Tallinn University
Institute of Informatics

Software Project Management

Peeter Normak

Tallinn 2014

Contents

Introduction

For a company to survive in a market economy it needs not only to react quickly to the changing needs of its customers, but also to do so effectively and adequately. Moreover, ICT itself increasingly becomes a source of innovation. This means that ICT experts should take a proactive role in offering innovative solutions to be implemented in different areas. It is estimated that about half of the productivity increase in last years has resulted from ICT implementations. Below we will discuss some important aspects of ICT projects.

The following is based mostly on the personal experience gained during preparing and executing different types of projects, therefore, a majority of the examples deal with projects that were executed in Estonia.

1 Software projects

It became evident during the seminars for informatics students that were held in Estonian companies in years 2000-2002 that there is a huge need for ICT project managers, particularly in software development. The lack of good project managers and managers in general was not limited to commercial companies. For example, the report of an international evaluation committee on computer science in Estonia (2001) found that the lack of an IT research coordinator at one of the Estonian universities should be considered as a serious obstacle to quality improvement: small research groups that do not coordinate their research activities cannot become internationally competitive.

1.1. Specific character of software projects

Until the second half of 1980s, most software was developed to be run on a single computer without interoperability possibilities between different software solutions, platforms or computers; computer security was almost not an issue. This is no longer the case today. Most software operates on networked computers, interacts with different devices and conforms to the different roles of the users. Therefore, modern software is extremely complex. And software development depends on the developers' creativity, motivation, focusing and the ability to withstand heavy workload. The complexity of modern software is also the main reason why software projects have a relatively high-risk factor and low success rate.

Studies of software projects from the mid-1990s have revealed the following:

- Only about 15% projects were completed on time and did not exceed their budget;
- Project failure was caused mainly by the low quality of process management;
- The level of rework was the main factor in determining the maturity of the software processes.

Continually increasing problems in software development led to a common understanding at the beginning of 1990s that software development in general was in crisis.

Next, we list some aspects that should be taken into account when planning and executing software development projects:

1. The software solutions should support the main processes of the institutions that are using these solutions. The software should also support the effective use of personnel and other resources. This assumes that the developers have competences in other areas (psychology, sociology, cognitive sciences etc.).
2. As ICT solutions are usually used by diverse users, the needs of a wide user group (which can sometimes be contradictory) need to be satisfied. Therefore, users should be involved in software development.
3. Changes in society and industry prescribe the quick implementation of software products and the possibility of adapting them to different needs. The needs at the end of a project can sometimes greatly differ from those agreed upon at the beginning of the project.

Some aspects, which may not be significant in other areas, can be decisive for software projects. For example, the productivity of software developers is higher if they work in small rooms designed for one or two persons. There have been cases when companies have been forced to fire a good specialist who constantly disturbed his or her colleagues while they were working.

These and some other aspects of IT projects cause certain uncertainty and higher risks compared to the projects in other areas. Although there different software process models and standards have been developed, according to the "Extreme Chaos" document (2001) of The Standish Group, the average success rate for software projects during 1996-2000 remained almost unchanged. This means that the pace of the increase in software complexity was at almost the same level as the improvement of the skills of project managers and project team members.

Software projects are output oriented. If a training project fails it only affects a small number of people, however, a few mistakes in a software project may affect a huge number of the software users.

Different conditions that influence the projects may vary a great deal; procedures that are successful in certain conditions may not be applicable if the conditions change. In many cases, case studies can be helpful in optimising software development, (see, for example, *www.cse.dcu.ie/spire/case.html*).

As in many other professions, software project managers have formed a network called *Software Program Managers Network* (http://*www.spmn.com*).

Exercises

1. Based on available sources (for example, http://www.it-cortex.com/Examples_f.htm, and http://www.codinghorror.com/blog/archives/000588.html), find out the most common factors that cause ICT projects to fail. Compare these causes with the lists of failure factors for arbitrary projects (see, for example, www.projectsmart.co.uk/seven_rules_to_guarantee_project_failure.htm).
2. Analyse the job descriptions for the profession of *IT project manager*. List the most important competences represented in these job descriptions.

1.2. Critical success factors for software projects

Critical success factors and critical failure factors are strongly related to each other. In most cases, the absence of a success factor can be considered a failure factor. The following factors are often considered as the most significant success/failure factors of software projects:

1. ***User involvement.*** A 1995 study by *The Standish Group,* which was based on an analysis of 8,380 software projects, showed that user involvement was the most significant success factor and *vice versa* – insufficient user involvement was the most significant failure factor. The fact that the users' wishes and needs often change in the course of a project should be considered already in a project's planning phase. The users should be involved from the very beginning of the project, by employing the relevant means (informing the users about the project, including users in the project team, inviting users to project meetings, etc.).
2. ***Change management.*** In addition to the users, upper management, donors and other institutions/persons may require changes. The total amount of work may increase considerably if the changes are performed in a non-coordinated or non-systematic way, or the changes are not properly documented. The general conclusion of the *The Standish Group* study mentioned above was that the software project management is chaotic (*www.standishgroup.com/sample_research/index.php*, registration is needed).
3. ***Quality assurance.*** The testing and correction of mistakes can turn to an endless task if the mistakes are not corrected immediately after they are detected, since mistakes that are detected at the initial phase of the project may even be forgotten later. To ensure high quality, one should try to achieve justified simplicity in all the phases of a software project.
4. ***Integration of components.*** If the integration of the components developed by different designers is made in a later phase of a project, the additional work required for the development of interfaces that can assure the interoperability of components may take too much time.
5. ***Adequate corrections of the schedule.*** If the schedule is not adhered to (which occurs very often) priority will be given to personal deadlines. Time for communication with the users, upper management, testers etc. will be reduced. The quality of project's coordination will suffer and the risk that new problems will emerge will increase.

Conclusion: increased process management during the initial phase of a project can avoid huge cost overruns for correcting mistakes in the final phase of the project; problems should be solved immediately!

For example, at the NASA Software Engineering Laboratory software development costs were decreased by an average of 50%, and the number of mistakes by 75%, during the period of 8-years of applying Software Process Improvement (SPI) method. The application of SPI improves the general work culture as well: in companies where SPI is not used only 20% of workers consider the work culture in their institution to be high (comparing to 60% in institutions that utilise SPI) [McConnell 1997, p. 26-27].

A rough estimation is that the costs for correcting an error/bug that was made while determining requirements during the final phase of the project are two degrees higher (that is, about 100 times higher) than the error being corrected immediately. The reason is that decisions made in initial phase are more important than the ones made in the final phase of a project (for example, what platform will be used is decided in the initial phase; how to design the help information in the final phase).

Although it is usually impossible to assess the quality of software in the initial phase of a project (because the number of decisions that are not made to that time is too big), the general opinion is that the success or failure of a software project will be determined during the first 10% of a project. This again emphasises the importance of the initial phase of a project.

The development procedures – general structure of the activities and principles for performing them – are also formed during the initial phase of a software project.

1.3. Structure of the software process

The activities of software development can be divided into certain type of activities (*SWEBOK – The Guide to the Software Engineering Body of Knowledge* – uses the concept of *knowledge areas*). The most general and probably the most frequently used model is based on the *ADDIE* model, in which *A* means *analyse*, *D – design*, *D – development*, *I – implementation*, *E – evaluation*. The *ADDIE* model can be applied to small-scale software projects. This model is not suitable for large-scale software projects because:

- these five types of activities do not adequately describe enough of the activities related to software development;
- the linear order that is assumed for performing the activities in *ADDIE* model is usually not effective enough.

The phases of the software development life cycle based ADDIE model are defined as follows: the determination of the requirements of the software, software design, coding, testing and implementation.

Determination of requirements means determining the functional and non-functional specifications of the system. The possibilities of using previously developed (sub-) systems should also be taken into account.

Software design means composing software architecture, determining the input/output interfaces and technologies to be used. This phase can be divided into two sub-phases: general design and detailed design.

Coding also means the integration of the modules, composition of a test plan and a preliminary version of the user documentation.

Testing is for checking conformity with the specified requirements. The final version of user documentation will be prepared.

Implementation of software depends to a great degree on the type of software (for example, was it developed for a certain company or as general-purpose software for wholesaling).

These phases intersect partly on a time scale: the determination and analysis of requirements predominate at the beginning of a project and can almost be nonexistent during the final part of the project. The structure of activities depends on the development model that is used.

The design needs to be more detailed when more complicated projects and when the developers are less experienced. It is better to place a little bit more emphasis on design – the reduced risks compensate for the higher design costs. The detailed design consists of a series of design diagrams with the possible inclusion of some pieces of code.

Different designers can use different classifications. For example, *SWEBOK* considers 15 knowledge areas; these in turn are divided into subareas/topics (for example, the requirements phase is divided into ten topics including the following four: 1) requirements elicitation, 2) requirements analysis, 3) requirements specification, 4) requirements validation.

Different sets of indicators are developed that allow the progress of a software project to be evaluated (see, for example, *www.construx.com/survivalguide*).

Different software development models are considered in the next chapter. In this chapter, we will consider some aspects that are common to almost all phases. The technologies used for software development is a subject for a software engineering course.

1.4. Preliminary planning of a software project

During the preliminary planning for software, an initial software development plan will be developed (initial version of a Charter). This plan usually contains the following:

- The vision of the project,
- The main authority/decision maker,
- The objectives of the project,
- The main risks,
- Personnel needs,
- Estimated duration of the project.

The vision of the project. Before initiating a software project, a clear vision about the project's main goal should be formulated. This should be ambitious and realistic. The worse that can happen is that the project team realises that the main goal is not realistic and their motivation becomes very low. *For example, the development of MS Word for Windows 1.0 took five years (instead of one year that was planned initially) because the goals were set too high.* The main goal "To develop the best text processing software in the world" is too general; "Develop the most user-friendly text processing software in the world" would be much better because it provides some guidelines for what the software should and should not be.

Determination of the main authority/decision maker. The main decision maker can be one person or a group of people (project board). In case of a project board, it should consist of representatives of all the major interest groups.

Determination of the objectives of the project. The objectives can be changed or concretised during the project execution depending on the resources available, on the progress of the project, etc. *For example, NASA SEL (Software Engineering Laboratory) concretises the objectives of its projects up to five times, taking into account certain adjustment coefficients according to the following table:*

The coefficients are determined	***Upper limit***	***Lower limit***
After specification of requirements	*2.0*	*0.5*
After requirements analysis	*1.75*	*0.57*
After general design	*1.4*	*0.71*
After detailed design	*1.25*	*0.80*
After coding	*1.1*	*0.91*
After testing	*1.05*	*0.95*

To increase the adequacy of the objectives and work plans, the whole project team should be involved. If the project team does not accept the plans, they will usually not be followed. The progress of a project should be available to the project team (for example, through the Intranet), with information on at least the following:

- List of completed tasks
- Error statistics
- List of basic risks
- Completion rate of the project (by duration and/or by resources)
- Project reports

Risk management and personnel management are discussed in separate sections.

When the decision is being made on whether or not to prepare and submit a bid, all essential aspects of the possible project should be analysed. For example, John M. Smith recommends [Smith, 2001] that an Opportunity Qualification Worksheet is compiled that includes the following eleven questions to be answered/evaluated on a scale of zero (low) to 5 (high):

1. Tangible requirements?
2. Aligned with your strategy?
3. Relationship with this project?
4. Good solution?
5. Effort available to bid and execute?
6. Time available to prepare a winning bid?
7. Size of budget known/adequate?
8. Competition known + strengths/weaknesses?
9. Only me! Do you have uniques?
10. Price that will win?
11. Engagement with prospect possible?

It is most important to have high marks to the questions 3, 4, 9, 10 and 11. Before the decision is made, the following questions should also be answered:

1. We will win because …
2. If we lose, it will because …
3. The top three risks are …

The answers to these questions help you to emphasise your strengths, eliminate the reason why you might lose and mitigate the possible risks.

1.5. Needs for personnel

Depending on the software development model, the needs for personnel varies a great deal during the development process, both in terms of the number of people and their qualifications. Higher quality and more experience are needed during the initial phase. For example, the distribution of the total workload and its duration can be the following:

	Work load	Duration
Determination and analysis of requirements	10%	15%
General design	10%	15%
Detailed design	20%	20%
Coding	30%	25%
Testing	20%	15%
Implementation	10%	10%

In fact, it is very difficult to measure the duration of the phases because the activities that belong to different phases are sometimes combined and performed simultaneously (for example, a small piece of new code that is tested immediately by the developer). The corresponding numbers in two columns indicate the personnel needs. For example, the fact that general workload for the general design (10%) is smaller that the duration of the general design (15%) means there is a relatively smaller number of people involved in the general design. It is estimated that in the case of a classical *waterfall* development model testing can take up to 40% of the total workload. For a larger software projects, the *Brooks* rule for estimating the duration can be applied: 1/3 of the total time will be devoted to planning, 1/6 – coding, 1/4 – testing and integration, 1/4 – integration testing.

Personnel costs are usually the largest costs in software development projects. Therefore, the project team should be assembled with care; personnel changes in the later phases can be very expensive (according a study conducted in the U.S. in 1990, the replacement of a single software developer during a software project cost $20,000 to $100,000).

Another factor that determines the personnel needs is the quality of the people involved. However, it is relatively complicated to consider this aspect because actual practices may not necessarily be connected to the knowledge and skills of the people, as was clearly proved by Gunnar Piho in his master's thesis. For example, about 95% of IT specialists agreed that a holistic quality system is necessary; but in fact, only few software companies have one.

In order to reduce the risk of time overruns, some experts suggest planning only a maximum of 75% of the available resources (including personnel).

1.6. Personnel management

In this section, we will discuss the aspects of effective personnel management. The basic idea is that improvement of the quality of human resources is considered as one of the project's outcome, and consequently, that the project manager is responsible for that. A company will have considerable losses if, for example, five people leave a project/company because of poor management. This is why the following aspects are important:

- Every team member should have opportunities for his/her professional development;
- The ability to keep/consolidate a project team is one of the quality indicators of a project manager.

It is recommended that greater efforts be made to find competent team members rather than hiring inexperienced people and hoping for their professional development. According to widespread opinion, top-quality developers are up to ten times more effective than low-quality ones. A study (B. Lakhanpal, *Information and Software Technology*, 35 (8), 468-73) of 31 software development teams showed that harmony among the project team members is the most significant success factor for a software project, i.e. how smoothly project team members cooperate. This is why people that cause problems when interacting with their colleagues should not be included on a project team, even if they are good experts. Another study (Carl E. Larson, Frank M. J. LaFasto), which was based on the analysis of 75 projects, showed that the poor ability of problematic people to solve problems was the greatest weakness of project managers.

A project manager should have full authority to put together the project team and should not immediately agree to accept the people that are recommended, for example, by upper management.

Roger Woolfe of Gartner Group recommends that preference be given to personality traits when choosing project team members; personality traits are difficult to change while technical skills can be acquired relatively quickly. He proposes 25 key competences for an IT organisation, ten of which are personal competences (the other six describe technical skills and nine describe business processes).

The different roles of people should also be taken into account when assembling a project team; all the basic roles should be represented. Rob Thomsett [Thomsett, 1990] defines the following eight basic roles:

1. *Chairman*: determines the basic methods of project execution; is able to determine the strengths and weaknesses of a project team and the most effective usage of every single person.

2. *Shaper*: formulates the results of discussions and other joint activities; this role has usually assumed by the project manager or lead designer.
3. *Plant*: suggests new ideas and strategies, tries to implement new technology and find new solutions.
4. *Monitor-evaluator*: analyses the possibilities to solve problems as well as the suitability to implement new technologies.
5. *Company worker*: executes the tasks; most of the analysts, programmers, testers, etc. belong to this category.
6. *Team worker*: helps and motivates the team members, tries to improve interpersonal relations and strengthen the team spirit.
7. *Resource investigator*: organises communications with the partners outside the project team, tries to find additional resources; has personal contacts with a broad range of people that he/she also intensively exploits.
8. *Completer*: observes and motivates the project team members to be goal-oriented, tries to minimise the occurrence of mistakes and the domination of personal interests over the project's interests.

Make sure that the roles that project people had/have in some other context will not dominate when forming the project team. *For example, at Victoria University (Melbourne, Australia) students formed the project teams for performing a project in software engineering themselves; as a rule the teams consisted of groups of friends. The success rates of the projects varied a great deal because the roles in the friendship communities and software engineering are quite different. In subsequent years when the project teams were assembled by the university teacher, the success rate was considerably higher.*

Effective personnel management also assumes effective time management. The main tool here is monitoring the time usage by the project team members, especially during the initial phase of a project. This improves the quality of the time estimation for further activities as well as in planning new projects. An example of time usage categories (that is, activities that will be measured and analysed) can be found at *www.construx.com/survivalguide*. Special software for time management has also been developed. Sometimes – for solving an urgent problem – it is recommended that a small (1-2 persons) temporary "tiger team" be formed that freed up from other duties for that time. Here the possible different attitudes of people should be considered -- some perceive membership in a "tiger team" as a promotion, others as disrupting their main tasks.

Exercises

1. Based on the document "*IEEE Standard for Software Project Management Plans*" (IEEE STD 1058.1-1987) of IEEE (*Institute for Electrical and Electronic Engineers*), compose its commented summary.
2. What are the basic differences of abovementioned roles proposed by R.Thomsett of those proposed by R. M. Belbin (*Management Teams: why they succeed or fail*. 1999. Oxford; Boston: Butterworth-Heinemann)?

1.7. Change management

It became evident in the 1990s that project planning should last throughout the project's entire life cycle. it was more and more difficult, and even not useful, to follow the initially planned activities and procedures exactly. Changes in the market, in technology usage and the needs of customers should be taken into account throughout the project's lifetime. Although most of the changes concern source code, it is recommended that the changes in management principles be applied to all project activities. It is also recommended that a special *change management board* be formed, especially for large projects. The board should include representatives of all the interested parties, from both the project team and outside the project team. The main task of the board should be approval (or rejection) of change proposals.

The planning and implementation of changes should be performed according to a commonly accepted and transparent scheme like, for example:

- Initiators of a change should compose and submit a written *Proposal for a Change* that describes the reasons why changes are needed and what the expected results will be once the changes are implemented,
- The board forwards the proposal to the concerned parties for acceptance or suggestions for improvement, accompanied by the comments/suggestions/opinions of the board; the concerned parties are also asked to estimate the possible costs and benefits from their point of view;
- Based on the feedback, the board will draw a conclusion and inform the concerned parties. The board should consider a variety of factors, for example: 1) how the changes affect the project's schedule, quality and distribution of resources (for example, will the workloads of busy people increase or not); 2) whether it will be reasonable to postpone the changes; 3) what – if at all – are the possible additional risks etc.

For example, the list of documents considered by the board can be the following:

1. Change management plan	8. The basic risks
2. Proposals for changes	9. Software tests
3. The main objective of software	10. Graphical and other media elements
4. General software development plan	11. User interface prototype
5. Plans for phases/stages	12. User manual
6. Coding standards	13. Installation program
7. Quality assurance plan	14. Software delivery check list

Systematic change management has a number of advantages:

- Solutions used for software development are more broadly discussed and motivated (a rejection should be explained as well!) and software developers are safe from unjustified demands being made by upper management;
- The project's progress is constantly monitored; for example, a nonworking version cannot be declared as finished,
- The project is documented more completely, and the documents are available to the project team;
- Change management encourages cooperation between the project team members, with the people discussing each other problems.

1.8. Risk management

Most projects do not pay enough attention to the reduction of risks. Experts recommend that about 5% to 10% of project resources be spent on risk management. Higher costs for risk management would make the project too bureaucratic and clumsy, and negatively affect the quality of the project's outcomes (as fewer resources would be available for software development). Risk management is based on a risk management plan that is regularly renewed/updated. For mid-size and large projects, it is recommended that a risk manager be hired. Risk manager should be required to prepare regular (e.g. biweekly) lists of the most significant current risks as well as possible solutions for the reduction of these risks.

There is no reliable methodology for measuring/assessing the risks; expert evidence is mostly used.

Example 6.1. A table of the most significant risks.

Nr.	How many weeks in the list	Name of the risk	Possibilities for risk reduction
1.	4	Low quality of software	A prototype of a user interface is developed to assure user's satisfaction. A systematic development process model will be used. Testing will cover the entire functionality. Independent testers will test the system.
2.	4	Schedule not adhered to	The schedule will regularly be updated. Active process monitoring reveals shifts in schedule A suitable software development model will be used.
3.	1	High expenses	Detailed planning creates clear expectations. The environment supports high productivity, motivation and saving.
4.	5	The premises are used ineffectively	After the prototype is developed, new rooms will be rented.

Additional columns can be included in the table, for example, a column for the people who will be involved in reducing the risks or the causes of the risks. For each risk, a document that answers the relevant questions should be drawn up. This could include the following items:

1. The probability that the risk will be realised and its consequences.
2. Possible ways to reduce the risk.
3. Concrete steps and measures to be taken if the risk cannot be reduced.
4. Persons responsible for performing the measures.
5. Deadlines.
6. Resources required for risk reduction, for each step/measure separately.

For risk management, it is vitally important to monitor the progress of the project. Sometimes people are not willing to articulate the problems; therefore, it may be useful to create an anonymous or hidden channel for delivering "bad news".

1.9. Co-operation with upper management in planning a project

As shown above (see Appendix 2), the success of a project depends greatly on the support of the upper management of the institutions involved. Therefore, cooperation between the project manager and upper management is vitally important at all phases of a project. Studies have revealed that some general schemes/attitudes exist that are used relatively often by project managers and chief executive officers (CEO). The following is a list of some of them:

1. In order to ensure that the project is executed on time, the project manager tries to allocate more time for a project than it is ultimately necessary.
2. Since CEOs are aware of the attempts to increase the duration of projects in the planning phase, they reduce it, sometimes without any analysis or explanation.
3. A CEO has a personal opinion about the adequate duration for a project but he is not explicit in order to avoid responsibility. The project plan will not be approved until the timetable is close enough to one wanted by the CEO.
4. A CEO criticises a project plan without knowing the details. He hopes to achieve that a better solution will be found in the next version of the project plan.
5. A CEO has promised to deliver the software to a third party by a certain date; he insists on the work being completed that date to protect his reputation.
6. An institution is interested in getting a contract for the development certain software and makes an unrealistic offer/bid with a dumping price. This can have several unpleasant consequences like "political" agreements with the customer, low-quality software, overspending, replacement of the project manager (if a scapegoat needs to be found), etc.

These kind of actions are based mainly on political decisions and do not take into account the real possibilities. To what extent these political decisions are made depends primarily on the personal capabilities of the project manager.

There are also some indirect methods used by CEOs to check the quality of project planning and execution. One of these methods is called the "alcohol test". This method consists of asking different – unexpected – questions of the project manager and/or team members that allow a decision to be made about whether the project is realistic or runs smoothly enough. For example, these questions can include following:

- Who are the main customers for the project?
- What exactly are your responsibilities in the project?
- What are the most significant risks of the project?
- What are the most significant external factors that can influence the project?
- What size will the software have? How did you calculate that?
- What knowledge does the project team have about the area software is developed for?

To be protected from unpleasant situations the project manager and other team members should constantly ask these kinds of questions of themselves and of each other and try to find the answers.

1.10. Requirements development

According a The Standish Group study, user involvement is the most important success factor for software projects (see Appendix 2). The main objective of user involvement is to develop software that will satisfy the users' needs as much as possible. This is particularly important during the

determination and analysis of requirements for the software. During these phases, a market analysis should be made as well.

Usually customers need some support to understand formulate their real needs. There can be a number of reasons for that: customers and software developers use "different languages", customers are not experienced in software usage, they are not aware of the possibilities and limitations of computer software etc. Therefore, the requirements development can contain the following steps:

1. Selecting a group of reliable end users who can help to make decisions about the implementation of the users' activity patterns and user interface. The representatives from all main user groups should be involved starting with completely inexperienced users and ending with expert users.
2. Interviewing the users to determine the initial set of requirements. Usually users and software developers have different the understandings of software quality. It is recommended that joint seminars and workshops be organised, that is, the *Joint Application Design* methods be used.
3. Determining and modelling the activities/actions of users to develop a set of use cases.
4. Developing some (different) user interface prototypes. The prototypes should be relatively simple but provide an adequate understanding of working with the software. A prototype should be enhanced until the customers approve it. On the other hand, the customers should understand that this is just a prototype and therefore too much time should not be devoted to it. It is recommended to use some other tools for development of a prototype, different from that used for the development of the software (for example, drawing on paper).
5. Developing a style guide. The guide determines the visual elements of the software: font types and sizes, buttons and icons and their positions, style of messages, mnemonics of basic operations etc.
6. Preparing a user manual that is based on the user interface prototype. In practice, the user manuals are often prepared at the very end of a project. However, this can be risky because a) there is no time to get and implement the users' feedback and b) the structure and logic of the user manual does not harmonise with the activity patterns of the users (the users are interested that performing certain tasks – not a single operation – are optimised.
7. If some functions cannot be described by the user interface or user manual (for example, different algorithms, interoperability with other software/devices etc.), a separate document should be developed.

Requirements should be in written form and should be available to all stakeholders. The following shows what can happen if this is not done.

Example. A system for registering students to the courses.

The IT department of a university was asked to develop software that would enable the students to register for their courses online. Since the university's Study Information System contained all the necessary databases, the task seemed to be relatively easy and this was given to a novice developer without a list of requirements from the academic departments. After a couple of months, the system was opened for testing. Huge problems immediately emerged: 1), not all the schedules were available online (for example, the final lab schedules were not posted until after registration, since the usage of labs depended on the number of registered students) and the students had to check the schedules in the academic departments; 2) only users of the university computer network (LAN) could register online (authentication!), therefore the departments were forced to introduce a parallel registration (as not all students were users of the university LAN); 3) when registering for courses, no control was

performed as to whether all the prerequisite courses had been completed, etc. As a result, only 1.5% of the students used this service; some academic departments even recommended their students not use the system. The university administration decided that the whole system should be redeveloped from scratch. An additional analyst was involved and the project was repeated.

1.11. Quality management

The quality of software measures what level of requirements the software satisfies (needs of the customers). Quality is the most important indicator for software. The fact that the software was delivered a few weeks late will soon be forgotten, but dissatisfaction with the quality will last until the software is replaced. Quality control should be based on the following general requirements:

- the quality control activities are planned,
- these activities are conducted throughout the entire lifetime of the project (starting at the beginning of a project);
- quality assurance forms a separate task with clearly stated responsibilities;
- person(s) responsible for quality assurance have the relevant competences;
- quality assurance activities are sufficiently financed.

The basic instrument for quality assurance is the *Quality Assurance Plan*. One of the main topics of Quality Assurance Plan concerns the organisation of the testing. Testing is a structured activity that consists of different level of activities:

- ***Error check***. Errors are documented (the location and description; how critical the error is, who found and corrected, etc.) and announced. Special software can be used for the error check (see, for example, www.construx.com/survivalguide/);
- ***Examining the source code by an interactive checker***. This is usually the duty of a developer and is conducted before integration;
- ***Developer does integrity testing*** and its aim is to ensure the interoperability of the code with the earlier code. "Smoke tests" where the code will be integrated every day can be used as well; this provides an opportunity for the quick detection and solution of integration problems;
- ***Unit testing***. This is also usually a duty of the developer;
- ***Technical examination*** for controlling the quality of the technical results (user interface prototype, requirements specification, architecture, detailed design etc.). This is usually the task of a designated person (tester), or of a quality group, that follows a fixed procedure. For example: 1) the developer submits the necessary materials; 2) the tester(s) analyses the results; 3) the developer and tester(s) discuss the results; 4) A report is compiled (includes a list of errors and correction plans); and 5) correcting the errors. There are some general recommendations for technical examinations. 1) The aim is to detect the errors, not to propose solutions; 2) It is purely technical, customers and upper management will not be involved; 3) Double check that the errors really are corrected; 4) The results of the examination is made available to the project team; 5) Special time is reserved for error correction. It is important that developers take part in the technical examination as well because: 1) different people find different errors; 2) people are informed about each other work and are able to replace each other if needed (illness, business trip, vacation etc.). *For example, a module of a school information system was developed primarily by one person. When this person moved to another company nobody was able to replace him, the project was terminated and a new tender was announced*; 3) an unnecessary code can be detected. It was reported that the *Ariadne* missile exploded because some superfluous code from the previous version of

software was not removed; 4) conformity with standards can better be assured; 5) it can shorten the duration of a software project by 10-30%.

- ***System testing*** is used for testing completed components. Some general principles: 1) Users or customers should be heavily involved, 2) Testing should cover 100% of the features (functions, use cases. etc.), 3) There should be sufficient resources available. The more critical the software the more testers should be involved.

The quality assurance plan describes general procedures of quality assurance as well. For example, the general scheme from code creation until integration can be described as follows: 1) the developer composes a new code; 2) the developer tests the code; 3) the developer integrates the new code into its private copy of the software; 4) the developer submits the code to technical examination; 5) software is tested; 6) developer corrects the errors; 7) improved software is checked; and 8) the code is integrated into the master copy.

One should notice that testing allows the quality level of software to be assessed but is not a quality assurance tool (like weighting does not reduce weight). As the 80:20-rule (80% errors are in 20% of the modules) applies it is important to determine the problematic modules as soon as possible. The process is assessed to be effective if at least 95% of errors are corrected in the phase they are discovered. Testing is usually part of a critical path because some testing is performed at the end of the project or modules.

The Quality Assurance Plan should also determine the indicators for deciding when software can be released (for example, “if all critical errors are corrected” or “average interval between determination of two error is at least 8 hours” etc.).

For quality assurance, it is important to ***assess the progress*** of the project. The following general principles are used for the assessment: 1) it is possible to assess the software projects adequately; 2) adequate assessment is time consuming, it should be properly planned and documented; 3) quantitative methods are necessary and possibly some assessment tools as well; 4) adequate assessment is based on analogy with previously completed projects; and 5) assessments should systematically be refined during the project.

One should assess the basic features that are indicated in the *Software Development Plan* (incl. the deadlines of stages/modules) and correct them if needed. All relevant aspects should be taken into account. For example, national holidays when correcting the schedule, the developers falling ill, in correction of errors, training the project team members, time for servicing the previous projects, implementing changes, cooperation with the clients, etc. There are guides and tools for assessing project progress (see, for example, http://www.construx.com/Estimation/). This type of software can be used for simulations as well: by changing certain conditions, we can assess changes in the budget and schedule caused by these changes.

The following general rule applies to software projects as well: if some stage requires more time than initially planned then being able to catch up during subsequent stages is very unlikely, most likely, the project will lag even further behind at the end. In this case, the project team members should not be forced to catch up; instead, the timetable should be revised.

Beta-testing. Complex software will often be given to a broader community (experts, journalists, customers etc.) for testing. The goal is to get the broadest and most comprehensive feedback possible (for example, about interoperability with all possible hardware and other software). However, Beta-testing is relatively ineffective because 1) many testers do not bother to report the mistakes and 2) concretising huge amounts of information of different quality is very time consuming. For beta-testing, it is more effective to hire representatives of user groups that will do the testing under the guidance of experts.

There are special quality assurance institutions (for example, https://www.isqi.org/en/index.html International *Software Quality Institute*) and websites (for example *www.cs.umu.se/~jubo/Projects/QMSE, Quality Management for Small Enterprises*). A survey of open source testing software can be found at *http://opensourcetesting.org/*.

Exercises

1. Compile a short (up to one page) overview of *International Software Quality Institute* (www.isqi.org).
2. What are the main principles of the *Software Quality Function Deployment* (SQFD) model?

1.12. Software general design and architecture

There are no commonly accepted definitions for the general design and architecture of software. As most software engineers treat these two concepts as almost synonyms – especially for smaller projects – we will not differentiate between them (although *general design* can be used as a verb) and will use the term *architecture* in this section only.

Software architecture determines the general structure of the software's subsystems and their interaction principles. For the sake of simplicity and conceptual integrity, the number of subsystems should not be very big. For example, the following subsystems can be included: 1) user interface, 2) functional subprograms, 3) storing data, 4) output, 5) user tools, 6) lower level tools (for example, memory management).

The architecture describes the functions and module division of each subsystem, as well as the information exchange between the subsystems. It is recommended that a determination be made about the components that will be adapted from already existing software (skilful usage of ready-made components can considerably save time and other resources), the components that will be purchased, and the components that will be developed from scratch.

During this phase, the answers to the following questions are found:

- With what software, protocols and data should the new software be able to interact?
- To what extent is the user interface related to the other components of the software?
- What is the structure of databases?
- How the data will be stored?
- What basic algorithms will be used?
- How are the concurrent processes and multiple-users network operations solved?
- What are the principles of data protection and security?
- What are the localisation possibilities to other languages and platforms?
- What are the error handling principles?

The document that describes the software architecture formulates the general principles of the architecture (for example, to what extent should the software be extended by adding functions) and outline how the architecture phase is related to other phases of the project. Here the 80:20 principle can be applied (development of architecture can be started when about 80% of work in requirement analysis is completed). The architecture greatly determines the size and duration of the project.

The *UML* (*Unified Modelling Language*) is currently the main tool for describing the architecture of software.

It is often useful to consider different views of the architecture that only cover some predetermined elements of the architecture; especially for complex software, the views make it easier to understand the logic of the software.

For example, Philippe Kruchten suggests using the "4+1" view model of software architecture (see http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/Pbk4p1.pdf):

- The design view (logical view) describes the basic structures of the software, functions and relations between them. Classes form the basic components and first of all the users need to determine the view.
- The process view describes the aspects of concurrency and timely dependency and synchronisation. The view is based on non-functional requirements (performance, integrity of software, fault tolerance, etc.). A process is defined as a whole performed unit with tasks as the basic components. The process view can be presented on different abstraction levels.
- The components view (development view) describes the structure of the software modules. The modules and subsystems form the components.
- The physical view describes the interaction of the software with the physical infrastructure (computers, networks, other devices). The components are comprised of different devices.

The views described above are related to each other, especially the design view to the process view and to the components view, as well as the process view to the physical view. An additional scenarios view is also defined (why the model is called the 4+1 model); scenarios can be considered as certain use cases. The use case describes an activity pattern for performing a certain task that is described in the user terms/language.

Example *(P.Kruchten). The following scenario is composed for the development of a dial-up software: 1)the controller of a phone detects when the receiver has been raised from the telephone apparatus and sends an activation signal to the terminal; 2)the terminal reserves the necessary resources and transmits a signal for sending a dial-up tone to the controller; 3) the controller detects the code inserted by the user and transmits it to the terminal; 4) the terminal analyses the code; 5) if the code is valid, the terminal opens connection to the user.*

1.13. Release of a software

In order to make the decision to release the software, different techniques are used; we describe three of them below.

1) ***Counting of errors***. The errors are divided according to the gravity into three groups: *critical, significant, cosmetic*. The simplest method is to decide on the density of errors (that is measured by the average number of errors per 1,000 lines of code). If, for example, the density was 7-9 in previous projects and 250 errors have been detected in the new software with 50,000 lines of code, the software is most probably not ready to be released. Estimating the average time spent on correcting an error can provide an estimation of the total duration of error correction.
2) ***Predicting the number of errors using two test groups***. If the two groups detected M and N errors respectively, of which L were detected by the both groups then the total number of errors is estimated to be N*M/L. Using two test groups is relatively costly; this should be used mainly for the development of critical software where the number of errors should be minimised.

3) ***The Probability Method***. A number of errors will be generated and the total number of errors will be estimated based on the number of detected errors: if M errors were generated and N errors were afterwards detected by a group of testers (who are not aware of the generated errors) and L were from the set of generated errors, the total number of errors is estimated to be N*M/L. The generated errors should in some sense be representative (the generated errors should subsequently be corrected!).

The decision to release the software should be based on more than one indicator. A checklist of necessary activities is often used for that purpose. The checklist can consist up of to a few hundred positions. For general-purpose software, it could consist, for example, of the following (the responsible person indicated in parentheses):

- Update the version information (developer)
- Remove the information necessary for testing from the code (developer)
- Remove the generated errors (developer)
- Check that all the registered errors are removed (tester)
- Install the program from a CD (tester)
- Install the program from the Internet (tester)
- Install the program from a CD into a computer where an earlier version has been installed (tester)
- Check to make sure the installation program creates the correct Windows registers (tester)
- Uninstall the program (tester)
- Fix the list of distribution files (release group)
- Synchronise the date and time of all release files (release group)
- Burn the final program CD (release group)
- Check to make sure all the program files are present on the CD (release group)
- Perform the virus check (release group)
- Perform the check of bad sectors (release group)
- Create a spare copy and apply the change management scheme (release group)
- Check the version of readme.txt file on the CD (documents group)
- Check the version of help files on the CD (documents group)
- Check the copyright, license and other legal materials (project manager).

All the key persons on the project team should sign a protocol certifying the readiness of software.

The ***project's history*** is based on the project logs and basic data and it contains both quantitative and qualitative information (see, for example, Appendix 1). The opinions of the project team members can be collected by specially designed questionnaires where certain aspects are assessed, for example, on the Likert scale. The history document should be discussed at the general meeting of the project team with the aim of gaining the maximum benefit for subsequent projects.

1.14. Cost models of software development

For the further development of the software process models, it is important to know the major cost factors of the software. A general relationship of software costs can be expressed by the formula (in the order of decreasing influence on the costs):

$$\text{Amount of work} = (Size^{Process})(Personnel)(Environment)(Quality), \text{ where}$$

- *Size*: is expressed by the number of code lines or number of functions,

- *Process*: the ability to avoid non-productive activities (redesign, bureaucracy etc.) and optimise supporting activities (process monitoring, risk analysis, financial analysis, quality control, testing, continuing education, management etc.),
- *Personnel*: competency of developers, incl. experience in performing similar projects,
- *Environment*: usage of development tools and technologies,
- *Quality*: quality of the software (performance, adaptability etc.).

As *Process* > 1, the bigger the code the more expensive an average unit (for example, the average cost is at least 1.5 times higher for a program that is 10 times larger). Therefore if the development of a program with 25 000 lines of code requires 20 man-months 140 man-months will be needed for the development of a program with 75,000 lines of code (the relative expenses are more that two times higher).

The adequate application of this formula is relatively complicated because its elements are dependent on each other: for example, the usage of new development tools and technologies may cause a reduction in the size of code as well. The formula takes into account the code created by developers only and cannot be applied if readymade components and automatic code generation is used.

A large number of different cost models and tools has been developed including COCOMO and COCOMO II, CHECKPOINT, ESTIMACS, knowledgePlan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, SPQR/20. Some websites devoted to cost models have been developed as well (see, for example *www.jsc.nasa.gov/bu2*).

Which model to choose depends on a number of aspects:

- Although measuring the number of lines of code is easy, in many cases it is not enough (especially when using object-oriented programming); in many cases, using the number of functions is preferred because this does not depend on the technologies used. There is even *The International Function Point User Group*. In general, it seems that the usage of a number of functions is more adequate in the initial phases of a project and the usage of lines of code in the final phases of a project. To convert between different models the average numbers for lines of code that is needed for programming one function in different programming languages has been found: assembler – 320, C – 128, C++ – 56, Java – 55, Visual Basic – 35 etc.
- Cost models can be used for risk analysis and project planning if the costs are given (changing parameters until the costs fall into a given area).
- The estimation of costs should be based on competent analysis of existing practices and therefore project manager and other key persons should be involved; solely outsiders cannot do it.
- The estimation of costs should be made sufficiently detailed do that the basic risks and success opportunities can be understood.

Based on the above formula, here is a list of the main opportunities for cost reduction:

For the reduction of size:

- Multiple usages of design elements, processes, development tools and software components. If the number of usages is up to ten times then the costs are in logarithmic dependency (for example, double usage increases the costs on average by 50%, five usages by 125%).
- Usage of object-oriented technology (incl. *UML*) makes it easier to visualise the software and to concentrate on important aspects; this increases the ability to understand the problem and have the interested parties (incl. the end users) participate in the development process.
- Usage of automatic code generation (CASE tools, visual modelling tools, tools for development of graphical interfaces) and components.

- Usage of higher programming languages.

The application of these principles may reduce the number of lines of code 2 to 3 times; this in turn increases simplicity and decreases the probability of errors in the program.

Process is considered on three levels:

- Meta level (the organisation level) covers the long-term strategies of an organisation, the return on investments and organisation-wide regulations. Basic problems: bureaucracy and standardisation.
- Macro level (the project level) concerns the application of Meta level processes in the context of a project; covers policies in relation to the project, procedures and practices with the aim of producing a predetermined software product with planned resources. Basic problems: quality and optimisation of costs.
- Micro level (the project group level) covers policies, procedures and practices necessary for achieving midterm sub-goals. Basic problems: content and timetable.

The process can considerably reduce the total amount of development work. For example, in case of an n-step process, one can try to increase the efficiency of each step, but one can also try to eliminate some steps and run some steps simultaneously. The overall goal is to increase resources for productive activities and decrease the demand for supporting activities related to the resources needed for productive activities.

Personnel should cover the needs and be balanced (like a football team). The project manager has the central role: a perfectly managed project is usually successful even when executed by a modest project team, while poorly managed project usually fails even if the project team is of high quality. The project team members should mutually support each other, a team that consists of only ambitious persons is very risky in terms of cooperation. It is estimated that the average difference in productivity between beginners and experts is four times; although educating people is a role of an institution, proper training is sometimes needed in the framework of projects as well.

The following general principles usually apply to personnel:

- Use less and better people,
- Adapt the tasks according the competences and motivation of people (promoting an top designer to position of project manager may have catastrophic consequences),
- People should be motivated to achieve self-realisation,
- Choose people that are in harmony with and complement each other.

Consequently, the project manager should be able to:

- Hire the appropriate people,
- Avoid unfriendly relations between different parties,
- Make judgments,
- Develop team spirit, trust, common positions, motivate achievement etc.,
- “Sell” his/her decisions, priorities, positions etc. to the parties involved.

Environment covers tools for planning, the handling of requirements, visual modelling, programming, change management, configuration management, quality analysis, testing and development of the user interface. The overall goal is to reduce manual labour. The effective use of appropriate tools may reduce the total amount of work up to 40% and the use of a single tool usually by up to 5%.

Quality improvement is possible by several different ways, for example:

- Concentrating on the most important requirements already in early stages of a project;

- Using indicators and metrics that measure the progress and quality of a project,
- Using integrated development tools that cover the whole life cycle of the software, support change management, documentation and the automation of testing,
- Using visual modelling and higher level languages that support designing, programming, multiple usage of components and the adequate passing of information between the phases,
- Having early and continuous information on the level of software.

Workflow control can sometimes be used as well (especially of novice workers by experts); this can also be considered as training for the development of a work culture and for rooting out bad practices. Total control is very costly and therefore attention should be concentrated on critical aspects.

Different dynamic models are developed for detecting interdependence between the different factors. For example, overtime work may have a short-term effect but will be ineffective if used long term: labour productivity will decrease; the number of errors will increase, etc.

2 Software process management

2.1. Introduction to software process management

Research in software project life cycles and process models was intensified in 1990s because of the increased complexity and low success rate of software projects. As a result, a number of different models and development methods were concretised; proponents of agile software development founded "The Agile Alliance" (see www.agilealliance.org) in 2001.

In this chapter, we will describe some models and methods: the waterfall model, two-phase and multi-phase models, Rational Unified Process (*RUP*) and extreme programming (*XP*). As we will see later, these methods have some similarities but have some significant differences as well.

Some general suggestions have already been formed for choice and implement of software development methods:

1. Competent usage of a method is more important than the method itself. Therefore, implementation of a new method should be justified and could be performed after a competent analysis only.
2. Experience of implementing a new methodology obtained in other institutions may prevent dramatic failures.
3. A method should be adapted to the organisational culture as well as to skills and habits of the project team; a majority of the team should accept the (adapted) method.
4. The development and other tools depend sometimes from software process method. It is recommended to use three level divisions in usage the tools: compulsory, recommended, and acceptable.

The tools used in software projects should in addition to processing methods take into account some other factors as well. For example, Edward Yourdon [Yourdon, 1997] said that the following tools should be compulsory or recommended in running complex software projects:

- Electronic mail and groupware

- Tools for prototyping
- Tools for configuration management
- Tools for testing and debugging
- Tools for project management and assessment
- Repositories of reusable components
- CASE tools.

2.2. The waterfall model

The waterfall model (sometimes the term c*ascade model* is used) consists of a chain of consecutive activities that usually are the following (cf section 6.3, Structure of software process):

- Determination of requirements
- Analysis of requirements
- General design
- Detailed design
- Coding
- Testing
- Implementation.

Depending on the type of software, some more types or configuration of activities can be considered. For example, the *integration* of software can be considered separately (between *coding* and *testing*); just *design* can be used instead of *general design* and *detailed design,* or the term *architecture* can be used instead of general design, etc.

Comment. The name "waterfall model" comes from the fact that if the activities are presented as boxes on a xy-coordinate plane where the x-axis represents time and the y-axis represents the cost-axis (the costs of an error if made during this particular period and corrected at the end of the project) then a waterfall shape will be obtained.

The waterfall model has a major deficiency, most of the errors will be detected during testing and the correction may have high costs. Other deficiencies include: 1) software developed is requirements cantered (often change during the project); 2) the progress and risks of the project are difficult to trace and are mainly based on reports.

There are different methods for mitigating the risks, for example, the following:

- Executing a preliminary design of the software between the determination and analysis of requirements (planning memory management and data flows, interfaces, input/output etc.);
- Documenting the software design (documenting supports cooperation and later modification of software);
- Delivering the second version to the customers only (the first version will be used for complex tests and controlling of hypotheses);
- Planning and controlling the testing using specialised testers (testing requires relatively large amount of resources);
- Involving the customers in the development of software from the very beginning.

Barry Boehm has offered the basic economic characteristics of the waterfall model. These can be briefly formulated as follows:

1. Correction of an error that was made during design at the end of the project is a hundred times more expensive than correcting it immediately.
2. The duration of software development can not be shortened by more than 25% compared to the nominal duration: hiring more developers complicates the coordination of their work (for example, if 10 developers need 10 months to complete a project, then 20 developers need at least 7.5 months).
3. For every dollar spent for the development of software two dollars is needed for its management.
4. Development and management costs depend primarily on the number of lines of code.
5. The competence of developers is the main productivity factor in software development.
6. Between 1955 and 1985, the ratio between software and hardware has been changed from 15:85 to 85:15.
7. Programming consumes only 15% of software development costs.
8. One code line for software systems costs three times more than for single programs.
9. Only 60% of errors can be detected by reading the code.
10. *Pareto rule* is applicable in software development as well: 80% of problems are caused by 20% of the elements (80% of the activities to 20% of the requirements; 80% of costs to 20% of the components; 80% of errors in 20% of the components; 80% of the results achieved by 20% of the developers, etc.).

Exercises

1. What are the main principles of the Boehm spiral model?
2. What are the main principles of Microsoft Solutions Framework (MSF) Process Model?

2.3. Two-phase model

According the two-phase model, a software project is executed in two phases, the first focuses on planning and the second on programming and subsequent activities. The first phase covers 10-20% of the total project and consists mainly of analytical work. In fact, a two-phase model can be considered as a model of two consecutive projects.

The two-phase model has a number of positive aspects:
1. it provides an opportunity to interrupt the project if it turns out unreasonable to complete after relatively few expenditures have been made;
2. enables a project to be planned more adequately (because activities and costs are planned in two phases);
3. motivates the project managers to pay more attention to project planning.

During the first phase, the project plan is compiled. The plan contains the following:
- The vision and objectives of the project,
- Determination of the project team,
- Change management scheme,
- Requirements,

- Software architecture,
- Prototype of a user interface,
- User manual,
- Quality assurance plan,
- Software development plan incl. estimations of work, timetable and costs,
- Requirements/standards for design and code composition,
- Risk analysis.

The second phase is planned based on these documents (or it can be decided to interrupt the project if the costs or risks are too high).
Adequate planning is the main instrument for mitigating the risks. During the first phase, the model for the software development process should be determined as well.

Following common standards in code generation, it is much easier to understand each other work. For example, these common standards may cover the following aspects:

- Visual design of different parts (modules, subprograms, classes) of the program,
- Commenting,
- Marking of code that need further elaboration;
- Names of variables and functions,
- The maximum number of lines of code (subprograms) in a subprogram (correspondingly in a cluster),
- Agreements on technical questions (max. number of inserted cycles, usage of *GOTO* command etc.),
- Agreements on tools and repositories/stacks.

The larger the development team and the bigger the project, the more important is to agree on common standards.

It is useful to divide the **second phase** into stages. Each stage can be considered as a small project (see the next section).

2.4. Multi-stage development

According to the multi-stage development model, the second phase of the software development is divided into several stages. During each stage, certain functions are realised, so that the software is functional at the end of each stage. The distribution of the functions between the stages depends first on the needs and priorities of the customer: the most important functions are realised first, leaving the realisation of less important functions to the later stages. Therefore, each stage can be considered a mini-project, with most of the attributes of the full-featured project (including archiving the stage-related materials, the stage log etc. at the end of the stage).

The general timing structure for the multi-stage model is the following:

- The first phase: requirements, architecture/general design, initial list of stages.
- Stage 1: detailed design, coding, testing, integration and release
- Stage 2: detailed design, coding, testing, integration and release
- …
- Stage n: detailed design, coding, testing, integration and release
- Final release of software

Some stages can partly overlap. Nevertheless, it is recommended the 80:20 rule be followed: at least 80% of the stage should be completed before the next one is started.

The multi-stage model has a number of advantages in relation to the classical ADDIE model, including the following:

1. Early introduction of software. As the most important functions are realised first, the basic needs of the customers can be satisfied already with early releases of the software.
2. Risks are decreased. Integration of the software takes place step by step, problems are revealed almost immediately and can be corrected before having influenced other modules of the software.
3. Problems appear in the early phase. Multi-stage development prevents the situation where "90% of the project is completed, but nothing works"
4. The time for compiling reports decreases: working software approved by customers is better than any written report.
5. More possibilities/choices for increasing functionality. The stages are dynamically planned and the software can be released at the end of every stage.
6. Planning is more adequate. The feedback obtained during a stage can be taken into account in planning subsequent stages.
7. Better flexibility and efficiency of software process. Changes are discussed and realised at the end of each stage.
8. Correction of errors is more effective. Most of the errors are made during the last stage, and therefore their localisation is less time consuming.
9. Work distribution is more even throughout the project life cycle. For example, testers can start working at the end of the first stage.

Multi-stage development is used in a number of different software development methodologies including agile ones.

2.5. Agile software development methodologies

Although the principles of multi-stage software development started to spread already in 1950s, their systematic and massive use began during the second half of 1990s. A number of methodologies were developed, including *Extreme Programming* (XP) and *Scrum*.

The main principles of agile development were formulated in the *Manifesto for Agile Software Development* that was published in 2001. The principles were the following:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximising the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organising teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Extreme programming was created in 1996 by Kent Beck. It is based on five basic practices: Communication (between the customers and the fellow developers), Simplicity (of design), Feedback (by testing software starting from the project beginning), Respect (for the unique contribution of every team member), and Courage (in changing requirements and technology). Extreme programming follows a number of rules, including (see http://www.extremeprogramming.org/rules.html):

1. It is based on user stories, and the realisation of a story is 2 weeks on average.
2. Changing roles and tasks in development ("Move People Around").
3. Every day starts with a stand-up meeting.
4. Intense and continuous collaboration with the customer.
5. Agreed rules, specifications, and standards.
6. Development of a unit begins with creating a test.
7. Pair programming.

There are four core roles in XP: *developer*, *customer*, *coach*, and *tracker*. However, additional roles can be defined, for example *consultant*, *tester*, and *big boss*.

Using XP presumes high discipline, and team member with top-quality and broad qualifications. The teams are small, requirements can be changed throughout the whole project, and human work is used effectively.

The *Scrum* methodology (the term "scrum" was comes from rugby, where it denotes restarting the game after a fault has occurred) generally follows similar principles and rules, but has slightly different focuses and principles. These are based on the requirement that software should be easily changed if necessary. The main reasons are that 1) the customers' needs can change, and 2) the problems cannot be fully understood or defined. Software development is divided into iterations called sprints, which last from one to four weeks. The teams are cross-functional and self-organising; decisions are made by the team and there is no team leader as such.

The core roles in Scrum are the following: Product Owner (represents the customers' needs), Development Team, and Scrum Master (removes the obstacles preventing the application of the Scrum processes).

2.6. Capability Maturity Model for Software SW-CMM

In 1993, the Carnegie Mellon University Software Engineering Institute published a model for assessing the software development maturity (http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=11955) of institutions, the key practices for each maturity level of the model (see http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=11965, Key Practices of the Capability Maturity Model Version 1.1), and later the *Maturity Questionnaire* for performing software process appraisals (http://resources.sei.cmu.edu/asset_files/SpecialReport/1994_003_001_16265.pdf).

The model, referred to as CMM (and later SW-CMM when other types of maturity models were developed) distinguishes five levels of maturity of software processes: initial, repeatable, defined, managed, and optimising. Version 1.1 of the Capability Maturity Model for Software gives a general characterisation of these levels as follows:

1) ***Initial***. The software process is characterised as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.
2) ***Repeatable***. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
3) ***Defined***. The software process for both management and engineering activities is documented, standardised, and integrated into a standard software process for the organisation. All projects use an approved, tailored version of the organisation's standard software process for developing and maintaining software.
4) ***Managed.*** Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
5) ***Optimising.*** Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

The distribution of software development institutions between the levels is extremely uneven: according one study it was estimated that about 70 percent were on level one, while only about one percent on level five.

CMM defines key process areas, as well their descriptions and goals.

For example, the key process areas for level two are:

- Software configuration management
- Software quality assurance
- Software subcontract management
- Software project tracking and oversight
- Software project planning
- Requirements management

As an example, the goals for software configuration management are listed next:

1. The activities for software configuration management are planned.
2. Selected software work products are identified, controlled, and available.
3. Changes to identified software work products are controlled.
4. Affected groups and individuals are informed of the status and content of software baselines.

The Maturity Questionnaire contains questions concerning each Key Process Area. For example, the questions for *Software Project Planning* (level two) are:

1. Are estimates (e.g., size, cost, and schedule) documented for use in planning and tracking the software project?
2. Do the software plans document the activities to be performed and the commitments made for the software project?
3. Do all affected groups and individuals agree to their commitments related to the software project?
4. Does the project follow a written organisational policy for planning a software project?
5. Are adequate resources provided for planning the software project?
6. Are measurements used to determine the status of the activities for planning the software project (e.g., completion of milestones)?
7. Does the project manager review the activities for planning the software project on both a periodic and event-driven basis?

In 2003, Gunnar Piho assessed 69 software development institutions. It turned out that none of them fully satisfied the conditions for the second maturity level. Consequently, he proposed that three sub-levels of level two be defined:

SW-CMM-2-requirements (requirements are well managed).

SW-CMM-2- plans (SW-CMM-2-requirements + activities are planned and adequately resourced).

SW-CMM-2-results (CMM2-plans + monitoring activities/results and quality assurance).

The main shortcoming of this model is considered the fact that the CMM appraisal is mainly based different documents. Therefore, it is not suitable for assessing the maturity of institutions using agile processes. Nevertheless, the analyses show that the institutions with higher maturity levels are more successful. For example, the productivity of institutions on maturity level three is on average at least 25 percent higher than of those on level one. Some institutions have calculated the costs of achieving the next maturity level. For example, it took two years and cost $450,000 to bring an institution from level two to the level three. On the other hand, the yearly savings thereafter were estimated to be about two million US dollars.

2.7. Capability Maturity Model Integration CMMI

CMMI is intended to develop and improve processes that meet the business goals of an institution. It is a generalised version of CMM that includes other activity/interest areas, such as 1) product and service development (CMMI-DEV, http://resources.sei.cmu.edu/asset_files/TechnicalReport/2010_005_001_15287.pdf), service provision (CMMI-SVC, http://resources.sei.cmu.edu/asset_files/TechnicalReport/2010_005_001_15290.pdf), and product and service acquisition (CMMI-ACQ, http://resources.sei.cmu.edu/asset_files/TechnicalReport/2010_005_001_15284.pdf).

CMMI provides guidance for improving an organisation's **processes.** The process areas vary, depending on the areas of interest. However, 16 core process areas are present for all areas of interest:

1. Project Planning
2. Requirements Management
3. Quantitative Project Management
4. Risk Management
5. Integrated Project Management
6. Project Monitoring and Control
7. Organisational Process Definition
8. Organisational Process Focus
9. Organisational Performance Management
10. Organisational Process Performance
11. Organisational Training
12. Causal Analysis and Resolution
13. Configuration Management
14. Decision Analysis and Resolution
15. Measurement and Analysis
16. Process and Product Quality Assurance

Process areas are divided into categories – *Project Management* (number 1-6 above), *Process Management* (7-11), and *Support* (12-16). In some areas of interest a fourth category is formed – *Engineering*.

For every area of interest, the maturity levels are defined. For example, Maturity Level 2 of CMMI-DEV considers the following process areas:

- Configuration Management (pages 137-147 of CMMI-DEV)
- Measurement and Analysis (175-190)
- Project Monitoring and Control (271-279)
- Project Planning (281-299)
- Process and Product Quality Assurance (301-306)
- Requirements Management (341-347)
- Supplier Agreement Management (363-372)

NB! Compared with the key process areas of CMM level 2 (see the previous section), the word "software" is skipped and one process area is added (Measurement and Analysis).

Specific goals are defined for every process area. For example, the following specific practices are defined for *Integrated Project Management* in CMMI-DEV (Maturity Level 3):

1. Establish and maintain the project's defined process from project start-up through the life of the project.
2. Use organisational process assets and the measurement repository for estimating and planning project activities.
3. Establish and maintain the project's work environment based on the organisation's work environment standards.
4. Integrate the project plan and other plans that affect the project to describe the project's defined process.
5. Manage the project using the project plan, other plans that affect the project, and the project's defined process.

6. Establish and maintain teams.
7. Contribute process related experiences to organizational process assets.
8. Manage the involvement of relevant stakeholders in the project.
9. Participate with relevant stakeholders to identify, negotiate, and track critical dependencies.
10. Resolve issues with relevant stakeholders.

Beside the maturity levels that offers staged representation of the ***maturity levels***, CMMI also considers the continuous representation of ***capability levels***. There are four capability levels:

Level 0 (*incomplete*): a process that is either not performed or partially performed. One or more of the specific goals of the process area are not satisfied.

Level 1 (*performed*): a process that satisfies the specific goals of the process area.

Level 2 (*managed*): a process that is planned and executed in accordance with policy, employs skilled people having adequate resources to produce controlled outputs, involves relevant stakeholders; is monitored, controlled, reviewed, and evaluated.

Level 3 (*defined*): a process that is tailored from the organisation's set of standard processes according to the organisation's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organisational process assets.

The specific method was developed – *Standard CMMI Appraisal Method for Process Improvement A* (SCAMPI A, see http://resources.sei.cmu.edu/asset_files/Handbook/2006_002_001_14630.pdf). It provides benchmark-quality ratings relative to CMMI models and enables one to:

- gain insight into an organisation's capability by identifying the strengths and weaknesses of its current processes
- relate these strengths and weaknesses to the CMMI reference model(s)
- prioritise improvement plans
- focus on improvements (correct weaknesses that generate risks) that are most beneficial to the organisation given its current level of organisational maturity or process capabilities
- derive capability level ratings as well as a maturity level rating
- identify development/acquisition risks relative to capability/maturity determinations.

CMMI seems to be more beneficial for big institutions, and less beneficial for small ones that have fewer resources and use agile methods.

2.8. NASA Software Process Improvement methodology SPI

Software Engineering Laboratory of NASA/Goddard Space Flight Center, Software Engineering Operation at Computer Sciences Corporation, and the Institute for Advanced Computer Studies and Department of Computer Science at the University of Maryland jointly developed a three-phase software development model that was piloted on approximately 120 software projects from 1976 to 1993. Over this period, the error rate of the completed software has dropped by 75%; the cost of software has dropped by 50%; and the cycle time to produce equivalent software products has decreased by 40%.

This model follows an iterative and relativistic (based on the current level) paradigm, and consists of the following three phases (steps):

1. Understanding: Improve insight into the software process and its products by characterising

the production environment, including types of software developed, problems defined, process characteristics, and product characteristics. The objectives and possible processes, models, relations and indicators for achieving these objectives are specified. The goal is to understand what processes can lead to the objectives.

2. Assessing: Measure the impact of available technologies and process change on the products generated. Determine which technologies are beneficial and appropriate to the particular environment and, more importantly, how the technologies (or processes) must be refined in order to best match the process with the environment.
3. Packaging: After identifying process improvements, package the technology for application in the production organisation – the most suitable methods and tools are implemented in the organisation's everyday practice. This includes the development and enhancement of standards, training, and development policies.

The underlying principle was the reuse of software experiences to improve subsequent software tasks (see http://resources.sei.cmu.edu/asset_files/TechnicalReport/1994_005_001_16334.pdf). Moreover, the following rules were followed:

- Software measures will be flawed, inconsistent, and incomplete; the analysis must consider this. Do not place unfounded confidence in raw measurement data.
- Measurement activity must not be the dominant element of software process improvement; analysis is the goal (successful analysis of experiments should consume approximately three times the amount of effort that data collection activities require).
- Measurement information must be treated within a particular context; an analyst cannot compare data where the context is inconsistent or unknown.

The most significant differences between NASA SPI (below *SPI*) and SW-CMM (below *CMM*) are the following:

1. Concerning the conception: SPI – bottom-up (changes are defined by a local goal and prior experience instead of by a universal set of principles or practices), CMM – top-down;
2. Concerning measurement: SPI – relative with individual indicators (uses a detailed understanding of local process, products, characteristics, and goals to develop insight), CMM – absolute with universal indicators;
3. Concerning the scope: SPI – is based on concrete needs and experience, CMM – is based on general process quality indicators;
4. Concerning the scale of improvement: SPI – continuous, CMM – discrete (5 levels).

Note that several NASA SPI basic principles are realised in CMMI.

2.9. Software process assessment methodology SPICE

Above we have briefly described the *Standard CMMI Appraisal Method for Process Improvement A* (SCAMPI A). This method is applicable to a wider area of processes. Specifically for the assessment software processes, and related business management functions are also developed: SPICE – **S**oftware **P**rocess **I**mprovement and **C**apability d**E**termination. SPICE is approved as the ISO/IEC 15504 standard „Information technology – Process assessment". The standard consists of nine parts with the following names, each comprising an independent standard,:

Part 1: Concepts and vocabulary (ISO/IEC 15504-1:2004)

Part 2: Performing an assessment (ISO/IEC 15504-2:2003)

Part 3: Guidance on performing an assessment (ISO/IEC 15504-3:2004)

Part 4: Guidance on use for process improvement and process capability determination (ISO/IEC 15504-4:2004)

Part 5: An exemplar Process Assessment Model (ISO/IEC 15504-5:2012)

Part 6: An exemplar system life cycle process assessment model (ISO/IEC 15504-6:2008)

Part 7: Assessment of organisational maturity (ISO/IEC 15504-7:2008)

Part 8: An exemplary process assessment model for IT service management (ISO/IEC 15504-8:2012)

Part 9: Target process profiles (ISO/IEC 15504-9:2011)

The main use of the standard is reflected in its name – SPICE: 1) Process improvement and 2) Capability determination.

Part four of the standard specifies the requirements for improvement activities and provides guidance on planning and executing improvements, including a description of an improvement programme. This part can also be used for making supplier selection decisions. Moreover, this part, together with other parts of the standard, can also be used for self-assessment.

2.10. The principles of modern software development

The following is a list of the most important principles of modern software development as described by Walker Royce [Royce, 1998], the former vice-president of Rational Software Corporation and one of today's leading software development ideologists. These principles are also the basis for the definition of the ***iterative*** RUP software process:

1. ***The process must be based on architecture,*** i.e. planning of the fundamental requirements, architecture and life cycle plans must be done in a balanced way before the project resources are planned.
2. ***Use the iterative life cycle of the software*** in order to deal with the risks at a relatively early phase.
3. ***Design must be based on component technology,*** in order to reduce the code created by people (a component is defined as a coherent collection of code lines with a defined function and interfaces).
4. ***Establish a change management environment.*** Change management is especially important in iterative software development, in which one and the same elements work simultaneously in several work groups.
5. ***Use developmental tools that support spiral development,*** i.e. which automates and synchronises the information in various formats (definition of requirements, design models, initial code, supplementary code, test cases).
6. ***When designing, use graphic*** model-based depiction (e.g. UML).
7. ***Implement objective quality control and progress assessment.*** These must be integrated into the process.
8. ***Use demos for the assessment of interim results.*** Among other things, this enables the early discovery of design errors.

9. ***Plan interim outputs based on user scenarios,*** by constantly increasing the level of detail. This is the best way to determine the content of the iteration, assess the implementation and conduct the testing.
10. ***Establish a configurative process that is financial measurable.***

Graphically, the process based on the aforementioned principles can be depicted as a spiral, at the level where the first quarter is "Planning and Analysis", the second quarter is "Design", the third quarter is "Implementation" and the fourth quarter is "Assessment".

Based on this model, the initial version of the software is ready relatively quickly, whereas the emphasis is placed on the fields with the highest degree of risk; this is followed by the iterative development until the desired functionality, performance and reliability is achieved. The requirements are constantly supplemented and the architecture constantly developed. The implementation of these principles has an especially large impact on the cost of the software since the corresponding formula exponent ("*Process")* depends primarily on the following factors (the numbers of the principles that have the most impact on the factors are shown in the parentheses):

- Previous experience (2, 9)
- Flexibility of the process (4, 10)
- Solution of the errors in architecture (1, 3, 8)
- Team coherence (5, 6)
- Maturity of the software process (7).

As a comparison, here The Agile Alliance manifesto (www.agilemanifesto.org):
We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan
That is, while there is value in the items on the right, we value the items on the left more.

The above manifesto is based on the following principles (which are the basis for *XP*):

1. The greatest priority is satisfying the user with the early and constant output of software.
2. We accept changes in the requirements even in the late developmental phase. Agile processes implement changes for achieving the users' competitive advantages.
3. Release software at periods ranging from every two weeks to every two months with a preference for the shorter period.
4. The business people and developers must work together daily throughout the project.
5. Gather motivated people around the projects. Supply them with the necessary environment and tools and trust them to perform the work.
6. The most effective method for delivering information to the project team is face-to-face conversations.
7. Functioning software is the fundamental measure of progress.
8. Agile processes support sustainable development. Sponsors, developers and users must be capable of constant development.
9. Constant attention to technical excellence and good design increases efficiency.
10. Simplicity is important, i.e. the skill to maximize the work that does not need to be done.
11. Self-organising teams create the best architecture, requirements and design.

12. The team must regularly analyse the possibilities for increasing efficiency and adapt their activities accordingly.

2.11. Positive and negative experience with the current software development process

In connection with the production of software, processes related to software production that are based on the need to review the models and practices of software development have become intensified during the last few years: increased competition and the need to optimise costs based thereon, the expansion of the circle of potential clients/users, extremely rapidly developing technology, etc. Based on the above, attempts have been made to formulate principles that should be taken into account according to the current practices related to the software development process. Here a few examples (from a possible few hundred):

1. ***Introduce the software product to the users as early as possible.*** No matter how you try to learn about the user's needs during the phase when the requirements are formulated, the most effective means is to give the users the product and let them work with it.
2. ***Use an appropriate process model.*** A process must be chosen for every project that is most appropriate based on the organisational culture, risk readiness, clarity of the requirements, application area, etc.
3. ***Minimise intellectual distance*** (between the parties in the development process). The software structure must be as similar as possible to the structure of the real world; this is based on the utilisation of object-oriented techniques, component-technologies and visual modelling.
4. ***Before making the program faster, make it accurate.*** It is much easier to make a functioning program simpler than to make a fast program accurate, i.e. it is not worrying too much about optimising the program when initially coding.
5. ***Good management is more important than good technology.*** Bad management is not compensated by good technology and ample resources, but good management can compensate for bad technology and scarce resources. Not only do good managers attract good people, they also maximise their necessary competences and create good teams.
6. ***Understand and be guided by the user's priorities.*** A user will accept the fact that 90% of the functionality is delayed if only the most important 10% is ready on time.
7. ***Change-readiness of the design.*** The architecture, components and specification techniques must allow for changes, whereas the changes must not significantly increase complexity.
8. ***Most of that which is created in the course of software development must be self-documenting.*** Design without documentation is not design; they must be created simultaneously (we often hear people say, "The design is complete all that's left is the documentation", but this alludes to a weakness in the process).

For the handling of negative experiences, problematic IT projects have to be identified. Although it is probably not possible compile a complete list, some of the most frequent experiences can still be listed, for example:

1. The duration for completing the project exceeds the planned time by at least 50% (this is approximately twice as large as the average time overrun of successful projects).

2. The cost of the project exceeds the planned cost by at least 30% (this is somewhat greater than the cost overrun of the average IT project).
3. The developed software does not meet the client's expectations, and he raises the possibility of either amending or cancelling the contract.
4. The client does not implement the software to the necessary extent (does not utilise it, does not carry user training, does not establish the necessary regulations, does not implement the necessary work procedures, etc.).
5. The ordered software does not sufficiently support the business process (although formally the project may be successful).
6. The client and/or supplier (software developer) feel exploited by the other party or feels he/she has been unjustly/improperly treated.

In the article [Cole 1995], the projects carried out in 120 organisations have been analysed; the most significant reasons for problems were the following:

- The project goals are insufficiently defined (was mentioned in 51% of the cases)
- Bad planning and assessments (49%)
- Use of new technology or technology that the developers are insufficiently familiar with (42%)
- Problems in project management (42%)

In many cases, the problems were caused by the suppliers, inadequate communications, a weak project team and insufficient user training.

In the book [Smith 2001] based on his experiences and literature, the author has analysed the reasons for the problems that occurred in 40 IT projects (See Annex 9).

Exercises

1. Various models have been created for the implementation of software. What are the main features of the Technology Acceptance Model (TAM) and the Task-Technology Fit (TTF) model?
2. What are the more significant problems that have occurred in the fulfilment of IT projects carried out by the governmental sector of Great Britain (see www.parliament.the-stationery-office.co.uk/pa/cm/cmpubacc.htm)?

3 Project management support services

In this section, we examine the problems and activities that may not occur in the course of a specific project or preparations, but which is still related to the field of project management and more or less connected to its development.

3.1. Project portfolio management

The term *project portfolio* means a group of projects that are currently being updated or planned. Project portfolio management includes the methods for analysing and supporting the projects and their interaction, with the goal of ensure the optimal structure for the projects (incl. determining the priority of the individual projects and their mutual connections). The criteria for optimality and the indicators to be considered may differ, but as a rule, they are based on the organisation's strategic objectives.

Examples of optimality criteria include:

1. Improving the organisation's financial indicators as much as possible;
2. Achieving a competitive edge through the creation of new products and services;
3. Increasing the organisation's efficiency through the implementation of new technologies.

The indicators to be considered may include:

1. Costs;
2. Utilisation of resources;
3. Time schedule;
4. Investment schedule;
5. Revenue dynamic;
6. Risks.

Signs indicating a need for the PPM implementation:

- there are not enough resources to continue ongoing projects, and therefore the projects are late, competition for obtaining resources that inhibits the cooperation of various projects, etc.;
- key people being overworked and the risk of burnout;
- confusion in projects, incl. fulfilling projects that do not provide sufficient support for the achievement of the organisation's strategic objectives.

PPM is implemented in stages:

1. The necessity of the PPM implementation is ascertained.
2. An inventory of the ongoing projects is conducted: the main indicators and assessments of the projects.
3. The projects are prioritised, based on the specified objectives.

4. The projects are reorganised (resourced increased for some, resources decreased for some, some are suspended, some are stopped, and some are initiated).

A large number of methods and software solutions are being, and have been, created for PPM needs. All projects are assessed in association with other projects. Therefore, it can happen that the impact of a project that staying within its budget and is on schedule has less impact than expected on other projects and it is practical to finish the execution of the project and to redistribute the freed up resources among other projects.

If the goal of executing one project is to perform the project correctly (i.e. efficiently), the goal of a project portfolio is to perform the right projects.

When implementing PPM, the following questions should be answered:

1. Are we investing in the right things? When answering, one should analyse various business cases, the best practices and different possible solutions.
2. Are we optimising our resource utilisation? Essentially this means that the existing resources conform to resource needs. The main measures are shifting the project schedules, shifting the activities within a project, changing the scope of the project, suspending/stopping a project.
3. How well are we performing our tasks? There need to be concrete yardsticks, for the assessment the performance of the tasks based on which the assessments can be made. One yardstick should be the dynamic nature of the task performance, i.e. the ability to react quickly to changing needs.
4. Are we able to implement the right changes at the right time)? Not every idea deserves to be realised and not every good idea needs to be implemented immediately.
5. Are we achieving our desired objectives?

The successful implementation of PPM presumes good mutual level of information in order to guarantee the comprehensive and harmonious functioning of the organisation.

Comment. Project portfolio management is a specific field of application for the general management of portfolios or briefcases. Other examples of portfolios/briefcases include product portfolios, investment portfolios, application portfolios, etc. For example, application portfolio management in an organisation deals with the goals and needs of the (IT) applications used in an organisation. As a result, the IT maintenance costs may be reduced, the efficiency of the applications may be increased, better business models for the use of the applications may be found, etc.

3.2. Certification of the project managers

The best-known certification system for project managers has been developed by the Project Management Institute (PMI) (http://www.pmi.org/careerdevelopment/pages/certification-and-the-job-market.aspx). A special certification system for IT project managers was created by CompTIA (Computing Technology Industry Association).

Since PMI's certification system for project managers was the first in the world to receive ISO 9001 quality certification in 1999, then we take a closer look at it (for the manual, see http://www.pmi.org/certification/~/media/pdf/certifications/pdc_pmphandbook.ashx).

The following preconditions need to be filled in order to receive certification as a PMP Project Management Professional:

1. At least 4,500 hours of project management experience during the three years prior to application (for people without higher education, 7,500 hours within 5 years) and 35 contact hours of project management education; a survey of one's work must be submitted and the relevant form filled out.
2. The proper document must be signed, whereby the applicant undertakes to observe the procedures for applying for the PMI certificate.
3. The exam fee must be paid (was $405 for PMI members in 2012).
4. Successfully complete the online test. The test is comprised of 200 individually generated multiple-choice questions and there is 4.5 hours to complete the exam. At least 135 questions must be answered correctly.

The PMP exam is based on the matrix of process groups and fields of competency provided by the PMBOK Guide 2000 (for instance, questions are mainly divided into the following: Initiation, Planning, Performance, Control, Completion) which include questions about the concepts, and the solution of specific assignments (for example, finding the critical path), as well as problem-solving assignments.

For retaining your PMP certificate, the PMI has established certain requirements and created the relevant support programme (the corresponding manual also exists).

Assignments

1. Prepare a survey of the ICT certificates provided by ComTIA (www.comptia.org).

3.3. Standards and specifications

Standards are the normative documents established by a specific standardisation organisation, which establish the requirements for some process of its result. Normative documents that are not established as standards are called *specifications*. Standards and specification can be classified by geography (international, European, national) and by the field it regulated.

Project manages should be familiar with the following international standards:

1. ISO 9001 – 1994 is the "Model for Quality Assurance in Design, Development, Production, Installation and Servicing"),
2. ISO 9000-3 – 1991 is the "Quality Management and Quality Assurance Standards"); part 3 thereof establishes the requirements for applying the ISO 9001 standard to the development, supply and management of software.

The following software and system engineering standards should be mentioned (see JTC 1/SC 7 standards at www.iso.ch/ and those related specifically to system development at www.iso.org/iso/iso_catalogue.htm):

1. ISO/IEC 9126 –1991 "Software quality characteristics",
2. ISO/IEC 12207 – 1995 "Information technology – Software Life Cycle Process",
3. ISO/IEC 12119 – 1994 "Information technology – Software packages – quality requirements and testing".

4. ISO/IEC TR 15504 (Parts 1-9): 1998 “Information technology – Software process assessment”
5. IEEE standard 1058.1 – 1987 “Standard for Software Project Management Plans”,
6. ISO/IEC TR 13335 “Guidelines for the Management of IT Security”,
7. ISO/IEC TR 13569 “Financial services and related information security guidelines”
8. ISO / IEC 17799 and BS7799 “Information technology -- Security techniques -- Code of practice for information security management”.

The abbreviation IEC means that the standard has been prepared in cooperation with the International Electrotechnical Commission.

The following examples of national (local) standards should be mentioned.

1. United States: Information Technology Laboratory of the National Institute of Standards and Technology (www.itl.nist.gov) provides guidelines for IT research and develops IT test measures and standards,
2. Australia: National Competency Standards for Project Management (http://www.projects.uts.edu.au/resources/pdfs/ProjectManagementCompetencyStandardsver4.pdf),

Currently, the PMBOK can also be considered an international specification.

When executing projects, a whole series of other standards and agreement that regulate the field of activity must be observed. For instance, developers of electronic learning administrative systems must use the SCORM model (Sharable Content Object Reference Model, see www.adlnet.org), which is not an official standard, but the observance of which is considered to be almost self-evident. The standards that are translated into another language are specially designated, e.g. EVS-ISO/IEC 12207 indicates the Estonian version of ISO/IEC 12207.

The specifications for digital signature can be found online at http://searchsecurity.techtarget.com/definition/Digital-Signature-Standard. Many agreements that order the field of activity have not been formulated into standards or specification; these include various classification systems (e.g. www.acm.org/class).

The standards in very rapidly developing fields of activity are still in the process of being developed or are a matter of consensus, i.e. have the status of specifications (have not been officially confirmed. for example, the guidelines developed by the IMS Global Learning Consortium (www.imsglobal.org). Below, you will find short descriptions of the most important IT-related standards.

1. Standard ISO/IEC 12207 “Software life cycle processes”

The standard defines the terms used in the description of software life cycles, and deals with the following:

- 5 primary processes: acquisition, supply, development, operation and maintenance;
- 8 auxiliary processes: documentation, configuration management, quality assurance, auditing, verification, validation, joint review, problem solving;
- 4 organisational processes: management, infrastructure, improvement and training.

Acquisition is comprised of initiation, request for proposal preparation, concluding and updating the contract, supplier monitoring, acceptance and completion.

Supply is comprised of initiation, answer preparation, contract, planning, performance and management, review and assessment, delivery and completion.

Development is comprised of executing the process, defining the functional requirements, designing the system architecture, analysing the software requirements, designing the software architecture, the detailed designing of the software, programming and testing the software, integrating the software, testing the functional requirements of the software, integrating the system, testing the functional requirements of system, installing the software and supporting the acceptance of the software.
Operation is comprised of the execution of the process, test operation, system operation and user support.
Maintenance is comprised of the execution of the process, analysis of problems and changes, executing changes, review and approval of maintenance, migration, removal of software.
The other processes are similarly structured, for instance:
Quality assurance is comprised of the execution of the process, guaranteeing the products and guaranteeing the processes.

2. Standard 15504 "Software process assessment"

IT standard ISO 15504 is a framework for assessing software processes (acquisition, development, support etc.). The basis for the standard is the SPICE method for software process assessment (see chapter 8). Similarly to the SW-CMM requirements, here too a scale assesses capability or incapability. Standards ISO 12207 and 15504 are coordinated on (see, for instance, the comparison at http://csqa.info/iso/iec_12207_and_iso_15504).

Exercises

1. Formulate the main concept of the ISO 10006 standard for quality management in projects.
2. Formulate the scope of project management standard ISO 21500.
3. Become thoroughly familiarised with one of the guidelines developed by the IMS Global Learning Consortium (e.g. the IMS e-Portfolio Best Practice and Implementation Guide, http://www.imsglobal.org/ep/epv1p0/imsep_bestv1p0.html).
4. What are the main assignments of the Information Society Standardisation System of the European Committee for Standardization (CEN; Comité Européen de Normalisation), the sub-system for the standardisation of an information society?
5. Become familiarised with the survey of the standards related to system development presented in "A Review of Systems Engineering Standards and Processes" (G.-S.Chang, H.-L. Perng, J.-N. Juang, *Journal of Biomechatronics Engineering 1*, Nr.1 (2008), 71-85).

3.4. Software development theory and other leading institutions

During the last decade, several institutions have developed whose research and developments have a greatly shaped software development strategies and methods. Below is a short list thereof.

1. Software Engineering Institute at Carnegie Mellon University

The SEI has developed a whole series of methods for the assessment of the quality of the structures related to software development. The CMM-SW (Capability Maturity Model for Software) is best known and is implemented worldwide. The following documents (see www.sei.cmu.edu/publications) should also be mentioned:

- The Software Process Maturity Questionnaire (document CMU/SEI-94-SR-7), which makes it possible to make a self-assessment of the maturity of a software development project by answering certain questions (totalling 41 pages!). (For the final assessment, it an on-site inspection by a specialised company is necessary). The questions are comparatively general (for example, "Is the quality of the workers' in-service training assessed?") and the answers are "Yes", "No", "Not relevant" and "Don't know".
- The goal of the People Capability Maturity Model (document CMU/SEI-95-MM-002) is to help organisations recruit, develop, motivate, organise and hand onto good people, in order to increase the efficiency of the entire organisation.
- The People CMM-Based Assessment Method Description (Version 1.0, document CMU/SEI-98-TR-012 and ESC-TR-98-012) provides an extremely detailed description of how to carry out People CMM-based accreditation in an institution. Generally, this takes 5 months and sometimes the activities related to accreditation are planned down to the hour!
- A Systems Engineering Capability Maturity Model (Version 1.1, document CMU/SEI-95-MM-003).
- A Description of the Systems Engineering Capability Maturity Model Appraisal Method, (Version 1.1, CMU/SEI-96-HB-004).
- CMM-Based Appraisal for Internal Process Improvement (CBA IPI): Method Description (CMU/SEI-96-TR-007).
- Software Acquisition Capability Maturity Model (CMU/SEI-2002-TR-010, see http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr010.pdf) deals with how to buy/order software and software services. SA-CMM identifies five levels in a company's ability to order and administer software, starting with the company does not know what it wants and ending with software being ordered that maximally supports the institution's fundamental processes, and the quality of the utilisation being monitored and optimised.
- Software Acquisition Process Maturity Questionnaire (CMU/SEI-97-SR-013), questionnaire for the self-analysis of the SA-CMM level.
- Software Acquisition Risk Management Key Process Area (KPA) – A Guidebook, Version 1.0 (CMU/SEI-97-HB-002) deals with how to implement risk management in software development.
- CMM Appraisal Framework (ver. 1.0, CMU/SEI-95-TR-001).
- Software Capability Evaluation. Implementation Guide for Supplier Selection (ver. 3.0, CMU/SEI-95-TR-012). This is primarily directed at the public sector and deals with how to choose a software or service provider (among other things deals with the risks related to ordering and partly with assessing the quality of the suppliers' software processes).
- Capability Maturity Model Integration (CMMI). This is directed at increasing the maturity of organisations as a whole (see also www.sei.cmu.edu/cmmi).

2. NASA Software Engineering Laboratory

The Software Engineering Laboratory, (SEL) at NASA's Goddard Space Flight Center was the first institution to win the IEEE prize in 1994 for achievements in the area of software processes. If software with 100,000 lines of code contains an average of 850 errors when it is completed, the ones developed at NASA SEL have only 50 errors. Approximately 10% of NASA's personnel dealt with software development (in the mid-1990s 8,400 people).

Although the activities at the lab are currently halted, a whole series of documents were developed there which still maintain their topicality, such as:

- Manager's Handbook for Software Development, Revision 1, Document SEL-84-101, 1990.
- Relationships, Models, and Management Rules, Document SEL-91-001, 1991.
- Recommended Approach to Software Development, Revision 3, Document SEL-81-305, 1992.
- Software Process Improvement Guidebook, SEL-95-102.
- Software Measurement Guidebook, Revision 1, Document SEL-94-102, 1995.

Based on its long-term experience, the SEL formulated eight elements for successful software projects:

1. Compile and adhere to the software development plan, concretising it when necessary;
2. Strengthen your personnel by creating a productive environment, by establishing clear obligations and rights for their fulfilment;
3. Minimise bureaucracy; there must be a definite purpose for every meeting and discussion; ***NB****! The best report is functioning software*.
4. Define the basic requirements and manage their changes. Requirements should be stabilised as early as possible; draw up a list of potential changing and undefined requirements and calculate their impact on the schedule and costs. Relevant questions should be resolved in the architecture phase or in the detailed design phase at the latest;
5. Regularly compare the project's progress to the initial plans and to similar projects that were executed recently. If the need for significant changes becomes apparent, re-plan the project, if possible with a reduction in the future workload. Try not to be overly optimistic;
6. Regularly reassess the project's size, workload and schedule; you must not stick to you initial assessment too rigidly;
7. Define and manage the phase transition, while not losing too much time. The preparatory work for the next phase (stage) must start a few weeks before the current phase is completed;
8. Stimulate team spirit, by emphasising a common vision. Constantly inform the project team of the status, risks and other parameters of the project. The project should start with a relatively small team who works out the vision and concept.

Also, identify the eight elements that should be missing from a successful project:

1. No one should be allowed to work unsystematically. Although software development is a creative process, it must be based on the rational application of certain principles, experiences and techniques.
2. Unrealistic goals should not be established. If the team does not believe in the goals, work efficiency will decline rapidly; rushing the team causes errors;
3. Do not implement changes before assessing their impact. Even constantly introducing small changes can turn out to be very costly in the long run;
4. Only implement what is necessary; so not increase the project's complexity;
5. So not overdo it with executors, only add additional employees if they are guaranteed to have work;
6. Do not assume that a lag in the schedule can be made up later.
7. Do not lower requirements in order to save on costs and time. This can reduce motivation and give bad signs to the users/clients;
8. Too much documentation does not ensure success; it must totally be justified.

3. Rational Software Corporation

Rational (www.rational.com) was perhaps the main innovator in software development in the 1990s. The general software development method called RUP (Rational Unified Process), the modelling language called UML (*Unified Modelling Language*) that supports it and the Rational Rose integrated development tool were developed. In addition, a whole series of corresponding informational and study materials were developed, such as the following white papers:

1. Rational Unified Process for Systems Engineering RUP SE 1.0
2. Applying Requirement Management with Use Cases, Rational Software White Paper, TP505.
3. Building Web Solutions with the Rational Unified Process: Unifying the Creative Design Process and the Software Engineering Process, A Rational Software & Context Integration white paper,
4. A Comparison of RUP and XP, Rational Software White Paper,
5. Rational Unified Process: Best Practices for Software Development Teams,
6. Reaching CMM Levels 2 and 3 with the Rational Unified Process,
7. Software Process Engineering Management: The Unified Process Model (UPM),
8. Unifying Enterprise Development Teams with the UML.

In December 2002, Rational stocks were acquired by IBM.

4. Project Management Institute

The PMI (www.pmi.org) was founded in 1969 and it is becoming the leading institution in the field of project management. In addition to developing the Project Management Body of Knowledge (PMBOK) and the PMP Exam, the certification system for project managers (for instance, 12,100 project managers were certified in 2002), the PMI has also created the following:

- PMI code of ethics for the project managers that have completed the training (Code of Ethics and Professional Conduct, http://www.pmi.org/en/About-Us/Ethics/~/media/PDF/Ethics/ap_pmicodeofethics.ashx*)*.
    1. The profession development standard for project managers called Project Management Competency Development Framework.
    2. In 2002, the Certificate of Added Qualification was instituted for the certification of IT project managers.
    3. The organisational maturity model for project management called OPM3 – Organisational Project Management Maturity Model (http://eng.mft.info/UploadedFiles/gFiles/cd9a1c903438439.pdf). This web address also provides a survey of several other models.
    4. Virtual bookstore for project management materials (http://marketplace.pmi.org/Pages/default.aspx?Category=ProjectManagement).

The number of important institutions is much larger, and a whole series of resource-related links, portals, etc.

5. Project management information sources

A series of periodicals appear in the field of project management, for example:

- *The International Journal of Project Management* (www.elsevier.com/locate/issn/02637863).
- *Project Management Journal* (http://www.pmi.org/Knowledge-Center/Publications-Project-Management-Journal.aspx); published by the Project Management Institute.

3.5. Project cost-effectiveness assessments

When planning projects it is important that the project result be greater than the costs related to the execution of the project. Whereas, the investments made in the project must be more profitable than investing the same resources for some other purpose. Let's take a brief look at the following concepts related to the calculation of profitability: *analysis of net value, return on investment, analysis of the payback period,* and *weighted scoring model*.

***Net present value* (*NPV*)** is the total profit received from the project during a certain period at current prices. If the costs of executing the project during the years 1 … n are $k_1, k_2, \ldots, k_n$ respectively, and the return on the project is correspondingly $t_1, t_2, \ldots, t_n$, and if the annual *discount rate* is equal to r (is equal to the minimal required annual productivity of the investment), then *the net value of the project* is calculated using the following formula:

$$NPV = \sum_{i=1}^{n} \frac{t_i - k_i}{(1+r)^i}$$

The greater the NPV, the greater the profitability of the project. It is only sensible to start up projects for which *NPV>0*.

***Return on investment* (*ROI*)** is calculated using the following formula:

$$ROI = \frac{NPV}{\sum_{i=1}^{n} \frac{t_i}{(1+r)^i}}$$

Therefore, the investment profitability is 0 when the earned discounted resources are equal to the expended discounted resources.

Payback period is the period during which the resources spent on the investment are earned back (considering the discounted value of the revenues and costs).

Weighted scoring model is used when several factors have to be used when deciding on a project. These factors may include qualitative indicators (the degree of support for the company's main objectives, the degree of support for the client/user, the degree of risk for the success of the project, etc.), to which certain ratings are assigned (for example from 0 to 100; the better the indicator, the higher the rating). Depending on the company's priorities, each of the indicators is assigned a weight coefficient: the more important the indicator, the greater the weight coefficient. Generally, the weight coefficients are chosen so that they total equals 1. If the ratings of a project's indicators are $h_1, \ldots, h_m$ and the weight coefficients are $c_1, \ldots, c_m$, then the calculation made using the following formula:

$$H = \sum_{i=1}^{m} h_i \cdot c_i$$

The greater the value of H, the more sensible the project.

Example. If the costs related to the project by year (in thousands of EEK) are 850, 320, 150 and 150 and the revenue received from the project is 0, 440, 850 and 950 corresponding, using a discount rate of 10%, the corresponding values per year would be:

Year	1	2	3	4
Discounted cost	772.73	264.46	112.70	102.45
Discounted revenue	0	363.64	638.62	648.86

We get:

NPV = (0 – 772.73) + (363.64 – 264.46) + (638.62 – 112.70) + (648.86 – 102.45) = 398.78

ROI = 398.78/(772.73 + 264.46 + 112.70 + 102.45) = 398.78/1252.34 = 31.84%

The payback period for the investment is a little over three years: during the first three years, the costs exceed the revenues by 147.63 thousand EEK; in the fourth year, the revenues exceed the costs by 546.41 thousand EEK.

Exercises

1. What goal does the amount $(1+r)^i$ serve in the net value calculation formula?
2. Why is it sensible to choose a weight coefficient so that it totals 1?
3. Compile a weighted scoring model for calculating the score of the following cars, when the indicators to be considered are 1) the inverse of the car's price (in thousands of EEK), multiplied by 10^4; 2) the length of the car in decimetres; 3) the company's prestige (MB – 100, Audi – 90, Toyota –80, Volvo – 70, Opel – 50, Ford – 40, Lada – 20); 4) the average occurrence of breakdowns during ten years (MB – 50, Audi – 60, Toyota – 20, Volvo – 60, Opel – 80, Ford – 80, Lada – 140). If the weight coefficient is 40, 30, 20, 10 respectively.

   Calculate the ratings for the following cars in the table:

Brand	Price	Length of the car
MB	440	48
Audi	400	47
Toyota	280	44
Volvo	360	44
Opel	260	44
Ford	300	47
Lada	100	41

   Which car gets the best rating?
4. Compile a weighted scoring model by identifying 3 to 5 indicators and assigning them weight coefficients. With the help of this model, calculate the rating for an actual project.
5. The profitability of an investment can be assessed in respect to a single aspect – for instance project management. Which method is used for this in the article "Calculating Project Management's Return on Investment"?

Literature

[Boehm 2000] Boehm, Barry, et al, *Software Cost Estimation with Cocomo II*, Prentice-Hall, 2000; ISBN 0-13-026692-2.

[McConnell, 1997] McConnell, Steve, *Software Project Survival Guide,* Microsoft Press, 1998; ISBN 1-57231-621-7.

[Royce 1998] Royce, Walker, *Software Project Management. A Unified Framework*, Addison Wesley, 1999, ISBN 0-201-30958-0.

[Smith 2001] Smith, John M., *Troubled IT Projects: Prevention and Turnaround*, The Institution of Electrical Engineers, 2001. ISBN 978-0852961049.

[Thomsett, 1990] Thomsett, Rob, "Effective Project Teams: A Dilemma, a Model, a Solution". *American Programmer*, July-August 1990.

[Yourdon, 2003] Yourdon, Edward, *Death March* (2nd Edition)., Prentice Hall, 2003, ISBN 978-0131436350.

Appendices

Appendix 1: Possible structure of a history document of a software project

Introduction

General information about the objective of software, target group etc.

Progress review

Description of (sub-) goals, main risks, timetable, personnel etc. for each phase of the project
Description of the phases:

- Determination of requirements and prototyping the user interface
- Planning of quality assurance
- Architecture
- Planning of stages
- Stages 1…n
- Delivery of software

Basic data of a project

Description of the organisational structure, team members and their roles as well the actual contribution.

Actual timetable and amount of work:

- Data on time counting
- The number of subsystems
- Multiplying the used lines of code
- Usage of media tools (voice, graphics, video etc.)
- Number of errors
- Proposed and accepted changes
- Comparison of actual timetable to one initially planned
- Comparison of actual amount of work to one initially planned (graphically)
- Graph expressing the weekly growth in the numbers of lines of code
- Graph expressing the weekly numbers of detected and corrected errors.

Gained experience

Planning: Were the plans useful? Have the plans been followed? Were the personnel sufficiently qualified? Were there enough personnel in each sector?

Requirements: Was the amount of requirements sufficient? Were the requirements stable enough or were they changed considerably? Were they understandable or inadequately interpreted?

Development: General description of design, coding and testing. How was the work organised? How was the integration organised? How did the deliverables work?

Testing: How was the planning of testing, development of test cases and smoke testing organised? Description of how the automatic testing was performed.

New technology: How did new technology influence the costs, timetable and quality? Did the managers and developers interpret this influence similarly?

Appendix 2: Success factors of IT-projects (by Chaos of The Standish Group)

The following (http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS) is a summary of IT project managers' assessments regarding the main factors influencing the success and failure of IT projects.

Top ten reasons for success:

1. User involvement
2. Executive management support
3. Clear business objectives
4. Optimising scope
5. Agile process
6. Project manager expertise
7. Financial management
8. Skilled resources
9. Formal methodology
10. Standard tools and infrastructure

Main reasons of failure of IT-projects:

1. Incomplete requirements
2. Insufficient user involvement
3. Lack of resources
4. Unrealistic expectations
5. Insufficient executive management support
6. Changes of requirements and specification
7. Insufficient planning
8. Needs changes
9. Deficiencies in IT-management
10. Insufficient possession of technologies

Appendix 3: the most significant reasons for problems in software projects (grouped by development phase)

The lists below are adapted from [Smith 2001, pp. 18-19].

Project conception

- Reliance on inadequate presumptions and business cases
- Unclear/insufficient definition of the goals, expected results and success criteria by the client
- Reliance on outdated or immature technologies
- A lack of dedication and/or competence on the part of the client
- The client's expectations related to financing and expenditure of time are unrealistic
- The inability of the client to divide complicated projects into phases or smaller projects

Project initiation/mobilisation

- The developer compiles an unrealistic budget and time schedule and overrates its capability
- The client's errors in the definition and documentation of requirements
- The lack of open, direct and equal cooperation between the client and developer
- The developer's superficiality in defining the scope of the project before the contract is concluded
- The lack of involvement of the end users on the part of the client
- Underestimation of the need for resources (primarily human resources) on the part of the developer
- The insufficient definition of the project jobs, results and responsibility procedures on the part of the developer
- Weak planning for risk management and dealing in the unexpected
- Weak project planning, management and performance
- Vagueness in the roles and responsibilities related to the contractual relations
- Performing a full-volume project at a fixed price based on the contract

System design

- Poor change management
- Wrong choice related to the technical platform and/or architecture
- The developer initiating a new phase before the previous phase is executed
- Wrong choice related to the design/development method
- Inability to carry out effective project review
- The developer's insufficient skills

- Insufficient observance of the standards related to the design, coding, testing, configuration management and other standards
- Lack of observance of the developer's requirements
- The client interferes in the design process

System development

- In the case of a delay, the technology being used may change
- Poor change management
- The developer's inadequate training and guidance of younger team members
- The inadequate reviews of design/code/documentation on the part of the developer
- Weak management of sub-contracts
- Formal integration and testing on the part of the developer
- Insufficient attention to non-functional requirements on the part of the developer

System implementation

- Insufficient change management on the part of the client
- Inadequate training/testing of the users/systems
- Unexpected crashing of the system without an effective recovery plan
- Lack of a date for the final commissioning

System operation, benefit delivery, stewardship and disposal

- The client's inability to measure the received revenue and to adjust the systems
- The client's inability to manage and improve the system
- Changes in the market and macroeconomic environment