with progress information) was developed using MS Excel and gradually refined and expanded with the research conducted. The final prototype was developed in MS Visual Basic and used a MS Access database for data storage.

The research part started with software development basics. The terms “Software engineering”, “Project” and “Project management” were discussed and then we went on to look at different lifecycle models available, starting with classical models like “Waterfall” and ending with modern agile models and methodologies like “XP”.

The requirements of modern software development as discussed in Chapter 3 identified several important issues:

- Workload estimation: A team sets up a project by estimating the workload (not duration) of individual tasks; tasks can be grouped to define goals and team members can be assigned to tasks. Based on the estimates, priorities and team member availability, a scheduling engine creates project schedules; only then are task durations and probable completion dates visible.
- Automatic estimate and schedule refinement: During a project, progress information entered by the project team is used to automatically refine estimates and schedules.
- Monitoring: A metrics collection engine and a graphical interface were developed. Data analysis is possible at all project levels and between different projects. The monitoring system can be used to determine and compare a project’s status at different times, identify trends and implement an early warning system.

Requirements revisited

Based on the requirements identified in Chapter 3, a software project management approach was developed and put to the test by re-managing two real-life projects. Chapter 6 compared the results to actual values and to values

obtained using commercial tools. The following sections will review the requirements to determine which were fulfilled.

Project management fundamentals

- **Provide the project team (customer, management, developers and all concerned) with information to effectively control the variables time and scope.**

[Beck]'s solution to controlling the four variables of time, scope, cost and quality is by making them visible. Due to time constraints, the tool only concentrated on 2 variables, time and scope.

The hierarchical model used to define goals, sub-goals and tasks allows grouping different parts of the project and provides the team with possibilities to view projects at more general or more detailed levels.

The possibility of changing an object's status and priority also helps in controlling the variables. Setting an object's status to "On Hold" recalculates a project's workload and forecast completion dates, allowing team members to study how changes to certain features of the project will affect the whole project. Changing priorities of objects also recalculates completion dates, again providing the team with the means to analyze effects of focusing on different features.

Estimation and Scheduling Requirements

- **Estimation model that allows breaking down the complexity.**
- **Estimation model that can be extended easily.**
- **Dynamic scheduling based on the estimation, priorities, dependencies and resource availability.**

Both estimation requirements are fulfilled by the tool. The hierarchal structure allows for the breaking up of a project into smaller parts. During a project, new subprojects can be added at any time and the consequences are visible. Even new projects can be added to the structure, allowing project management to take place individually at project level and together at system level. As we saw above (Metrics Refined estimates), re-estimation of our test projects produced very good results. However, the projects were small and did not experience many problems. More test projects, larger in size and with more changes, have to be re-estimated to confirm the results.

Due to time constraints, dependency scheduling was not included in the tool. The schedules created were based on workload estimates, defined priorities and availability of team members. All schedules were created and updated automatically. Almost all authors (e.g. [McConnell]) stress the need to estimate a project and to build a schedule based on the estimate, so manual and duration scheduling (as implemented in Microsoft Project and other tools) were omitted to try to reduce the possibility of creating unrealistic (see Chapter 4) schedules. The resulting schedules were the shortest possible schedules and each new scheduling generated different completion dates. Such schedules can be used internally, but a project manager would have a hard time trying to sell such schedules to a customer.

Extending the model to incorporate dependency scheduling and, even though it is not desirable, allow defining starting dates manually would produce better schedules.

Monitoring and forecasting requirements

- **Monitor estimation and actual progress values on all levels of the project.**
- **Forecast the remaining amount of work and completion dates on all levels of the project.**
- **Automatically collect all possible estimation and scheduling metrics.**

Progress values and refined estimates are always visible at all levels of all projects. Schedules, completion dates and metrics are generated/ collected by the click of a button and the process can be repeated as often as required.

As we saw above, the tools metrics section offers great flexibility and detail in analyzing data; trends can be analyzed and an early warning system can be implemented as well. But the above discussion also showed that the results of one metric alone can be interpreted in several ways; more correct conclusions can be made if the tool were to be extended to allow analyzing several metrics together.

Controlling requirement

- **Project structure has to allow for changes.**
- **Implementing changes has to be quick and easy.**
- **The consequences of changes have to be visible.**

As discussed above, the advantages of a hierarchical structure combined with priorities and different status also fulfill the controlling requirements. Projects can be extended easily by adding new goals or tasks and setting the correct priority. The effects of eliminating sub-goals and tasks can be studied by merely setting the status to “On Hold”.

Lifecycle requirements

- **Support linear, hierarchical and iterative lifecycles.**
- **Support multiple projects.**
- **Support projects using a mix of different lifecycle structures.**

Hierarchical lifecycle models are the primary type of model supported. Linear models can be implemented easily by defining priorities. Iterative models are also supported, but not to the same degree as hierarchal and linear ones. Implementing iterative models requires that the team updates priorities to initiate new iterations.

Due to the size of the test projects and teams, a hierarchal model was sufficient. Linear models, even though they are found in literature, are not suited to software development; both test projects could not be developed using a linear model. Iterative models are more suited to bigger projects; further test projects are needed to see whether iterations can be successfully implemented using the tool.

People requirements

- **Team members cannot be overbooked; overtime cannot be planned.**

By only supporting automatic scheduling, a team member is never scheduled to work outside the working hours defined. However, the resulting schedules were the shortest possible schedules. The team member model should be extended to include holidays and non working periods.

Secondary requirements

- **Keep the tool as simple as possible.**

The tool consists of two main windows, one to define and manage projects and one to perform detailed analysis of collected metrics. To facilitate ease of use,

information is either color and symbol coded (management window) or presented graphically (line graphs in analysis window).

Conclusions and Outlook

The results presented in Chapter 6 illustrate how the process of management can be eased by adopting the approach developed. Of the three main focus areas, estimation and monitoring delivered very good results; the generated completion dates however, changed too often.

Estimation

During our discussion (see Chapter 4) we identified workload as an estimation indicator. The results presented in Chapter 6 confirmed the usefulness of such an indicator. While researching literature (e.g. [McConnell]), we also saw range estimates as a means of reducing uncertainties in a project's beginning phases. However, implementing ranges will have an effect on all other areas of the approach. How can an object's (goal, task) status be assessed? How will schedules be generated? How can re-estimation be implemented? Will ranges be narrowed down automatically as project progresses? …

Re-Estimation

The re-estimation algorithm implemented only considers tasks that have been completed or are currently in progress. Still, the automatically generated re-estimates converged early to actual values, especially on project MIS. Project MDMs results were good, but not as encouraging as MIS. This is because the algorithm works better for projects that produce steady progress.

Chapter 4 discussed a slip/gain factor which was however, due to time constraints, not implemented in the prototype. Extending the approach to incorporate a Slipgain factor might increase the accuracy achieved and speed up convergence to real values. Further research areas could include:

- Using non-linear (the current implementation only sums up values) algorithms to aggregate values up the project hierarchy.
- Extending the model to include trends identified by analyzing historical data.
- Including individual team member achievements.

Scheduling

The scheduling engine still needs a fair amount of work before it can be used to generate schedules that can be presented to a customer. The generated schedules were the shortest possible schedules and each new scheduling generated different completion dates. Such schedules can be used internally, but a project manager would have a hard time trying to sell such schedules to a customer. However, the combination of hierarchical projects' structures and priorities resulted in very flexible schedules that allow rescheduling to be performed by merely changing a few priorities.

Further research areas that could produce more stable schedules include:

- Extending the model to combine dependency and priority scheduling.
- Considering buffers while generating schedules; buffer size could take into account project stage, task size, team member availability, etc.
- Including more detail in team member working hours modelling.

Metrics (Monitoring)

Metrics collection produced very good results. Data collection is performed by the click of a button and the process can be repeated as often as required.

The tool offers great flexibility and detail in analyzing data:

- Values are visible at all project levels.
- An individual metric can be compared between different tasks, goals and even different projects in one graph.
- Trends can be analyzed.
- An early warning system can be implemented.

However, the analysis performed in Chapter 6 also showed that the results of one metric alone can be interpreted in several ways. More research is required to avoid data misinterpretation. This could include finding new metrics, combing metrics to create new ones, and allowing only certain combinations of metrics to be viewed together.

Data Visualization

Even though data visualization was not an explicit requirement, the data presented in the application's main window and the metrics analysis module simplified the application considerably, thus increasing the chances of acceptance by users.

However, the scheduling reports were not of good quality. A lot can be done by presenting schedules graphically. To what degree the already implemented visualizations increase user acceptance and what other techniques can be implemented is a further research area resulting from this thesis.

A starting point for schedule presentation could be a 3-Dimensional Gantt graph having a timeline on the X-axis, team members on the Y-axis, priorities on the Z-axis and dependencies between objects presented by connecting objects. Such a graph would allow viewing individual team member schedules as well as overall project schedules simultaneously and combines priority and dependency scheduling in one view. If accepted by users, the chart can be extended to include color coded status information and fill patterns to indicate completion levels.

Bibliography

Abdel-Hamid Tarek, “Software project dynamics : an integrated approach”, Prentice-Hall, 1991

“AGILE METHODOLOGIES Survey Results“, SHINE TECHNOLOGIES, 2002

Balzert Helmut, “Lehrbuch der Software-Technik. Band 2“, Spektrum , Akad. Verl., 1998

Biffl Stefan, Grechenig Thomas, Köhle Monika, Zuser Wolfgang, “Software Engineering mit UML und dem Unified Process”; Pearson Studium, 2001

Beck Kent, “Extreme Programming”, Addison-Wesley, 2000

Boehm, Barry W., “Software cost estimation with Cocomo II ”, Prentice Hall, 2000

Boehm Barry w., ”Software Engineering Economics“, Prentice Hall, 1981

Brooks Frederick P. Jr., “The mythical man-month. Essays on software engineering”, Addison-Wesley Publishing Company, Jan 1982

Brown William J. Malveau Raphael C., McCormik Hays W., Mowbray Thomas J., “Anti Patterns, Refactoring Software, Architectures, and Projects in Crisis”, John Wiley & Sons 1998

Cusumano Michael, Selby Richard, “How Microsoft Builds Software”, Communications of the ACM, June 1997/Vol. 40, No. 6

Dornwaß Eric, “Softwaremanagement in Unternehmen- Konzeption und Planung“, VDE Verlag GmbH. 2002

Doucette, Martin, “MS Project 2000 für Dummies“, Bonn, 2000

Dumke Reiner, “Software Engineering, Eine Einführung für Informatiker und Ingenieure: Systeme, Erfahrungen, Methoden, Tools”; Vieweg & Sohn VerlagGmbH, 2001

Gärtner Johannes, “Das Problem ist ein Hund: Essayskizzen zum cleveren Projektdesign & Projektmanagement“, Wien 2003

Gareis Roland Univ.Prof.Dkfm.Dr., “PM baseline - Knowledge Elements for Project and Programme Management and for the Management of Project-oriented Organisations” Project management Austria, 2002

Hemamalini Suresh,“Change Management: Must for today’s Organization”, © Think Business Networks Pvt. Ltd., July 2001

IEEE Standard Glossary, 1990, in Encyclopaedia of Software Engineering, Vol. 2, 1994, P. 1177

Kimm, Koch, Simonsmeier, Tontsch, “Einführung in Software Engineering“Walter de Gryter, 1979

Kolawa Adam, “The ABCs of XP, RAD and PSP”, 2002

Kulik Peter, “How to Prevent Surprises in Software Projects”, KLCI, 08/1998

Kulik Peter, “Software Project Success Factors”, KLCI, 02/1997

Kulik, Peter, “Software Development Rules of Thumb”, KLCI, 02/1996

Kulik, Peter, “Improving Software Development Processes – Without Sacrificing Projects!”, KLCI, 04/1996

Kulik, Peter, “A Practical Approach to Software Metrics”, Institute of Electrical and Electronics Engineering Inc., 02/2000

Litke , Hans-D., ”DV-Projektmanagement : Zeit und Kosten richtig einschätzen“, München, Wien, 1996

Lutz Heinrich, “Management von Informatik-Projekten ”; München, Wien; Oldenburg, 1997

Mahnke Hans, „Software Engineering kurz und bündig“, Chipwissen, 1986

McConnell Steve, “ Rapid Development: Taming Wild Software Schedules”, Microsoft Press, 1996

McConnell Steve, “Software Project Survival Guide”, Microsoft Press 1998

Merkl ao.Univ.Prof. Dr. Dieter, „Projektmanagement“ TU Wien, 2002

Portny Stanley E., „Projekt-management für Dummies“; Bonn, mitp-Verlag, 2001

Project management Institute, “Project Management: A proven process for success”, http://www.pmi.org/projectmanagement Project management Institute, 2000

Project management today, “So what is project management?” http://www.pmtoday.co.uk/

Ricketts Ian W., “Managing your software project. A student’s guide”, Springer 1998

Shaw Mary, Garlan David, „ Software Architecture“, Prentice Hall, 1996

Tjoa A Min: Skriptum zur Vorlesung „Softwareprojekt-Management“; TU Wien, Wien:1997

Trauring Aron, „Software Methodologies: Battle of the Gurus“, Info-tech research group, 2002

Versteegen Gerhard, „Software-Management“; Springer, 2002

Vijay Shankar, “Estimation of effort using Function Points”, Cognizant Technology Solutions, Feb 2003.