CO557
Software Engineering

CW2

21611431

18th May 2018

Computing & Web Development
Buckinghamshire New University

Summary

Part A

Preamble

Section 1

Preamble

The theory and application for this coursework have been inspired by:

- Boehm (1988)
- Sommerville (2016)
- The lectures and the work done in class

Part B

Theory

Section 2

Risk Management

2.1 Overview

Risks can be seen as something we'd prefer not to have happen. They may . threaten the project, the software being developed or the organization. Risk management involved anticipating the risks that might affect either the projkect schedule or the quality of the software produced in order to take action to avoid them.

Risks can be categorized according to the type of risk. A complementary classification is to classify risks according to what these risks affect:

1. Project risks affect the schedule andor resources of the project
2. Product risks affect the quality andor proformance of the software produced
3. Business risks affect the organization developing or procuring the software.

Often ties, these risk categories can overlap. For example, losing an experienced engineer can present a project, product and business risk at the same time for the following reasons:

- A project risk because it takes time to get a new team member up to speed
- A product risk because a replacement may not be as experienced and make more errors
- A business risk because the engineer's reputation might have been critical to win contracts.

For large projects, the results of the risk analysis should be recorded in a risk register along with a consequence analysis. The consequence analysis sets out the consequences of the risks for the project, product and business. Effective risk management makes it easier to cope with problems and to ensure that these do not lead to unacceptable budget or schedule slippage. For smaller projects, formal risk recording may not be required, but the project manager should still be aware of the risks and their consequences.

The specific risks that may affect a project depen on the project and the organizational environment in which the software is being developed. However, there are common risks that are independent of the type of software that is being developed. These can occur in any software development project. Some examples of these are:

1. Staff turnover: Experienced staff will leave the project before it is finished
2. Management change: There will be a change of company management with different priorities
3. Hardware unavailability: Hardware that is essential for the project will not be delivered on schedule
4. Requirements change: There will be a larger number of changes to the requirements than anticipated
5. Specification delays: Specifications of essential interfaces are not available on schedule
6. Size underestimate: The size of the system had been underestimated
7. Software toll underperformance: Software tools that support the project do not perform as anticipated
8. Technology change: The underlying technology on which the system is uilt is superseded by new technology
9. Product competition: A competitive product is marketed before the system is completed

Software risk management is important because of the inherent uncertainties in software development. These uncertainties stem from loosely defined requirements, requirements changes due to changes in customer needs, difficulties in estimating the time and resources required for software development, and differences in individual skills. There is a need to anticipate risks, understand their impact on the project, the product and the business and take steps to avoid them. There may be a need to draw up contingency plans so that, if the risks do occur, immediate recovery action can be taken.

There is a process of risk management that involves several stages. Here is the outline of it:

1. Risk identification: Identify possible project, product and business risks

2. Risk analysis: Assessment of the likelihood and consequences of the risks identified
3. Risk planning: Make plans to address the risks, either by avoiding it or by minimizing its effects on the project
4. Risk monitoring: Regularly assess the risk and the plans for risk mitigation and revise the plans when more is known about the risk.

For large projects, the outcomes of the risk management process should be documented in a risk management plan. It should include a discussion of the risks faced by the project, an analysis of these, and information on how it has been planned to manage the risk if it seems likely to be a problem.

The risk management process is an iterative process that continues throughout a project. Onve an initial risk management plan has been drawn up, the situation needs to be monitored to detect emerging risks. As more information about the risks becomes available, the risks need to be re-analyzed and a decision needs to be made about whether the risk priority has changed or not. Depending on that, there may be a need to change the plans for risk avoidance and contigency management.

On an agile development process, risk management is less formal. The same fundamental activities should still be followed and risks discussed, although these may not be formally documented. However, agile development also has a downside. Because of its reliance on people, staff turnover can have significant effects on the project, product and business. Because of the lack of formal documentation and its reliance on informal communication, it is very hard to maintain continuity and momentum if key people leave the project.

2.2 Risk Identification

As seen previously, risk identification is the first stage of the risk management process. It is concerned with identifying the risks that could pose a major threat to the software engineering proecss, the software being developed or the development organization. Risk identification may be a team process in which a team gets together to brainstorm possible risks. Alternatively, project managers may identify risks based on their experience of what went wrong on previous projects.

As a starting point for risk identification, a checklist of different types of risk may be used. Six types of risk may be included in a risk checklist:

1. Estimation risks arise from the management estimated of the resources required to build the system
2. Organizational risks arise from the organizational environment where the software is being developed

3. People risks are associated with the people in the development team
4. Requirements risks come from changes to the customer requirements and the process of managing the requriements change
5. Technology risks come from the software or hardware technologies that are used to develop the system
6. Tools risks come from the software tools and other support software used to develop the system

Here is a list of the possible risks in each of these categories:

- Estimation
  1. The time required to develop the system is underestimated
  2. The rate of defect repair is underestimated
  3. The size of the software is underestimated
- Organizational
  1. The organization is restructured so that different management are reponsible for the project
  2. Organizational financial problems force reductions int he project budget
- People
  1. It is impossible to recruit staff with the skills required
  2. Key staff are ill and unavailable at critical times
  3. Required training for staff is not available
- Requirements
  1. Changes to requirements that require major design rework are proposed
  2. Customers fail to understand the impact of requirements changes
- Technology
  1. The database used in the system cannot process as many transactions per second as expected
  2. Faults in reusable software components have to be repaired before these components are reused
- Tools
  1. The code generated by software code generation tools is inefficient
  2. Software tools cannot work together in an integrated way

When the risk identification process has been finished, there should be a long list of risks that could occur and that could affect the product, the process and the business. This list then needs to be pruned to a manageable size. If there are too many risks, it is pratically impossible to keep track of all of them.

2.3 Risk Analysis

During the risk analysis process, we have to consider each identified risk and make a judgement about the probability and seriousness of that risk. There is no easy way to do it, and the project manager needs to rely a lot on their judgement and experience from previous projects and the problems that arose in them. It is not possible to make precise, numeric assessment of the probability and seriousness of each risk. Rather, the risk should be assigned to one of a number of bands:

1. The probability of the risk might be assessed as insignificant, low, moderate, high or very high
2. The effects of the risk might be assessed as catastrophic (threaten the survival of the project), serious (would cause major delays), tolerable (delays are within allowed contigency) or insignificant

After that is possible to tabulate the results of the analysis process using a table ordered according to the seriousness of the risk. To make this assessment, we need detailed information about the project, the process, the development team, and the organization.

Of course, both the probability and the assessment of the effects of a risk may change as more information about the risk becomes available and as risk management plans are implemented. We should therefore update this table during each iteration of the risk management process.

Once the risks have been analyzed and ranked, we should assess which if these risks are the most significant. The judgement must depend on a combination of the probability of the risk arising and the effects of that risk. In general, catastrophic risks should always be considered, as should all serious risks that have more than a moderate probability of occurence.

Boehm (1988) recommends identifying and monitoring the ”top 10” risks. The right number of risks to monitor, however, should depend on the project.

2.4 Risk planning

The risk planning process develops strategies to manage the key risks that threaten the project. For each risk, we have to think of actions that we might take to minimize the disruption to the project if the problem idenfiied in the risk occurs. We should also think about the information that we need to collect while monitoring the project so that emerging problems can be detected before they become serious.

In risk planning, ”what-if” questions need to be asked that consider both individual risks, combination of risks, and external factors that affect these risks. For example, questions that might be asked are:

1. What if several engineers are ill at the same time?
2. What if an economic downturn leads to budget cuts of 20% for the project?
3. What if the performance of open-source software is inadequate and the only expert on that open-source software leaves?
4. What if the company that supplies and maintains software components goes out of business?
5. What if the customer fails to deliver the revised requriements as predicted?

Based on the answers for those ”what-if” questions, we may then devise strategies for managing the risks. Usually, the risk management strategies fall into three categories:

1. Avoidance strategies: Following these strategies means that the probability of the risk arising is reduced. An example of that would be strategy to replace potentially defective components with bought-in components known for their reliability
2. Minimization strategies: Foolowing these strategies means that the impact of the risk is reduced. An example of that, for example, would be to reorganize the team to have more overlap of work and people understanding each other jobs to minimize the impact of staff illness
3. Contigency plans: Following those strategies means that the team is prepared for the worst and have a strategy in place to deal with it. An example of that would be to prepare a briefing document explaining the importance of the project and its contribuition for the business goals in case of organizational financial problems

It is obviously for the best to use strategies that avoid risks. If that is not possible, we should then use a strategy that reduces the chances of that risk having serious effects. Finally, we should have strategies in place to cope with the risk if it arises. These should reduce the overall impact of a risk on the project or product.

2.5 Risk Monitoring

Risk monitoring is the process of checking that our assumptions about the product, process and business risks have not changed. We should regularly assess each of the identified risk to decide whether or not the effects of the risk have changed. To do this, we have to look at other factors, such as the number of requirements change requests, which give us clues about the risk probability and its effects. These factors are obviously dependent on the types of risk. Factors that may be helpful in assessing these risk types are:

1. Estimation risks: Failure to meet agreed schedule; failure to clear reported defects
2. Organizational risks: Organizational gossip; lack of action by senior management
3. People risks: Poor staff morale; poor relationships among team members; high staff turnover
4. Requirements risks: Many requirements change requests; customer complaints
5. Technology risks: Late delivery of hardware or support software; many reported technology problems
6. Tools risks: Reluctance by team members to use tools; complaints about software tools; requests for faster computers/more memory, and so on

Risks should be regularly monitored at all stages in a project. At every management review, each of the key risks should be considered and discussed separately. We should decide if the risk is more or less likely to arise and if the seriousness and consequences of the risk have changed.

Section 3

Life-Cycle Models

3.1 Overview

Life-cycle models are a simplified representation of a software process. Each process model represents a process from a particular perspective and thus only provides partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities.

The generic models are high-level, abstract descriptions of software processes that can be used to explain different approaches to software development. They can be seen as process frameworks that may be extended and adapted to create more specific software engineering processes. The general process models that exist are:

1. The waterfall model: This takes the fundamental process activities of specification, development, validation and evolution and represents them as separate process phases such as requirements specification, software design, implementation and testing.
2. Incremental development: This approach interleaves the activities of specification, development and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
3. Integration and configuration: This approach relies on the availabitlity of reusable components or systems. The system development process focuses on configuring these components for use in a new setting and integrating them into a system.

There is no universal process model that is right for all kinds of software development. The right process depends on the customer and regulatory requirements, the environment where the software will be used, and the type of software being developed.

For example, safety-critical software is usually developed using a waterfall process as lots of analysis and documentation is required before implementation begins. Software products are now always developed using an incremental process model. Business systems are increasingly being developed by configuring existing systems and integrating these to create a new system with the functionality that is required.

The majority of practical software processes are based on a general model but often incorporate features of other models. This is particularily true for large systems engineering. For large system, it makes sense to combine osme of the best features of all of the general processes. We need to have information about the essential system requriements to design a software architecture to support these requriements. It is impossible to develop that incrementally. Subsystems within a larger system may be developed using different approaches.

Various attempts have been made to develop ”universal” process models that draw on all of these general models. One of the best known of these universal models is the Rational Unified Process (RUP), which was developed by Rational, a U.S. software engineering company. The RUP is a flexible model that can be instantiated in different ways to create processes that resemble any of the general process models we've seen.

3.2 The Waterfall Model

The first published model of the software development process was derived from engineering process models used in large military systems engineering. It presents the software development process as a number of stages. Because of the cascade from one phase to another, this model is known as the waterfall model. The waterfall model is an example of a plan-driven process. In principle at least, all of the process activities are planned and scheduled before starting the software development.

THe stages of the waterfall model directly reflect the fundamental software development activities:

1. Requirements analysis and definition: The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. System and software design: The systems design process allocates the requirements to either hardware or software systems. It establishes an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
3. Implementation and unit testing: During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

4. Integration and system testing: The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. Operation and maintenance: Normally, this is the longest life-cycle phase. The system is installed and put into practical use. Maintenant involves correcting errors that were not discovered in earlier stages of the life cycle, improving the implementation of system units, and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase of the waterfall model is one or more documents that are approved. The following phase should not start until the previous phase has finished. For hardware development, where high manufacturing costs are invovled, this makes sense. However, for software development, these stages overlap and feed information to each other. During design, problems with requirements are identified; during coding design problems are found, and so on. The software process, in practice, is never a simple linear model but involves feed-back from one phase to another.

In reality, software has to be flexible and accommodate change as it is being developed. The need for early commitment and system rework when changes are made means that the waterfall model is only apprapriate for some types of system:

1. Embedded systems where the software has to interface with hardware systems. Because of the inflexibitlity of hardware, it is not usually possible to delay decisions on the software's functionality until it is being implemented.

2. Critical systems where there is a need for extensive safety and security analysis of the software specification and design. In these systems, the specification and design documents must be complete so that this analysis is possible. Safety-related problems in the specification are usually very expensive to correct at the implementaion stage.

3. Large software systems that are part of broader engineering systems developed by several partner companies. The hardware in the systems may be developed using a similar model, and companies find it easier to use a common model for hardware and software. Furthermore, where several companies are involved, complete specifications may be needed to allow for the independent development of different subsystems.

The waterfall model is not hte right process model in situations where informal team communication is possible and software requirements change quickly. Iterative development and agile methods are better for these systems.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code.

The formal apprach simplifies the production of a safety or security case. This demonstrates to customers or regulators that the system actually meets its safety and security requirements. However, because of the high costs of developing a formal specification, this development model is rarely used except for critical systems engineering.

3.3 Incremental development

Incremental development is based on the idea of developing an initial implementation, getting feedback from users and others, and evolving the software through several versions until the required system has been developed. Specification, development and validation activities are interleaved rather than separate, with rapid feedback across activities.

Incremental development in some form is now the most common apprach for the development of application systems and software products. This approach can be either plan-driven, agile or, more usually, a mixture of these approaches. In a plan-driven approach, the systems increments are identified in advance; if an agile approach is adopted, the early increments are identified, but the development of later increments depends on progress and customer priorities.

Incremental software development, which is a fundamental part of agile development methods, is better than a waterfall approach for systems whose requirements are likely to change during the development process. This is the case for most business systems and software products. Incremental development reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, back-tracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the sustomer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer or user can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed, and possibly, new functionality defined in later increments.

Incremental development has three major advantages over the waterfall model:

1. The cost of implementing requirements changes is reduced. The amount of analysis and documentation that has to be redone is significantly less than is required with the waterfall model.
2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has

been implemented.. Customers find it difficult to judge progress from software design documents.

3. Early delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost effective to produce documents that reflect every version of the system.
2. System structure tends to degrade as new increments are added. Regular change leads to messy code as new functionality is added in whatever way is possible. It becomes increasingly difficult and costly to add new features to a system. To reduce structural degradation and general code messiness, agile methods suggest that you should regularly refactor (improve and restructure) the software.

The problems of incremental develoment become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. Large systems need a stable framework or architecture, and the reposibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than developed incrementally.

Incremental development does not mean that each increment has to be delivered to the system customer. The system can be developed incrementally and exposed to customers and other stakeholders for comment, without necessarily delivering it and deploying it in the customer's environment. Incremental delivery means that the software is used in real, operational processes, so user feedback is likely to be realistic. However, providing feedback is not always possible as experimenting with new software can disrupt normal business processes.

3.4 Integration and configuration

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of or search for code that is similar to what is required. They look for these, modify them as needed, and integrate them with the new code that they have developed.

This informal reuse takes place regardless of the development process that is used. However, since 2000, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a base of reusable software components and an integrating framework for the composition of these components. Three types of software components are frequently reused:

1. Stand-alone application systems that are configured for use in a particular environment. These systems are general-purpose systems that have many features but they have to be adapted for use in a specific application
2. Collections of objects that are developed as a component or as a package to be integrated with a component framework
3. Web services that are developed according to service standards and that are available for remote invocation over the Internet.

A general process model for the integration and configuration approach would be the following:

1. Requirements specification: The initial requirements for the system are proposed. These do not have to be elaborated in detail but should include brief descriptions of essential requirements and desirable system features
2. Software discovery and evaluation: Given an outline of the software requirements, a search is made for components and systems that provide the functionality required. Candidate componenets and systems are evaluated to see if they meet the essential requirements and if they are generally suitable for use in the system.
3. Requirements refinement: During this stage, the requirements are refined using information abotu the reusable components and applications that have been discoered. The requirements are modified to reflect the available components, and the system specification is re-defined. Where modificatoins are impossible, the component analysis activity may be reentered to search for alternative solutions.
4. Application system configuration: If an off-the-shelf application that meets the requirements is available, it may then be configured for use to create the new system.
5. Component adaptation and integration: If there is no off-the-shelf system, individual reusable components may be modified and new components developed. These are then integrated to create the system.

Reuse-oriented software engineering, based around configuration and integration, has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requriements compromises are inevetable, and this may lead to a system that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

Section 4

Code Of Ethics

Like other engineering disciplines, software engineering is carried out within a social and legal framework that limits the freedom of people working in that area. As a software engineer it is imperative to understand and accept that the job involves wider responsibilities than simply the application of technical skills. It is also very important to behave in an ethical and morally responsible way in order to be respected as a professional engineer.

Besides notmal standards of honesty and integritym a software engineer should not use their skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behaviour are not bound by laws but by the most tenous notion of professional responsibility. Some of these are:

- Confidentiality: Confidentiality of the employers or clients should be normally respected regardless of whether or not a formal confidentiality agreement has been signed.
- Competence: The level of competence of a software engineer should not be misrepresented by the individual, meaning that an engineer should not knowingly acceot work that is outside of their competence.
- Intellectual property rights: A software engineer should be aware of their local laws governing the use of intellectual property such as patents and copyright. They should also be careful to ensure that the intellectual property of employers and clients is protected.
- Computer misuse: A software engineer should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (playing a game on a employer's machine) to extremely serious (dissemination of viruses or other malware).

Professional societies and institutions have an important role to play in setting ethical standards, Organizations such as the ACM, the IEEE (Institute of Electronic Engineers), and the British Computer Society publish a code of professional conduct or code of ethics. Members of these organizations undertake to follow that code when they sign up for membership. These codes of conduct are generally concerned with fundamental ethical behaviour.

Professional associations, notably the ACM and the IEEE, have cooperated to produce a joint code of ethics and professional practice. This code exists in both a short form and a longer form that adds detail and substance to the shorter form. Here is the shorter form of the ACMIEEE Code of Ethics without the preamble:

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interest of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGEMENT - Software engineers shall maintain integrity and independence in their professional judgement.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ehical approach to the practice of the profession.

The general area of ethics and profession responsibility is increasingly important as software-intensive systems prevade every aspect of work and everyday life. It is therefore very important that every software engineer upholds to them whenever working their job.

Section 5

Problem Reporting and Correction

5.1 Problem Prioritising

During the life-cycle of a project, many problems can arise, and it is necessary to be able to keep track of them and correct them as they arise.

Depending on the threat that they pose to the development of the system, problems that need to be fixed and solved need to be prioritised so that the project has minimal delay.

Prioritising of problems to solve or fix needs to be done according to the risk assessment of the project, meaning that problems that cause bigger threats to the accomplishment of the project should be solved first.

5.2 Types of Problems

Many different types of problems may arise during the development of a system. Here are some of the problems that may occur when developing a system:

1. Requirement Analysis carried out incorrectly
2. Requirements change
3. Error in the software
4. Performance issue of software and hardware used
5. Delays to deliver the product
6. Staff turnover
7. etc.

5.3 Problem Reporting

Depending on the type of problem and the approach used for the development of the system, different approaches may be used to report a problem. Here are the different possible ways to report a problem:

1. Formal problem report: A standardized document describing the problem, when it occured and how it occured
2. Informal problem report: By talking to a superior or a person responsible about the problem, and communicating directly in person about it.

In an agile approach, the most common way to report a problem, especially regarding a software error (or bug) would be through an informal problem report, whilst a waterfall approach would be more keen to use a formal problem report to report a similar problem.

In regard to software errors, errors are most often found by testing the software and by user feedback. Whenever errors are found with tests, these produce test results that can be kept to track the progress in solving the error, and customer feedback can also be kept in the form of logs of issues encountered.

5.4 Problem Correction

Depending on the type of problem, a solution for the problem may have already been devised (for example in the risk assessment documents when the problem has already been identified as a risk). Other problems, such as a bug in the code, have methodologies already defined. For example, to solve a bug, the following appraoch is most often than not used:

1. Locate the error using test results
2. Modify the specification to design an error repair
3. Repair the error
4. Retest the program with the same test cases that produced the error

Section 6

Configuration and Version Management

6.1 Overview

Software systems are constantly changing during development and use. Bugs are discovered and have to be fixed. System requirements change and we have to implement these changes in a new version of the system. New versions of hardware and system platforms are released, and we have to adapt our systems to work with them. Competitors introduce new features in their system that need to be matched. As changes are made to the product, a new version of the system is created. Most systems, therefore, can be thought of as a set of versions, each of which may have to be maintained and managed.

Configuration management is concerned with the policies, processes and tools for managing changing software systems. We need to manage evolving systems because it is easy to lose track of what changes and component versions have been incorporated into each version of the system. Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems. Several versions mayy be under development and in use at the same time. If there is no effective configuration management procedures in place, there may a waste of effort in modifying the wrong version of a system, delivering the wrong version of a system to customers, or forgetting where the software source code for a particular version of the system or component is stored.

Configuration management isn't just useful for large projects, and it also very useful for individual projects, where it is easy for a person to forget what changes have been made. It is however essential for team projects where several developers are all working at the same time on a software system. In that case, the configuration management system provides team members with access to the system being developed and manages the changes that they make to the code.

Configuration management of a software system involves four closely related activities:

- Version Control: This involves keeping track of the multiple versions of system components and ensuring that changes made to components by dufferent developers do not interfere with each other.
- System building: This is the process of assembling program components, data, and libraries, then compiling and linking these to create an executable system.
- Change management: This involves keeping track of requests for changes to delivered software from customers and developers, working out the costs and impact of making these changes, and decidding if and when the changes should be implemented.
- Release management: This involves preparing software for external release and keeping track of the system verisons that have been released for customer use.

Because of the large volume of ingormation to be managed and the relationships between configurationitems, tool support is essential for configuration management. Configuration management tools are used to store versions of system components, build systems from these components, track the releases of system versions to customers and keep track of change proposals. Configuration Management tools range from simple tools that support a single configuration management task, such as bug tracking, to integrated environments that support all configuration management activities.

Agile development, where components and systems are changed several times a day, is impossible without configuration management tools. The definitive version of components are held in a shared project repository, and developers copy them into their own workspace. They make changes to the code and then use system-building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository. This makes the modified components available to other team members.

The development of a software product or custom software takes place in three distinct phases:

1. A development phase: Where the development team is reponsible for managing the software configuration and new functionality is being added to the software. The development team decides on the changes to be made to the system.
2. A system testing phase: Where a version of the system is released internally for testing. This may be a responsibility of a quality management team or an individual or group within the development team. At this stage, no new functionality is added to the system. The changes made at this stage are bug fixes, performance improvements, and security vulnerability repairs. There may be some customer involvement as beta testers during this phase.
3. A release phase: Where the software is released to customers for use. After the release has been distributed, customers may submit bug reports and change requests.

New versions of the released system may be developed to repair bugs and vulnerabilities and to include new features suggested by customers.

For large systems, there is never just one ”working” version of a system. There are always several version of the system at different stages of the development. Several teams may be involved in the development of different system versions.

These different version have many different common components as well as components or component versions that are unique to that system version. The Configuration Management system keeps track of the components that are part os each version and invludes them as required in the system build.

In large software projects, configuration management is sometimes part of software quality management. The quality manager is resposnible for both quality management and configuration management. When a pre-release version of the software is ready, the development team hands it over to the quality management team. The QM team checks that the system quality is acceptable. If so, then it becomes a controlled system, which means that all changes to the system have to be agreed on and recorded before they are implemented.

There are many terms used for configuration management, and, depending on the methodology used, different terms may mean the same concept. Here are a few of them:

- Baseline: A collection of component versions that make up a system. Baselines are controlled, which means that the component versions used in the baseline cannot be chaned. It is always possible to re-create a baseline from its constituent components.
- Branching: The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
- Codeline: A set of versions of a software component and other configuration items o which that component depends
- Configuration (version) control: The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
- Configuration item or software configuration item (SCI): Anything associated with a software project (design, code, test data, document, etc) that has been placed under configuration control. Configuration items always have a unique identifier.
- Mainline: A sequence of baselines representing different versions of a system.
- Merging: The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.

- Release: A version of a system that has been released to customers (or other users in an organization) for use.
- Repository: A shared database of versions of software components and meta-information about changes to these components.
- System building: The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
- Version: An instance of a configuration item that differs, in some way, from other instances of that item. Versions should always have a unique identifier.
- Workspace: A private work area where software can be modified without affecting other developers who may be using or modifying that software.

6.2 Version Management

Version management is the process of keeping track of different versions of software components and the systems in which these components are used. It also involves ensuring that changes made by different developers to these versions do not interfere with each other. In other words, version management is the process of managing codelines and baselines.

Baselines may be specified using a configuration language in which we define what components should be included in a specific version of a system. It is possible to explicitly specify an individual component version or simply to specify the component identifier.

Baselines are important because we often have to re-create an individual version of a system. For example, a product line may be instantiated so that there are specific system versions for each system customer. We may have to recreate the versino delivered to a customer if they report bugs in their system that have to be repaired.

Version control systems identify, store and control access to the different versions of components. There are two tyoes of modern version control system:

1. Centralized systems, where a single master repository maintains all versions of the software components that are being developed. Subverion is a widely used example of a centralized VC system.
2. Distributed systems, where multiple versions of the component repository exist at the same time. Git is a widely used example of a distributed VC system.

Centralized and distributed VC systems provide comparable functionality but implement this functinoality in different ways. Key features of these systems include:

1. Version and release identification: Managed versions of a component are assigned unique identifiers when they are submitted to the system. These identifiers allow different versions of the same component to be managed, without changing the component name. Versions may also be assigned attributes, with the set of attributes used to uniquely identify each version.
2. Change history recording: The VC system keeps records of the changes that have been made to create a new version of a component from an earlier version in some systems, these changes may be used to select a particular system version. This involves tagging components with keywords describing the changes made. We then use these tags to select the components to be included in a baseline.
3. Independent development: Different developers may be working on the same component at the same time. The version control system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.
4. Project support: A version control system may support the development of several projects, which share components. It is usually possible to check in and check out all of the files associated with a project rather than having to work with one file or directory at a time.
5. Storage management: Rather than maintain separate copies of all versions of a component, the version control system may use efficient mechanisms to ensure that duplicate copies of identical files are not maintained. Where there are only small differences between files, the VC system may store these differences rather than maintain multiple copies of files. A specific version may be automatically re-created by applying the differences to a master version.

6.2.1 Versions, Variants and Releases

- Version: An instance of a system which is functionally distict in some way from other system instances
- Variant: An instance of a system which is functionally identical but non-functionally distinct from other instances of a system
- Release: An instance of a system which is distributed to users outside of the development team

6.2.2 Version identification

Procedures should be defined that allow for version identification so there is an unambiguous way to identify the different component version. There are three basic techniques for component identification:

- Version numbering
- Attribute-based identification
- Change-oriented identification

6.2.2.1 Version Numbering

Version numbering is a simple naming scheme that uses a linear derivation. The actual derivation structure is a tree or a network rather than a sequence, where names are not meaningful and where a hierarchical naming scheme leads to fewer errors in version identification. For example, the first index of the version would indicate the version, the second would indicate the variant and the third and last the release.

6.3 Configuration Management Plan

The configuration management plan defines the types of documents to be managed and a document naming scheme, who takes responsibility for the configuration management procedures and the creation of baselines. It also defines policies for change control and version management the records which must be maintained.

It describes which tools should be used to assist the process and any limitations in their use, as well as the process for their use, as well as the database used to record the configuration information. It may also invlude information such as the configuration management of external software, process auditing and others.

6.4 Change management

A change management procedure should be put to practice to make sure that changes are tracked and that they are developed in the most cost-effective way. The following is an example of a change management process:

- Request change by completing a change request from
- Analyze change request
- If the change is valid then
  - Assess how the change might be implemented
  - Assess change cost
  - Submit request to change control board

- If the change is accepted then
  - Make changes to software
  - Submit changed software for quality approval until quality is adequate
- Create a new system version

6.4.1 Change Request Form

The definition of a change request form is part of the configuration management planning process. The form records the change proposed, requestor of change, the reason was suggested and the urgency of change. It also records change evaluation, impact analysis, change cost and recommendations.

6.4.2 Change control board

Changes should be reviewed by an external group who decide whether or not they are cost-effective from a strategic and organizational viewpoint rather than a technical viewpoint. It should be independent of the group responsible for the system. The group is sometimes called a change control board. The change control board may include representatives from client and contractor staff.

Part C

Application

Section 7

Risk Management

7.1 Risk Identification

For the FASAM, considering the current team and the type of project, the following risks have been identified:

- Estimation risks
    1. The time required to develop the software was underestimated
    2. The size of the software was underestimated
- Organizational risks
    1. The organization is restructured
    2. There is a cut in the project budget
- People risks
    1. Key staff leaves
    2. There is not enough skilled staff
    3. THere is no training available for staff
- Requirements risks
    1. Changes proposed require major re-design of the system
    2. The legislation changes
    3. Customers don't understand the impact of changes to the system
- Technology risks
    1. Faults in the communication interfaces with the hardware require reparation before being usable

    2. The technology chosen cannot cope with the performance needs of a real-time system

- Tools
    1. The code generated by software code generation tools is inefficient
    2. The different software tools don’t work together efficiently

7.2 Risk Analysis

After analyzing and studying the identified risks, here is the probability and effects identified for each of them:

- Estimation risks
    1. High probability; Serious effects
    2. High probability; Tolerable effects
- Organizational risks
    1. Low probability; Serious effects
    2. Low probability; Catastrophic effects
- People’s risks
    1. Medium probability; Serious effects
    2. Low probability; Catastrophic effects
    3. Medium probability; Tolerable effects
- Requirements risks
    1. Medium probability; Serious effects
    2. Low probability; Catastrphic effects
    3. Medium probability; Tolerable effects
- Technology risks
    1. Low probability; Serious effects
    2. Medium probability; Serious effects
- Tools risks
    1. Medium probability; Insignificant effects
    2. High Probability; Tolerable effects

After knowing this, we were able to prioritize the order in which to address the risks. To do that, we prioritized first the risks with the most effect, and, inside the group of risks with the same level of effects, we prioritized the one with the most likelihood of occuring, which leads us to the following list of risks to address:

1. There is not enough skilled staff
2. Legislation changes
3. There is a cut in the project budget
4. The time required to develop the software was underestimated
5. Key staff leaves
6. Changes proposed require major re-design
7. The technology chosen cannot cope with the performance needs of a real-time system
8. Faults in the communication interfaces with the hardware require reparation before being usable
9. The organization is restructured
10. The size of the software was underestimated
11. The different software tools don't work together efficiently
12. Customers don't understand the impact of changes to the system
13. There is no training available for staff
14. The code generated by software code generation tools is inefficient

7.3 Risk Planning

Considering that FASAM is a large-scale, safety-critical and real-time software certain aspects need to be taken more into account than in other types of projects. For example, delivering a software that is compliant with the current legislation and made by skilled people is very important, otherwise the security of people could be put at risk. With that in mind, the follwiing strategies have been devised to avoid the risks or minimize their effect. The list of strategies follows the numbering used for the priotization of the risks, therefore indicating that the strategy is to be applied for that risk:

1. Avoidance: Plan needed stagg in advance; Minimizing: Hire and train staff as necessary

2. Avoidance: Prepare for change in advance if changes are previewed. Minimizing: Keep updated about possible legislation changes
3. Avoidance: Listen to company rumors and plan the budget wisely; Contigency: Prepare to pitch the organization about the importance of keepping the budget
4. Avoidance: Give stakeholders a timeframe bigger than originally thought of about three weeks and give more time than planned for staff to complete tasks; Minimizing: Keep stakeholders up-to-date with the finish date
5. Avoidance: Make sure staff is happy; Minimizing: Train other people to take over position
6. Avoidance: Make software desigfn moduler for easier change; Minimizing: Talk to customers about the implications of change, plan more time in the timeframe for change
7. Avoidance: Study technologies carefully before choosing the ones for the project and hire experts in the technology chosen.
8. Avoidance: Choose hardware that has good feedback in similar setups and choose carefully the interfaces to use to communicate with them
9. Minimizing: Make sure everything is documented and at least one member of staff knows all the ins and outs of the project to put someone else up to speed quickly
10. Avoidance: Analyse the requirements carefully and thoroughly design the system to give the development and management team a good idea of the size of the project. Minimizing: Make staff work extra hours and give the customer thorough explanations on why it is taking more time to finish the project.
11. Avoidance: Plan to use tools that communicate well together. Minimizing: Make sure all staff is trained on how to carry over information between the incompatible systems.
12. Avoidance: Explain to customers the possible impact of changes; Minimizing: Always give the customers a revised timeframe and budget for the requested changes.
13. Avoidance: Choose widely used technologies; Minimizing: Hire staff experienced in that particular technology
14. Avoidance: Don't use code generation; Minimizing: Get experienced staff to verify the performance and efficiency of the auto-generated code and optimize it if necessary.

Those plans aren't the best, but give an idea of what to do in order to avoid and cope with the different risks and problems that may arise during the development of the software. They may need to be revised or different actions might have to be taken when monitoring the risks and as the development goes.

7.4 Risk Monitoring

Throughout the project, some risks happened and, because of our risk management strategy, we managed to deal with them effectively, allowing us to deliver the software without much delay and change in budget.

The following risks and ways to deal with them occured:

- The time required to develop the project was underestimated
  - We took longer than we thought it would take to complete the project
  - The risk didn't really affect the delivery of the project on time since we asked the client for more time to finish the project than we thought we would need
- There is not enough skilled staff
  - When we first started the project we realized that we didn't have skilled enough staff to develop the software.
  - We hired skilled enough people before even finishing the requirements gathering and analysis, therefore avoiding the risk from taking big proportions
- Customers don't understand the impact of change
  - Clients often asked us for changes that would require major redesign and even sometimes go against safety legislations
  - We had to negotiate with them and compromise on the changes they wanted to be made in order for the software to be developed on time and to comply to all health and safety legislations

Overall, not that many risks occured during the development, which we are grateful for, but it was still good to have plans to avoid most of the risks, and contigency and minimizing plans should the risks still occur.

Section 8

Life-cycle model

We chose to use the waterfall life-cycle model for the FASAM project.

As seen in the theory part of this report, the waterfall cycle model is very suitable for large-scale, safety-critical system, which is exactly what the FASAM project is.

Because of the implications that the project could have in the safety of members of the public, it is imperative that all requirements and safety measures to be implemented in the system are fully understood before beginning the design and implementation of the system, which is why the waterfall model is the most suitable for this type of project.

That fact that everything needs to be documented was also another factor that inclined our choice, since we can more easily keep track of all the requirements and safety features to be implemented, therefore leaving no room for aspects to be left out of the system.

Section 9

Code Of Ethics

FASAM intends to adhere fully to the ACMIEEE code of ethics. In order to do that, we will make sure that:

- All team members know what the code of ethics are and adhere to item
- Penalize anyone that doesn’t follow the code of ethics

FASAM is a safety-critical system, where people’s lives can be at stake if the caution isn’t taken and the code of ethics isn’t followed at all times. We need to put the safety of people before anything else when building this type of system, even if that means taking longer to build the project or going over budget. It is in the best of the public interest that everyone affected by the system stays safe, and therefore that will be our first priority in regards to the development of the project, from requirements gathering to release to the public.

Section 10

Problem Reporting and Correction Startegy

Because of the nature of the project and the methodology chosen, the approach taken to report problems and correct them during the development of the FASAM project will be of a very much formal nature. In order to report problems, members of staff will fill in a form describing the problem encountered, how the encountered it and a solution that they believe will solve the problem.

If the nature of the problem is in the code itself, that will be done thorugh a bug report in the version management system, otherwise it will join the rest of the documentation for the project.

All code related problems (meaning bugs), will be solved using the usual procedures of bug detection and correction.

After the problem has been found, all members envolved in that part of the project are to be gathered in a meeting to discuss the issue and devise a strategy to solve the problem, taking into account the possible solution written in the problem report form. Minutes of decisions made during that meeting are to be taken and added to the problem log.

Whenever a trial has been done to solve a problem, the same must be logged and the results of which must join the documentation as well.

After the problem has been solved, the problem must be declared so in the log so that all team members are aware that the problem no longer exists.

10.1 Example

During the development of FASAM, we encountered one problem with our documentation, where some requirements were missing that were to be implemented.

The person that found the problem opened a log in our management system and reported the problem to their superior, which then went and gathered the people responsible for documenting the requirements. They sat down and figured out which requirements were missing from the documentation that were present in the code, by going thorugh the version management system to find all the design processes for those requirements and the first elicitation of said requirements.

They then solved the problem by gathering all the information about those requirements and adding them to the documentation. Whilst doing that job, they added their progress to the problem log in the management system, and, when the problem was solved, they declared the problem as solved by closing the issue.

Section 11

Configuration Control and Version Management Strategy

In order to keep track of the software versions and the changes made to the system as it goes, we have decided to use the following software tools:

- Git as a version management system
- GitHub as the platform hosting the repository

By using these software tools, we have a system that allows us to completely keep track of the versions of the software that are being developed and those who have been currently released. Furthermore, because of the way GitHub allows users to work in their repositories, we can also keep track of issues and problems as they arise and allows us for a better organization of features to be implemented, bugs to be fixed, etc for the different version.

Thanks to the branching system of Git, we can also create different branches for when trying to implement a new feature or fix a bug without changing any of the code from the released version, allowing us to make the changes and new versions easily and effortlessly.

Versions released can be managed through the commit used when a release has been made from the master branch, which allows us to know at which point of the code the version was released at, and rollback if a customer is having an issue with that particular version of the software in an individual workspace.

To keep track of the versions released, we decided upon using the numbering system explained in the theory part of this work, adding a name to each major version for easier distinguishment of the versions.

This configuration control and versioning system came in handy when a user reported to us a bug on an older released version of the system. Thanks to our numbering system, we could very quickly find the release that the user was talking about, and thanks to the Git system, the person in charge of solving the issue could very quickly roll back their private repository to that version, create a new branch and work on solving the issue, to then release a new and improved release for that same version of the software.

References

Boehm, B. (1988), *A Spiral Model of Software Development and Enhancement*, IEEE Computer.

Sommerville, I. (2016), *Software Engineering*, Peason Education.