

Chapter 1

What is software? Computer program with a set of documentation (system design and structure and user documentation), configuration data and all additional sources of information directly related to the program.

There are 2 types of software product:

1. **Generic product.** Stand-alone systems, sold on open market to everyone. Examples – desktop operating system, office packs or graphical editors.
2. **Customized (bespoke) products.** Developed for particular customers with special features. Examples – special systems for traffic control or airbus flight operations.

From developers' stand of view, one of the main differences of those types is that for customized case the specifications for a software product is done mostly by the organization which buying it. For generic product development, specifications are created by the same development organization.

Nowadays, there is no straight line between those types as there was several years ago.

Software engineers, as any other engineers, make things work by applying theories, methods and tools (created, mostly, by computer scientists, but when it comes to practical problems, there's often need to change the solution and develop your own way, still, based on some theory). Software engineering is not all about coding programs, there are much more problems to solve during the work on project, for example project management, scheduling of work, testing and choosing the right way to go.

Software engineering is concerned with all aspects of the development and evolution of complex systems. System engineering is concerned with hardware development, policy and process design.

Software process consists of four fundamental activities:

1. Software specification where engineers or/and customers define what the product should do and how should it operate.
2. Software development is designing and actual coding.
3. Software validation is generally testing. It is important to check if the system is designed and implemented correctly.
4. Software evolution is modifying the system according to new needs of customer(s).

Different types of software need different development process.

Software process model is a simplified description of a software process that presents one view of a process. And again, choice of a view depends on the system developing, sometimes it is useful to apply a workflow model, sometimes, for example – a role/action model.

Most software process models are based on one of three general models or paradigms of software development.

1. The waterfall approach. In this case the development process and all activities are divided into phases such as requirement specification, software design, implementation, testing etc. Development goes phase-by-phase.
 2. Iterative development. An initial system is rapidly developed from very abstract specifications. Of course, it can be reimplemented according to new, probably more detailed specifications.
 3. Component-based software engineering (CBSE). The development process is done assuming some parts of the system already exist, so the process focuses on integrating parts together rather than developing everything from scratch.
-

A software engineering method is a structured approach to software development whose aim is to facilitate the production of high-quality software in a cost-effective way. If not talking about history, today most common method is based on UML (Unified Modeling Language), which was created from different approaches from the past.

As any other thing related to software engineering, methods may vary depending on different situations.

For a software engineer it is always important to make sure the product is good, and there are different tools to analyze that. Overall, good piece of software should meet these attributes:

1. Maintainability. Software should be developed in such way that it can be changed according to new specifications and needs of customers. This aspect is directly related to software evolution.
 2. Dependability. It includes reliability, security and safety. Good dependable system should make minimum (or no) economic damage in case of failure.
 3. Efficiency. Software should be efficient and should not waste system resources.
 4. Usability. With all hidden for an end-user requirements, first thing customer think of when they start using the program is how easy to do it and how good the documentation is. Software should be usable without too much effort from user. Therefore, developers should always keep in mind the type of an user system building for.
-
-

My thoughts

First chapter is introduction and most of things look familiar for me, especially after attending COMP3104 lectures. I think about software development as very complex process, which technically never ends because always has something to make better or add more features. While facing big problems and developing large systems, the necessity of every point of development become stronger. As such, I understand why maintainability is so important for both developers and customers.

Software engineering is a lot of responsibility: to other engineers, to customers and society around the product. It is a big challenge – applying theoretical methods on practice.

Chapter 2

The word 'system' is widely used, but the useful working definition of a system when we talk about particular parts and particular goals sounds like:

A system is a purposeful collection of interrelated components that work together to achieve some objective.

No matter how complicated and large any system might be, we can still apply this definition.

Now if we talk about software systems, there are two categories:

1. **Technical computer-based.** This type of system includes hardware and software but not procedures and processes. Most of simple desktop software, OS etc. are examples of technical computer-based systems. Main point is: using this software has a purpose, but software itself does not know it, knowledge of this purpose is not part of a system.

2. **Socio-technical systems** includes hardware or/and software (actually, we can say it includes technical systems), but most important part it includes knowledge of how the system should be used and operated in order to achieve final objective. This kind of system includes predefined operational processes, people (to operate a system). Carleton University is a socio-technical system, because all of resources (library, buildings, computer labs, all physical things as well as general "knowledge") are operated and used by certain people (profs, staff and students).

Socio-technical system is different from technical not only because of people. The book outlines three main distinctions:

- Parts of socio-technical system have properties that are properties of the system as a whole rather than associated with those parts particularly.

- Such system may be nondeterministic, which means that the same input don't always produce the same output. System works and operated by human, who doesn't always react the same way as some software.

- The way system works and achieves its objectives does not just depend on the system itself. It also depends on the stability of these objectives.

We should always keep in mind that the main principle of system is one of the main problems – system cannot be fully functional if even one part of it does not work.

Interdependency causes this statement. This statement must be applied not only when we look at system and its parts, but also when we look broader – at different systems within another system.

For example, we have a software program which is a system itself and consists of different components. They all work properly and system designed correctly, but if we actually plug this program into operating system (which is another system), and that OS operates all the hardware (which is also a system), we cannot be sure that the program will work as it suppose to. As a software engineer, person should always think wide.

One another source of possible problems is the ability of the system to involve and to be changed to accommodate modified system engineering requirements. It is very common when during such change software failure happens, even though software was working properly before an attempt to change it. Therefore, this kind of software failure is actually not software failure, but rather some mistake or series of mistakes during designing the system, so that modifying gone wrong.

There is a complex relationship between system components and this very fact says that the system is more than just a box with some functional stuff in it. Parts and relationships have different emergent properties and those properties are properties of the whole system. Often those properties can only be derived from sub-system interrelationships and are "invisible" when we look at particular system component. There are two types of emergent properties:

1. **Functional.** Everything related to how the system works. For example, functional requirement of a phone is to be able to make and receive calls.

2. **Non-functional.** Everything related to behaviour and operating environment. For that same phone non-functional emergent requirement may be "to be easy to use".

Emergent properties are often very complex and it is hard to describe them. It is even harder to analyze some properties. For example, reliability. First of all, we should know that reliability is such a property which cannot be seen and analyzed without detailed analyze of every part of a system. Even if 99 of 100 parts of a system are super reliable and solid-stone-never-fail-parts, one little piece may cause failure of a whole system.

There are three related influences on the overall reliability of a system:

1. Hardware reliability
2. Software reliability
3. Operator reliability

This is very obvious, because any problem may come from one of three main parts of a socio-technical system.

System engineers are not just concerned with software but also with hardware and human interaction. They must think of services that system provides and how useful and easy to use different parts may be.

System requirements specify what the system must do and its properties. An important part of the requirements definition is to establish a set of overall objectives that the system should meet. A fundamental difficulty in establishing system requirements is that the problems are so complex, that there are so many related entities and there is no definitive problem specification. The true nature of a problem emerges only when a solution is created.

System design is concerned with how the system functionality is to be provided by the components of the system. Process of designing a system is fairly complicated and consists of different steps or phases. It starts with organizing requirements into related groups, then – identifying sub-systems that can (alone or interacting with each other) meet those requirements and cover all the groups of them. Then developers should actually assign requirements to sub-systems. Then, every sub-system's functionality should be specified. It is also useful to specify relationships between sub-systems. When this step is done and all sub-systems are at least in some close to working state, software engineer should define sub-system interfaces. When this is done, it is possible to continue to develop these sub-systems in parallel.

Designing is one of the first and most important steps of product development. For almost all systems, there are many possible design decisions that meet the requirements. In perfect situation, developers choose design as they thing is the best for given requirements, but pretty often another facts affect this choice – customer's desire, federal laws etc.

Next big step in developing is system modelling. Model of a system is an abstract representation of all parts and mechanisms of system. Usually model of a system is developed and shown as graphical image. There might be different scales of a model so it is possible to look at the system from different points of view and different scale.

Sub-system development is basically process of creating building blocks for a project. In fact, those sub-systems are not always developed from scratch for purposes of this particular project. Quite often developers can use sub-systems of different projects or even integrate some side-developed components. Sub-systems are usually developed in parallel. If the system developing process goes step-by-step, one component after another, it becomes very hard and expensive to make modifications. And there is (almost) always modifications to be done during development.

Systems integration is a process of putting all independently developed sub-systems into one. Integrating all the components at a time sounds like good way, but on practice it is not always possible, because development of different sub-systems takes different time. Also, adding sub-systems one-by-one or at least by blocks or sections reduces number of errors.

At finally, software evolution. This process is more abstract and doesn't not directly related to first development, but still is very important. While developing software systems, changes can come even during designing and implementation, and of course more changes will come after some period of use. There may be just bug-fixing changes or adding new

requirements. Changes should not be just technically motivated, different points of view should be taken to analyze process, such as business and technical prospective.

System decommissioning means taking the system out of service after the end of its useful operation lifetime. Hardware can be recycled, software can be reused for some different purposes or just be 'forgotten'. Anyway, for a software, decommissioning is not such a big problem.

Software development might be such a complex process, so it is impossible to create a product alone or create it within small group of developers. In a complex projects organisational processes are important.

My thoughts:

After this chapter I can see the difference between two general types of systems when it comes to software or hardware, also now I understand why different objectives demand different sort of system. Any technical operation (mostly cyclic, or with no need to analyze situation in real-time, especially when it comes to something moral-related) can be done easily by technical systems, but there's no such machine that can do anything without human. Socio-technical systems come along when there's an objective and work on something to achieve that objective needs several decisions to make, and quite often those decisions can be made only by human.

Socio-technical systems seems more flexible and useful in many cases, still, there are a lot of disadvantages of this system. Human resource is not the best when it comes to errors (mistakes) and supporting socio-technical system in general is much more expensive.

Steps of developing model are familiar to me in general, but there are some new details we didn't face while developing model for our Monopoly game, because even though we work almost as people work during large project, we didn't face the same problems as, for example, Mac OS developers do. That's why I found this chapter useful.

Chapter 3

Errors, mistakes and fails in software are common, usually a fail cause inconvenience but no serious long-term damages or something as serious as huge money loss or even health damage. However, in some systems failure can have very big and serious consequences. This type of system is called **critical system**. There are three main types of critical systems:

1. Safety-critical systems. Fails in this system may result in injury, death or environmental damage. For example, space shuttle with astronauts on board. If something goes wrong with navigation system, people may die.
2. Mission-critical systems. Fails in this system may result in failure of some goal-directed activity and main objective of the system may not be reached. The same space-shuttle is an example of mission-critical system, because even without astronauts taken in count, the whole mission might be failed.
3. Business-critical systems. Fails may cause loss of money for customers using this system. Bank money management system is an example.

The most important emergent property of a critical system is dependability. Systems that are unreliable and unsafe are often rejected by users. Possible failure cost may be so big users refuse to use the system. A system that may easily loose the information is very unsafe too, because data is often the most expensive part of organization.

As a summary of all seriousness of critical systems and software fails we can say that only trusted methods and techniques must be used for development.

Dependability of a system is the main component in “calculating” trustworthiness. Trustworthiness is a degree of user confidence that the system will operate exactly as it suppose to. Of course, calculating is not the right word, because such a value cannot be expressed numerically, but some abstract terms like “not dependable”, “very dependable” are used.

Trustworthiness and usefulness are not the same and not even directly related. Program may be very useful and easy to work with in many areas, but it may crash every time user hits more than three buttons at a time. Or vice versa, system may work as a solid stone, but all it does is printing random numbers.

Four principal dimensions to system dependability are: Availability, Reliability, Safety and Security.

All of these may be decomposed into another, for example security includes integrity (ensuring that data is not damaged) and confidentiality. Reliability includes correctness, precision and timeliness. All of them are interrelated.

Some other system properties may be considered under the heading of dependability:

1. Repairability. How easy and fast system can be restored after fail. Unfortunately, most of today’s software systems use a lot of third-party components and therefore system’s repairability does not depend on system only
 2. Maintainability. How east and fast system can be changed to adopt new requirements and rules.
 3. Survivability. Ability of a system to continue work and deliver services after fail or attack.
 4. Error tolerance. Ability of a system to avoid input errors
-

System availability and reliability are closely related to each other. Both of them can be expressed as numerical probabilities – availability is the probability that system will be up and running to deliver services; reliability is the probability that the system will work correctly.

More precise definitions are:

- Reliability – the probability of failure-free operation over a specified time in a given environment for a specific purpose.
- Availability – the probability that a system, at a point in time, will be operational and able to deliver the requested services.

By definition, the environment for a system is quite important and has to be taken into account. Measuring the system in one environment doesn’t mean it will work with same results in different environment.

Three complementary approaches that are used to improve the reliability of a system are:

1. Fault avoidance. Development techniques used to minimise the possibility of mistakes before they result in system faults.
2. Fault detection and removal. Identifying and solving system problems before the system is used.

3. Fault tolerance. Techniques used to ensure that some system errors doesn't not result in failure.
-

Safety-critical systems are systems where it is essential that system operation is always safe. The system should never damage people or system's environment even in case of failure. There are two classes of safety-critical systems:

1. Primary safety-critical software. This is software which is used as a controller in a system.
2. Secondary safety-critical software. This is software that can be indirectly result in injury.

There is no 100% safe and reliable system, so various methods are used to assure safety. What we can do is to ensure that accidents do not occur or the consequences of an accident are minimal. Three complementary ways of doing that are:

1. Hazard avoidance. System is designed in such way so that hazards are avoided, for example cutting machine can be ran only by pressing two buttons at the same time, so operator's both hands are busy (still, he has plenty of other stuff that can be cut)))
 2. Hazard detection and removal. System should have some components responsible for analyzing possible hazards and removing them, for example speed limiter for cars.
 3. Damage limitation. If damage happened, system should make the result minimum. Cars always have airbags.
-

Security is a system attribute that reflects the ability of the system to protect itself from external attacks that may be accidental or deliberate. Nowadays, security is very serious issue, especially for Internet or network-related systems. Mistakes in designing security system can cause system faults because of possible attacks. Three types of damage that may be caused through external attack are:

1. Denial of service. The system may be forced into a state when it doesn't deliver services anymore and (from user's point of view) doesn't work at all.
2. Corruption of programs or data. The system itself may be corrupted under attack, but also the data system operates and has access to.
3. Disclosure of confidential information. The information system operates and has access to may be exposed to unauthorised people.

My thoughts:

We all got used to software failures and it became so common for us that programs are unstable and you can never be sure. I think this is bad thing, really bad, that's why working on this chapter I was thinking of some "perfect future" where all software-based systems somehow managed to work perfectly fine, but actually, I don't think this will ever happen. As you were saying on one lecture about mission-critical systems, the same software bug as there was 20 years ago came out and caused big damage. Even though computers are machines, they do mistakes, because they were created by human, and human indeed do mistakes.

Chapter 4

Software process is a set of activities that leads to the production of a software product. There's no, of course, ideal software process and every project needs different. Software processes are complex and, like all other intellectual and creative processes, rely on people making decisions and judgements.

Still, there are some common parts for almost any software process:

1. Software specification. Defining the functionality of the software and constraints on its operation.
2. Software design and implementation.
3. Software validation. Testing the product to ensure it meets all requirements.
4. Software evolution.

Software processes can be improved by process standardisation where diversity in software processes across an organisation is reduced.

Three process models covered in this chapter are:

1. Waterfall model. Dividing fundamental processes (specification, implementation etc) into phases.
2. Evolutionary development. An initial system is rapidly developed from abstract specifications.
3. Component-based software engineering. Development process is focused on integrating reusable components into a system rather than developing them from scratch.

In practice, of course, it is often impossible to work all the way through within only one model. For example, sub-systems may be developed using different model than main system. Often, models are combined.

Principal stages of the waterfall model are:

1. Requirements analysis and definition.
2. System and software design
3. Implementation and unit testing
4. Integration and system testing
5. Operation and maintenance.

The main principle is that the next phase cannot begin before previous is done. In practice, these stages may overlap and feed information to each other. During design, problems with requirements are identified. During coding design problems are found and so on.

During the final life cycle phase (operation and maintenance) the software put into use. Errors and omissions in the original software requirements are discovered. The system must evolve to be useful.

The advantages of waterfall model are that documentation is produced at each phase and that it fits with other engineering process models. Its major problem is its flexible partitioning of the project into distinct stages. Commitments must be made at early stage in the process, which makes it difficult to respond to changing customer requirements.

There are two fundamental types of evolutionary development:

1. Exploratory development where the objective of the process is to work with the customer to explore their requirements and deliver a final system. The development starts with the parts of the system that are understood. The system evolves by adding new features proposed by the customer.
2. Throwaway prototyping where the objective of the evolutionary development process is to understand the customer's requirements and hence develop a better requirements definition for the system.

This approach is often more effective than the waterfall approach in producing systems that meet the immediate needs of customers. However, there are two major problems with this model:

- 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.
- Systems are often poorly structured.

These problems are acute when the project is large, for the small systems (up to 500,000 lines of code) the author of the book thinks that the evolutionary process is the best choice.

In the majority of systems there is some software reuse. Pretty often this happens informally, when people working on the project know of designs or code which is similar to that required. They look at these, modify them and reuse in current project. It is often essential for rapid system development.

Stages for software process using component-based model are:

1. Component analysis
 2. Requirements modification
 3. System design with reuse
 4. Development and integration.
-

Change is inevitable in all large software projects. The system requirements change as the business procuring the system responds to external pressures. Management priorities change; new technologies become available, design and implementation change. This means that the software process is not a one-off process; rather, the process activities are regularly repeated as the system is reworked in response to change requests. There are two process models that have been designed to support process iteration:

1. Incremental delivery. The whole process of software development are broken down into a series of increments that are each developed in a turn. This process is

something between the waterfall model and evolutionary model. Customers identify the services to be provided by the system. They also identify which of the services are most important for them and which are less. After basic development, customers take early delivery of part of the system and then they can experiment with the system. This helps them clarify their requirements for later increments. New increments are developed and combined with the existing parts, so the system is getting better and more functional after each increment.

2. Spiral development. The development of the system spirals outwards from an initial outline through to the final developed system. This process may be represented as a spiral. Each loop represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility. Next loop – with requirement definition, etc.

My thoughts:

I really think mixing different software process models can give better results than using just one model. Even for such a small project as our Monopoly game, we started as a waterfall model: defining requirements and use-cases first, then – designing the system, components and signals. When coding takes place, I think it might be better to switch the process to something between waterfall model and evolutionary development. This sounds like incremental delivery, where customers are us, developers ☺ We develop some part of a system, make it executable and testable and test this part as if we were end-customers. This kind of work will reduce some errors, which can be seen only while executing and testing.

Chapter 5

Software project management is an essential part of software engineering. Good management cannot guarantee project success, however, bad management usually result in project failure. Software managers do the same kind of job as other engineering project managers. However, software engineering is different from other types of engineering. Some differences are:

1. The product is intangible. The software cannot be seen or touched; it is not physical at all.
2. There are no standard software processes. There is no specified process for particular task and quite often it is hard and takes time to manage which software development process is more suitable for the project.
3. Large software projects are often “one-off” projects. Large software projects are often very different in some ways from previous project. Therefore, even managers with large experience may find it difficult to anticipate problems.

Main activities of a software manager are:

- Proposal writing
- Project planning and scheduling
- Project cost
- Project monitoring and reviews

- Personnel selection and evaluation
- Report writing and presentations

Writing proposal to win a contract to carry out the work is important possible stage. It usually includes cost and schedule estimates and justifies why the project contract should be awarded to a particular organisation or team.

Project planning is concerned with identifying the activities, milestones and deliverables produced by a project.

The manager should keep track of the progress of the project and compare actual and planned progress and costs. Informal monitoring can often predict potential problems by revealing difficulties as they occur.

During a project, it is normal to have several reviews.

Project managers usually have to select people to work on their project. Ideally, skilled staff with appropriate experience will be available to work on the project, but in reality managers (in most cases) have to settle for a less-than-ideal project team. The reasons for this are:

1. The project budget may not cover the use of highly paid staff.
2. Staff with the appropriate experience may not be available either within an organisation or externally.
3. The organisation may wish to develop the skills of its employees, so inexperienced staff may be assigned to the project to learn and to gain experience.

Chapter 6

Requirements for the system are the descriptions of the services provided by the system and its operational constraints. The process of analyzing, documenting and checking services needed is called Requirements Engineering.

Requirement is quite important and widely used word in software engineering, but still it not used in industry in consistent way. In some cases requirement is some high-level, abstract statement of a service that system must provide. In some other cases – it is detailed definition of a system function.

There exist two general types of requirements while developing a system:

1. **User requirements** are statements in natural language, of what services the system is expected to provide and the constraints under which it must operate. They should be concerned on functional and non-functional requirements from user's point of view, describing in detail aspect of a system that are important for customer. While working on user requirements, it's important to avoid everything related to system design.
2. **System requirements** is a set of system functions descriptions in detail. Also, requirement document should be precise and define exactly what it is to be implemented. System requirements for software engineers are starting point for developing system, because they describe the way system should be designed and implemented. How user requirements should be provided.

Software system requirements are often classified as functional, non-functional or domain requirements.

Functional requirements describe what system should do, how it should react to different inputs and systems behavior at different situations. Statements of functional requirement should tell exactly what should happen in which cases, still, these requirements keep pretty abstract.

Non-functional requirements are constraints on the services of functions offered by the system. They include such aspects of development as timing, development process and standards. Non-functional requirements can be related to emergent properties such as reliability, response time etc. They are rarely associated with particular system features and more often used to describe more general aspects of a system, for example “user friendly graphical user interface”.

Domain requirements come from the application domain of the system and that reflect characteristics of that domain. They come more from the system itself rather than from specific need of system users. Domain requirements are important because they often reflect fundamentals of the application domain.

Chapter 7

The goal of the requirement engineering is to create and maintain a system requirement document. The overall process includes four high-level sub-processes. These are concerned with assessing whether the system is useful to the business (feasibility study); discovering requirements (elicitation and analysis); converting these requirements into some standard form (specification); and checking that the requirements actually define the system that the customer wants (validation).

Feasibility study produces feasibility report and leads to next process – requirements elicitation and analysis. Then, system models and requirements specification produced, next – user and system requirements and requirements validation processes come, and as result of all these sub-processes (they also rely on each other during requirement engineering process) the requirement document developed.

Feasibility studies answers number of questions:

1. Does the system contribute to the overall objectives of the organisation?
2. Can the system be implemented using current technology and within given cost and schedule constraints?
3. Can the system be integrated with other systems which are already in place?

In a feasibility study in it often that you consult information sources such as the managers of the departments where the system will be used, software engineers who are familiar with the type of system that is proposed, technology experts and end-users of the system.

In the next stage, the requirements elicitation and analysis, software engineers work with customers and end-users of a system to find out about application domain, what services the

system should provide, the general performance of the system. When these general requirements are ready, developers should check if the requirements actually define the system that the customer wants. These checks include:

- Validity checks. Further thought and analysis may identify additional or different functions that are required.
- Consistency checks. Requirements should not conflict.
- Completeness checks. Requirement document should include requirements, which define all functions and constraints intended by the system user.
- Realism checks. Using knowledge of existing technology, the requirements should be checked to ensure that they could actually be implemented.
- Verifiability. To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable.

The requirements for large project are always changing. Often, because the problem cannot be fully defined, the software requirements are bound to be incomplete. Requirement management is the process of understanding and controlling changes to system requirements. This includes keeping track of individual requirements and maintaining links between dependent requirements, establishing a formal process for making change proposals and linking these to system requirements.

My thoughts:

Requirements are, I believe, the most known part of this book to me. We discussed requirements quite a lot, and I see why they are important. During requirement engineering process (if I can call it that fancy way :)) for our Monopoly game, we faced problems, discussed in this chapter. For example, some sort of consistency checks were done after Iteration 1 and we found some requirements conflicts and overlap one another. Completeness check showed that we still missing two requirements, not really important and basic ones, but still, if we were starting implementation, at some point we would come back and think again.

We didn't do any feasibility studies, except for maybe one question – “Can the system be implemented using current technology and within given cost and schedule constraints?”. Given cost is a term mark and schedule was set, so we didn't really deep thinking and analysing. For large and real project, this is much more serious questions.

Chapter 8

User requirement should be written in natural language to be understandable by non-technical people, but during developing process requirements may be expressed in more technical way. One widely used technique is to document the system specification as a set of system models. Models are graphical representation of what system should do and how it should operate, that's why models are often more easy to understand than even detailed natural language description. Models are also an important bridge between the analysis and design process.

At an early stage of requirement engineering developers should define the boundaries of the system. This involves working with system stakeholders to distinguish what is the system and what is the system's environment. These decisions should be made early to limit the cost of developing and time for analysis.

Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Producing a simple architectural model is the first step in this activity.

Architectural models describe the environment of a system. However, they don't show the relationship between the other systems in the environment and the system that is being specified. Therefore, simple architectural models are normally supplemented by other models, such as process models, that show the process activities supported by the system.

Behavioural models are used to describe the overall behaviour of the system. Two types of behavioural models exist:

- Data-flow models, which model the data processing in the system. It is an intuitive way of showing how data is processed by a system. Data-flow models are valuable because tracking and documenting how the data associated with a particular process moves through the system helps analysts understand what is going on. This type of models show a functional perspective where each transformation represents a single function or process. They show the entire sequence of action that take place from an input being processed to the corresponding output that is the system's response.
- State machine models, which model how the system reacts to events. It shows system states and events that cause transition from one state to another. It does not show the flow of data within the system. A state machine model of a system assumes that, at any time, the system is in one of a number of possible states. When a signal is received, trigger may switch a transition to a different state.

These types of models can be combined.

Most large software systems make use of a large database of information. In some cases, this database is independent of the software system. In others, it is created for the system being developed. An important part of systems modelling is defining the logical form of the data processed by the system.

The most widely used data modelling technique is Entity-Relation-Attribute modelling (ERA modelling), which shows the data entities, their associated attributes and the relations between these entities. These models have been widely used in database design.

Like all graphical models, data models lack detail, and you should maintain more detailed descriptions of the entities, relationships and attributes that are included in the model. More detailed descriptions are often stored in a repository or data dictionary. A data dictionary is an alphabetic list of the names included in the system models. As well as the name, the dictionary should include an associated description of the named entity and, if the name represents a composite object, a description of the composition.

An object-oriented approach to the whole software development process is now commonly used. This means expressing the system requirements using an object model, designing using objects and developing the system in an object-oriented programming language.

Developing object models during requirements analysis usually simplifies the transition to object-oriented design and programming. However, as author notices, end-users often find object models unnatural and difficult to understand.

Object-oriented modelling involves identifying the classes of object that are important in the domain being studied. These are often organised into taxonomy. A taxonomy is a classification scheme that shows how an object class is related to other classes through common attributes and services. To display this taxonomy, the classes are organised into an inheritance hierarchy with the most general object classes at the top.

The design of class hierarchies is not easy, so the analyst need to understand, in detail, the domain in which the system is to be installed.

As well as acquiring attributes and services through an inheritance relationship with other objects, some objects are groupings of other objects. The classes representing these objects may be modelled using an object aggregation model.

A structured method is a systematic way of producing models of an existing system or a system that is to be built. Structured methods provide a framework for detailed system modelling as part of requirements elicitation and analysis.

My thoughts:

As everything big and important today, such process as software engineering is based on some abstract things. System models are abstract and detailed at the same time. Different types are used, but again, as I said before, my opinion is "mixing is better", or in this case using different types of models to describe different sides of development have to be done.

Modelling the system is important, because this is the base, most fundamental part of actual work after requirements are defined, or at least part of them. Even for such a small project as our Monopoly game, I cannot imagine starting coding from scratch with no requirements defined and model developed, especially when there are four people in the team. Models are also useful to divide work between developers I guess: when signals are defined and model is created, different components may be implemented by different people, sometimes – in the different parts of Earth (like most of Open Source developers work).

Chapter 9

Specification for critical systems is important. Because of the high potential costs of system failure, it is important to ensure that the specification for critical systems accurately reflects the real needs of users of the system.

System functional requirements may be generated to define error checking and recovery facilities and features that provide protection against system failures. Non-functional requirements may be generated to define the required reliability and availability of the system.

Critical systems specifications' objective is to understand the risks faced by the system and generate dependability requirements to cope with them. The process of risk analysis consists of four steps:

1. **Risk identification.** Potential risks that might arise are identified. These are dependent on the environment in which the system is to be used. In safety-critical systems, the principal risks are hazards that can lead to an accident. Experienced engineers, working with domain experts and professional safety advisors, should identify system risks. Group working techniques such as brainstorming may be used to identify risks.

2. **Risk analysis and classification.** The risks are considered separately. Those that are potentially serious and not implausible are selected for further analysis. Risks can be categorised in three ways:
 - a. **Intolerable.** The system must be designed in such a way so that either the risk cannot arise or, if it does arise, it will not result in an accident. Intolerable risks are those that threaten human life or the financial stability of a business and which have a significant probability of occurrence.
 - b. As low as reasonably practical (**ALARP**). The system must be designed so that the probability of an accident arising because of the hazard is minimised, subject to other considerations such as cost and delivery. ALARP risks are those which have less serious consequences or which have a low probability of occurrence.
 - c. **Acceptable.** While the system designers should take all possible steps to reduce the probability of an 'acceptable' hazard arising, these should not increase costs, delivery time or other non-functional system attributes.
3. **Risk decomposition.** Each risk is analysed individually to discover potential root causes of that risk. Different techniques for risk decomposition exist. The one discussed in the book is Fault-tree analysis, where analyst puts hazard at the top and place different states which can lead to that hazard above. States can be linked with 'or' and 'and' symbols. Risks that require a combination of root causes are usually less probable than risks that can result from a single root cause.
4. **Risk reduction assessment.** Proposals for ways in which the identified risks may be reduced or eliminated are made. Three possible strategies of risk deduction that can be used are:
 - a. **Risk avoidance.** Designing the system in such a way that risk or hazard cannot arise.
 - b. **Risk detection and removal.** Designing the system in such a way that risks are detected and neutralised before they result in an accident.
 - c. **Damage limitation.** Designing the system in such a way that the consequences of an accident are minimised.

In the 1980s and 1990s, as computer control become widespread, the safety engineering community developed standards for safety critical systems specification and development. The process of safety specification and assurance is part of an overall safety life cycle that is defined in an international standard for safety management IEC 61508 (IEC, 1998).

Security and safety requirements have something in common; however, there are some differences between these types of requirements.

The conventional (non-computerised) approach to security analysis is based around the assets to be protected and their value to an organisation. The stages in this process are:

1. Asset identification and evaluation

2. Threat analysis and risk assessment
3. Threat assignment
4. Technology analysis
5. Security requirements specification

Reliability is a complex concept that should always be considered at the system rather than the individual component level. Component in the system are interdependent, so a failure in one component can be propagated through the system and affect the operation of other components. Three types of reliability should be taken in consideration – hardware reliability, software reliability and operator reliability.

My thoughts:

I think almost every system can be dangerous somehow, it's a nature of things that if something does something, this something can be done in other, more insecure or unsafe way. Especially, in such a complex structures as software systems. Engineers do their best to develop better, more secure and safe systems, but while increasing the size of system all the problems and risks increase, so all techniques of risk avoidance or reduction are in this infinite race between system size and safety. Previous chapters were all about important thing about functional part of a system, how system should be developed etc. This chapter is also about developing and designing the system, but it shows the importance of other side.

Chapter 10

The term *formal methods* is used to refer to any activities that rely on mathematical representations of software including formal system specification, specification analysis and proof, transformational development, and program verification. In the 1980s many software engineering researchers proposed that using formal development methods was the best way to improve software quality. They also predicted that by 21st century a large proportion of software would be developed using formal methods. However, this prediction has not come true.

Critical systems development usually involves a plan-based software process that is based on the waterfall model of development. Both the system requirements and the system design are expressed in detail and carefully analysed and checked before implementation begins. The involvement of the client decreases and the involvement of the contractor increases as more detail is added to the system specification. In the early stages of the process the specification should be 'customer-oriented'. Stages of software specification and its interface with the design process are (in order):

User Requirements Definitions -> System requirements specification -> Architectural design -> Formal specification -> High-level design.

Depending on the process used, specification problems discovered during formal analysis might influence changes to the requirements specification if this has not already been agreed.

Two fundamental approaches to formal specification have been used to write detailed specifications for industrial software systems:

1. An algebraic approach where the system is described in terms of operations and their relationships.
 2. A model-based approach where a model of the system is built using mathematical constructs such as sets and sequences, and the system operations are defined by how they modify the system state.
-

Large systems are usually decomposed into sub-systems that are developed independently. Sub-systems make use of other sub-systems, so an essential part of the specification process is to define sub-system interfaces.

The algebraic approach was originally designed for the definition of abstract data type interfaces. In an abstract data type, the type is defined by specifying the type operations rather than the type representation, so it is similar to an object class.

The process of developing a formal specification of a sub-system interface includes the following activities:

1. Specification structuring. Organise the informal interface specification into a set of abstract data types or object classes.
 2. Specification naming. Establish a name for each abstract type specification.
 3. Operation selection. Choose a set of operations for each specification based on the identified interface functionality.
 4. Informal operation specification. Write an informal specification of each operation.
 5. Syntax definition. Define the syntax of the operations and the parameters to each.
 6. Axiom definition. Define the semantics of the operations by describing what conditions are always true for different operation combinations.
-

The simple algebraic techniques can be used to describe interfaces where the object operations are independent of the object state. Therefore, the result of applying an operation should not depend on the results of previous operations. As structure increases in size, algebraic descriptions become increasingly difficult to understand.

Model-based specification is an approach to formal specification where the system specification is expressed as a system state model. System operations are defined by how they affect the state of the system model.

While developing a model-based specification, state variables and predicates should be defined.

My thoughts:

Maybe I just don't have any good experience, but for me a model-based approach seems like too complicated case, and algebraic approach sound more natural.

Chapter 11

Architectural design is the initial design process of identifying sub-systems and establishing a framework for sub-system control and communication. Using large-grain components improves performance, and using fine-grain components improves maintainability, so if both of these are important system requirements developers should find some compromise solution. There is an overlap between the process of requirements engineering and architectural design.

Sub-system design is an abstract decomposition of a system into large-grain components, each of which may be a substantial system in its own right. Block diagrams are often used to describe sub-system designs where each box in the diagram represents sub-system. Sub-systems can have their own sub-systems, in that case boxes are placed into boxes. Arrows mean that data and or control signals are passed from sub-system to sub-system in the direction of the arrows.

Architectural design is a creative process so many decisions are being made depending on requirements, specific rules for particular project and experience of an architect. Architectural models that may be developed may include:

- A static structural model that shows the sub-systems or components that are to be developed as separate units.
- A dynamic process model that shows how the system is organised into processes at run-time.
- An interface model that defines the services offered by each sub-system through its public interface.
- Relationship model that shows relationships, such as data flow, between the sub-systems.
- A distributed model that shows how sub-systems may be distributed across computers.

Different models may be used. Sub-systems making up a system exchange information so that they can work together effectively. There are two fundamental ways in which this can be done.

1. All shared data is held in a central database that can be accessed by all sub-systems. A system model based on a shared database is sometimes called a repository model.
2. Each sub-system maintains its own database. Data is interchanged with other sub-systems by passing messages to them.

The majority of systems that use large amount of data are organised around a shared database or repository. This model is suited to applications where data is generated by one sub-system and used by another.

The client-server architectural model is a system model where the system is organised as a set of services and associated servers and clients that access and use the services. Clients may have to know the names of the available servers and the services that they provide.

The layered model of an architecture (sometimes called an abstract machine model) organises a system into layers, each of which provide a set of services. Each layer can be thought of as an abstract machine whose machine language is defined by the services provided by the layer. This 'language' is used to implement the next level. This approach supports the incremental development of systems.

After an overall system organisation has been chosen, different approaches may be used to decompose sub-systems into modules. Two main strategies of doing this are:

1. Object-oriented decomposition where a system is decomposed into a set of communicating objects. Object call on the services offered by other objects. The object-oriented approach is well-known and convenient, but still has some disadvantages, for example, to use services, objects must explicitly reference the name and the interface of other objects. If an interface change is required to satisfy proposed system changes, the effect of that change on all users of the changed object must be evaluated.
2. Function-oriented pipelining where the system is decomposed into functional modules that accept input data and transformed into output data. Each processing step is implemented as a transform.

The models for structuring a system are concerned with how a system is decomposed into sub-systems. To work as a system, sub-systems must be controlled so that their services are delivered to the right place at the right time. There are two generic control styles that are used in software systems:

- Centralised control. One sub-system has overall responsibility for control and starts and stops other sub-systems. It may also devolve control to another sub-system but will expect to have this control responsibly returned to it.
- Even-based control. Rather than control information being embedded in a sub-system, each sub-system can respond to externally generated events. These events might come from other sub-systems or from the environment of the system.

Chapter 12

A distributed system is a system where the information processing is distributed over several computers rather than confined to a single machine. Advantages of distributed system are coming from several important characteristics of distributed systems. These advantages are:

1. Resource sharing
2. Openness
3. Concurrency
4. Scalability
5. Fault tolerance

However, there are (as usual) some disadvantages:

1. Complexity
2. Security
3. Manageability

4. Unpredictability

The design challenge is to create the software and hardware to provide desirable distributed system characteristics and, at the same time, minimise the problems that are inherent in these systems.

Two generic types of distributed systems are:

- **Client-server architectures.** The system may be thought of as a set of services that are provided to clients that make use of these services. Servers and clients are treated differently. The simplest client-server architecture is called a two-tier client-server architecture, where an application is organised as a server (or multiple servers) and a set of clients. The client might be **thin** (when all of the application processing and data management is carried out on the server, and the client is just presenting data) or **fat** (where client implements the application logic and interactions with user, and server is responsible for data management)
- **Distributed object architectures.** There is no distinction between servers and clients, and the system may be thought of as a set of interacting objects whose location is irrelevant. There is no distinction between a service provider and a user of these services. A more general approach to distributed system design is to remove the distinction between client and server and to design the system architecture as a distributed object architecture. Objects may be distributed across a number of computers on a network and communicate through middleware. This middleware is called an object request broker.

The simplest model of a distributed system is a multiprocessor system where the software system consists of a number of processes that may (but need not) execute on separate processors. This approach is widely used in large real-time systems.

Peer-to-peer (p2p) systems are decentralised systems where computations may be carried out by any node on the network and, in principle at least, no distinctions are made between clients and servers. The overall system is designed to take advantages of the computational power and storage available across a potentially big network of computers.

My thoughts:

Our Monopoly game is an example of client-server architecture with a thin client. We have one server which is responsible for all computations and data management, and client is only a GUI. This is the easiest way for us, because technically one program will do everything and another will only show that. It is also better in terms of security:

Chapter 13

There are many types of application system and, on the surface, they may seem to be very different. However, after examining the architectural organisation of applications many of these superficially dissimilar applications have much in common. There are four broad types of architectures discussed in the book:

1. **Data-processing applications.** These are data-driven applications, they process data in batches without explicit user interventions during the processing. The specific action

taken by the application depend on the data that it is processing. Applications of this type are commonly used in business, where similar operations are carried out on a large amount of data. The nature of data-processing systems where records or transactions are processed serially with no need to maintain state across transactions means that these systems are naturally function-oriented. Functions are components that do not maintain internal state information from one invocation to another.

2. **Transaction-processing applications.** These are database-centred applications that process user requests for information and that update the information in database. These are most common type of interactive business systems. They are organised in such a way that user actions can't interfere with each other and the integrity of the database is maintained. To simplify the management of different terminal communication protocols, large-scale transaction-processing systems may include middleware that communicates with all types of terminal, organises and serialises the data from terminals, and sends that data for processing. Implementations of information and resource management systems based on Internet protocols are now the norm.
3. **Event-processing systems.** This is a very large class of application where the actions of the system depend on interpreting events in the system's environment. These events might be the input of a command by a system user or a change in variables that are monitored by the system. Real-time systems also fall into this category. However, for real-time systems, events are not usually user interface events but events associated with sensors or actuators in the system.
4. **Language-processing systems.** These are systems where the user's intentions are expressed in a formal language (such as Java). The language-processing system processes this language into some internal format and then interprets this internal representation. The best known language-processing systems are compilers, which translate high-level language programs to machine code. In the very abstract way the language-processing system's process can be described as: instructions go to the translator, which checks syntax and semantics and then generate some abstract instructions for interpreter. Interpreter takes those instructions along with data, executes the program and gives the output. Other components might also be included that transform the syntax tree to improve efficiency and remove redundancy from the generated machine code.

Chapter 14

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between them.

Object-oriented design is one phase of the development process:

- Object-oriented analysis is concerned with developing an object-oriented model of the application domain. The objects in that model reflect the entities and operations associated with the problem to be solved.
- Object-oriented design with developing an object-oriented model of a software system to implement the identified requirements.

- Object-oriented programming is concerned with realising a software design using an object-oriented programming language, such as Java.

An object is an entry that has a state and a defined set of operations that operate on that state. The state is represented as a set of object attributes. Objects created according to an object class definition.

In such a language as Java it is easy to take an object-oriented design and produce an implementation where the objects are concurrent processes. There are two kinds of concurrent object implementation:

1. Servers where the object is realised as a parallel process with methods corresponding to the defined object operations.
2. Active objects where the state of the object may be changed by internal operations executing within the object itself.

The general process that is described in the book has a number of stages:

1. Understand and define the context and the modes of use of the system.
2. Design the system architecture
3. Identify the principal objects in the system.
4. Develop design models
5. Specify object interfaces.

My thoughts:

The concept of object-oriented design seemed really unnatural for me five or six years ago when I used to code with Visual Basic and Delphi. I've seen no sense in making things so complicated and abstract rather than just write the program you need using procedures, jumps etc. Today it's the other way around – I don't see any other way of programming something bigger than bash script.

Chapter 15

Computers are used to control a wide range of systems from simple domestic machines to entire manufacturing plants. These computers interact directly with hardware devices. The software in these systems is embedded real-time software that must react to events generated by the hardware and issue control signals in response to these events. A good definition from author of the book is the following:

A real time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced. A soft real-time system is a system whose operation is degraded if results are not produced according to the specific timing requirements. A hard real-time system is a system whose operation is incorrect if results are not produced according to the timing specification.

One way of looking at a real-time system is as an input/response system. For a specific input system responds in a particular way. These inputs (or stimuli) fall into two classes:

1. **Periodic stimuli.** These occur at predictable time intervals.

2. **Aperiodic stimuli.** These occur irregularly.
-

Because real-time systems must meet their timing constraints, sometimes it is impossible to use object-oriented development for hard real-time systems. Object-oriented development involves hiding data and accessing attributes values through operations defined with the object, which means there is a significant performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations.

A state model of a system assumes that, at any time, the system is in one of a number of possible states. When a stimulus is received, this may cause a transition to a different state.

A real-time operating system manages processes and resource allocation in a real-time. There are many RTOS products available, from very small, simple systems for consumer devices to complex systems for cell phones and mobile devices and operating systems specially designed for process control and telecommunications. RTOS usually includes:

- **A real-time clock.** Provides information to schedule processes periodically.
 - **An interrupt handler.** Manages aperiodic requests for service.
 - **A scheduler.** Responsible for examining the processes that can be executed and choosing one of these for execution.
 - **A resource manager.** Allocates appropriate memory and processor resources for a scheduled process.
 - **A dispatcher.** This component is responsible for starting the execution of a process.
-

Real time operating system's actions required to start a process looks like this: Scheduler chooses process for execution based on process priority, the expected execution time and the deadlines of the ready processes; resource manager allocates memory and processor; dispatcher starts execution on available processor.

Chapter 16

User interface of a software product is the “face” of it. Many “user errors” are caused by the fact that user interfaces do not consider the capabilities of real users and their working environment. A poorly designed interface means that users will probably be unable to access some of the system features, will make mistakes and will feel that the system hinders rather than helps them in achieving whatever they are using system for.

While designing user interface, some physical and mental capabilities of the people who will use the software should be taken into account. Some of them are:

1. People have a limited short-term memory.
2. We all make mistakes, especially when we have to handle too much information or are under stress. Too much system warnings, alarms etc. can annoy stressed people thus increase the chances that they will make more operational mistakes.

3. We have a diverse range of physical capabilities. Some people see and hear better than other, some people are color-blind, etc.
4. We have different interaction preferences. Some people like to work with pictures, others with text.

These human factors are the basic for the design principles.

The principle of **user familiarity** suggests that users should not be forced to adapt to an interface because it is convenient to implement.

The principle of **user interface consistency** means that system commands and menus should have the same format, parameters should be passed to all commands in the same way, and command punctuation should be the similar. However, full consistency is neither possible nor desirable.

The principle of **minimal surprise** is appropriate because people get very irritated when a system behaves in an unexpected way.

The principle of **recoverability** is important because users inevitably make mistakes when using a system. The interface design can minimise these mistakes, but mistakes can never be completely eliminated. Consequently, system should include some tools allowing users recover from their mistakes. These can be of three kinds:

- Confirmation of destructive actions. To avoid accidental destructions or to explain consequences for user.
- The provision of an undo facility. Undo restores the system to a state before the action occurred. Multiple levels of undo are useful because users don't always recognise that a mistake has been made. (*Why mspaint has only three levels of undo, or do they think that people who use mspaint are only pros and don't do too much mistakes??!*)
- Checkpointing. It involves saving the state of a system at periodic intervals and allowing the system to restart from the last checkpoint. Similar to simple backup system for data.

A related principle is the principle of **user assistance**. Interfaces should have built-in user assistance or help facilities.

The overall user interface design process in general concerned with two main questions: **how should the user interact with the system** and **how should information from the system be presented to the user?**

User can interact with computer system in a number of ways, most common of them are:

1. Direct manipulation. The user interacts directly with objects on the screen. It usually involves a pointing device, e.g. a mouse or a finger on a touch screen (like on your white 16 gig iPhone).
2. Menu selection. The user selects a command from a list of possibilities (a menu).
3. Form fill-in. The user fills in the fields of a form. Some fields may have associated menus, and the form may have action to be initiated.
4. Command language. User enters a special command and associated parameters to instruct the system what to do (bash!)

5. Natural language. The user issues a command in natural language. This is usually a front-end to a command language. (*I also think that this includes voice commands*)

Presentation of information by a computer system depends on user of the system. Also, the issue of colour in presentation is important. The most important key guidelines for the effective use of colour are:

1. Limit the number of colours employed and be conservative how these are used.
2. Use colour change to show a change in system status.
3. Use colour coding to support the task users are trying to perform.
4. Use colour coding in a thoughtful and consistent way.
5. Be careful about color pairings.

The user interface design process includes sub-processes concerned with user analysis, interface prototyping and interface evaluation. The aim of user analysis is to sensitise designers to the ways in which users actually work. Different techniques (task analysis, interviewing, observation) are used during user analysis.

User interface prototype development should be a staged process with early prototypes based on paper versions of the interface that, after initial evaluation and feedback, are used as a basis for automated prototypes. The goals of user interface evaluation are to obtain feedback on how a UI design can be improved to assess whether an interface meets its usability requirements.

My thoughts:

As I was reading this chapter, I was thinking about how serious and professional Apple's user interface designers are. Every point of the stuff discussed in this chapter was a serious for them. Like the principle of user consistency: in Mac OS user can always be sure, that the program he never used before will have a menu line at the top of the screen. ALWAYS!

*Actually, the iPhone is one of the few products where I don't even want to customize user interface, because it feels like someone **did think**.*

In general, I think of user interface as a very important and really interesting field in software engineering. Right now I'm working on GUI for our project, so this chapter was pretty useful for me.

Chapter 17

Rapid software development processes are designed to produce useful software quickly. Generally, they are iterative processes where specification, design, development and testing are interleaved. The software is not developed and deployed in its entirety, but in a series of increments, with each increment including new system functionality. The two main advantages to adopting an incremental approach to software development are:

1. Accelerated delivery of customer services. Early increments of the system can deliver high-priority functionality.

2. User engagement with the system. Users of the system have to be involved in the incremental development process because they have to provide feedback to the development team on delivery increments.

However, as usual, there are some major difficulties with iterative development and incremental delivery are:

1. Management problems. Incrementally developed systems change so quickly that it is not cost-effective to produce lots of system documentation.
2. Contractual problems. When there is no system specification, it may be difficult to design a contract for the system development.
3. Validation problems. In a specification-based process, verification and validation are geared towards demonstrating that the system meets its specifications.
4. Maintenance problems. Continual change tends to corrupt the structure of any software system. This means that anyone apart from the original developers may find the software difficult to understand.

Agile methods are iterative development methods that focus on incremental specification, design and system implementation. They involve the customer directly in the development process. Reducing development overhead can make faster software development possible. Extreme programming (XP) is a well-known agile method that integrates a range of good programming practices such as systematic testing, continuous software improvement and customer participation in the development team.

In an XP process, customers are intimately involved in specifying and prioritizing system requirements. The requirements are not specified as lists of required system functions. Rather, the system customer is part of the development team and discusses scenarios with other team members.

Rapid application development involves using development environments that include powerful tools to support system production. These includes database programming languages, form and report generators, and links to office applications.

Throw away prototyping is an iterative development process where a prototype system is used to explore the requirements and design options. This prototype is not intended for deployment by the system customer.

My thoughts:

Rapid software development seems interesting for me, because it somehow related to our project. Even though we had quite a lot of time and a case study (which almost means we had a set of requirements ready to be organized and used), extreme programming sounds like a good description of last two weeks before iteration 2 😊

Anyway, this type of software development is interesting, but I'd prefer a regular, no-rush and planned process, it is much less stress and more creativity (at least for the type of person I am).

Chapter 18

Reuse-based software engineering is an approach to development that tries to maximize the reuse of existing software. The software units that are reused may be of radically different sizes. There may be an application system reuse, different particular component reuse or object and function reuse. A complementary form of reuse is concept reuse, where rather than reuse a component, the reused entity is more abstract and is designed to be configured and adapted for a range of situations.

An obvious advantage of software reuse is that overall development costs should be reduced. Fewer software components need be specified, designed, implemented and validated. However, there are also costs and problems associated with software reuse. There is a significant cost associated with understanding whether a component is suitable for reuse in a particular situation and in testing that component to ensure compatibility.

Design patterns are high-level abstractions that document successful design solutions. They are fundamental to design reuse in object-oriented development. A pattern description should include a pattern name, a problem and solution description, and a statement of the results and trade-offs of using the pattern. A huge number of published patterns are now available covering a range of application domains and languages. The use of patterns is an effective form of reuse.

Program generators are an alternative approach to concept reuse where the reusable concepts are embedded in a generator system. The designer specifies the abstractions required using a domain-specific language, and an executable program is generated.

Application frameworks are collections of concrete and abstract objects that are designed to be reused through specialization and addition of new objects. Three classes of framework are:

- System infrastructure frameworks. These frameworks support the development of system infrastructures such as communications, user interfaces and compilers.
 - Middleware integration frameworks. These consist of a set of standards and associated object classes that support component communication and information exchange.
 - Enterprise application frameworks. These are concerned with specific application domains such as telecommunications or financial systems.
-

A commercial-off-the-shelf product is a software system that can be used without change by its buyer. COTS product reuse is concerned with the reuse of large-scale systems. These provide a lot of functionality, and their reuse can radically reduce costs and development time. Potential problem with COTS-based reuse include lack of control over functionality and

performance, lack of control over system evolution, the need for support from external vendors and difficulties in ensuring that systems can interoperate.

One of the most effective approaches to reuse is creating software product lines or application families. A product line is a set of applications with a common application-specific architecture. Various types of specialization of a software product line may be developed:

1. Platform specialization.
2. Environment specialization
3. Functional specialization
4. Process specialization

My thoughts:

While working on our project we touched only one aspect of software reuse – program generators. Rose Real Time is a good example of program generator.

When we just started I was thinking about Rose RT as a weird way to develop a system, it was hard to understand how learning a new environment and using it may be useful or may lead to faster development, but today I see all advantages of this approach. We don't have to think about building a server, about network routines, about designing a system in a good way with straight C and Java, we don't even have to think about choosing the way of connecting java and c code.

So now I think of software reuse as actually useful thing that helps us reduce the development time.

Chapter 19

Reuse-based software engineering is becoming the main development approach for business and commercial systems. Component-based software engineering (CBSE) emerged in the late 1990s as a reuse-based approach to software systems development. Single object classes were too detailed and specific, and often had to be bound with an application at compile-time. CBSE is the process of defining, implementing and integrating or composing loosely coupled independent components into systems. The essentials of component-based software engineering are:

1. Independent components that are completely specified by their interfaces. There should be a clear separation between the component interface and its implementation so that one implementation of a component can be replaced by another without changing the system.
2. Component standards that facilitate the integration of components. These standards are embodied in a component model and define, at very minimum, how component interfaces should be specific and how components communicate.
3. Middleware that provides software support for component integration. Middleware such as CORBA handles low-level issues efficiently and allows developers to focus on application-related problems.
4. A development process that is geared to component-based software engineering.

As always, some problems with using this approach remain:

1. Component trustworthiness. Components are black-box program units, and the source code of the component may not be available to component users.
2. Component certification. This issue is closely related to trustworthiness, it is the issue of certification.
3. Emergent property prediction. Because components are opaque, predicting their emergent properties is particularly difficult.
4. Requirements trade-offs. Developers usually have to make trade-offs between ideal requirements and available components in the system specification and design process.

A component model defines a set of standards for components, including interface standards, usage standards and deployment standards. The implementation of the component model provides a set of horizontal services that may be used by all components.

Component composition is the process of writing components together to create a system.