Vi hjälps åt att svara på alla frågor.

Regler: Sätt ditt namn innan svaret. Om någon redan svarat och du vill lägga till något, sätt ditt namn framför det också.

Ta inte bort eller redigera någon annans svar, utan lämna istället kommentar.

(Hannes) OBS!!! Svaren till frågorna står i sin helhet på engelska uteslutande i boken, det räcker alltså inte att kolla lite snabbt i föreläsningarna för att kunna svara på en fråga.

Om ni inte har boken, Software Engineering ninth ed, kontakta Hannes.

Question Bank

Software Engineering

IV1300

2014(wat?)

Sista frågan i det här dokumentet kommer att ställas bara på första ordinarietentan.

Basics of Software Engineering

Question: **Definition of software** (1 point) (based on Mira's material, Chapter 1.3)
What is software? (1 point)

(Hannes) En väldigt enkel modell av mjukvara är att den endast består av Object Code och Source Code, men mjukvara är mycket mer än bara koden; den består av kravspecifikationer, designspecifikationer, implementation, testspecifikationer, användarmanualer (inkl kommentarer i koden), konfigurationsfiler, tillhörande websidor, rutiner för att sätta upp och underhålla mjukvarusystemet, instruktioner för att sätta upp och underhålla mjukvarusystemet och instruktioner för felhantering samt mycket mer.

Question: Definition of software engineering (2,5 points)
What is the definition of software engineering (0,5 points) and what two key phrases does this definition include? (2 points)

(Hannes) **Mjukvaruutveckling** (Software Engineering) är en ingenjörsdisciplin som innefattar alla aspekter av mjukvaruproduktion i allt ifrån de tidiga delarna av systemspecifikationen fram till att underhålla systemet efter att det har börjat användas. (0,5)

(Hannes) De två nyckelfraserna är:
1, **Ingenjörsdisciplin** (Engineering discipline) - Ingenjören får saker att fungera. De applicerar teorier, metoder och verktyg där det är lämpligt. Dock endast med omsorg och försöker hitta lösningar på problem även fast det inte finns några kända applicerbara teorier eller metoder. Ingenjörer förstår också att de måste jobba under organisatoriska och ekonomiska begränsningar och försöker då hitta lösningar inom dessa begränsningar.

2, **Alla aspekter av mjukvaruproduktion** (All aspects of software production) - Mjukvaruutveckling handlar inte bara om tekniska processer. Det inkluderar även aktiviteter såsom mjukvaruprojektleding och utveckling av verktyg, metoder och teorier som behövs för att stödja och underhålla mjukvaruproduktionen.
(2p)

Question: Relationship to other disciplines (1 point)

How is software engineering related to computer science and system engineering?

(Hannes)

Relation till Datavetenskap *(Computer Science)*:

Datavetenskap handlar om teorier och metoder som ligger till grund för datorer och mjukvarusystem, medan mjukvaruutveckling (software engineering) handlar om praktiska problem inom producerandet av mjukvara. För att kunna arbeta med mjukvaruutveckling behöver man kunna delar av datavetenskap, precis som en elektriker behöver kunna delar av fysikvetenskap.

Teorier inom datavetenskap är dock oftast bara applicerbara på relativt små program, då dessa teorier ofta är svåra att tillämpa på stora, komplexa system som kräver en mjukvarulösning.

Relation till systemteknik *(System Engineering)*:

Systemteknik innefattar alla aspekter av framtagning och utveckling av komplexa system där även mjukvara spelar en stor roll. Systemteknik handlar om utplacering/installation (deployment) av hårdvara, policy- och processdesign, system-deployment, och mjukvaruutveckling (software engineering). Systemtekniker är involverade i att specificera systemet, den allmänna arkitekturen och integrering av olika delar för att få ett färdigt system. Systemteknikerna är mindre involverade i de olika specifika delarna av systemet, t ex mjukvara eller hårdvara.

(1p)

Question: Diversity of software application (3 points)

One of the most significant factor in determining software engineering methods is the type of application that is being developed. Sommerville lists eight such types. List (1,5 points) and explain (1,5 points) six of them.

(Hannes) Här kommer alla 8 typer:

1. ***Fristående applikationer*** *(Stand-alone applications)* - Applikationer/program som finns på ett fristående system, typ PC eller smartphone, och som har all den funktionalitet de behöver i programmet för att fungera. De behöver t ex alltså inte internet för att fungera. Exempel: Photoshop, Word, CAD.

2. ***Interaktiva transaktionsbaserade applikationer*** *(Interactive transaction-based applications)* - Tänk: Klient-Server-baserade program. Program som körs på en server eller i molnet, serversidan, och används på användares PC eller liknande, klientsidan, genom webläsare eller ett speciellt gränssnitt, typ en bankomat.

3. ***Inbyggda kontrollsystem*** *(Embedded Control Systems)* - Inbyggd mjukvara som kontrollerar specifik hårdvara. Det kan vara mjukvaran som kontrollerar en mikrovågsugn eller låssystemet i en bil. Det kan också vara mjukvaran som kontrollerar en smartphone och är då ett helt operativsystem.

4. **Batchbehandlingssystem** *(Batch processing systems)* - System oftast för företag som behandlar stora mängder data, hos t ex banker, rösträkningskontor och försäkringsbolag. Stora mängder input av liknande slag som producerar motsvarande output i stora batchar.

5. **Underhållningssystem** *(Entertainment systems)* - System som är primärt utvecklade för att underhålla användaren. Det är kvaliteten på användarinteraktionen som är den viktigaste egenskapen.

6. **System för modellering och simulering** *(Systems for modeling and simulation)* - System utvecklade av forskare och ingenjörer som modellerar fysiska processer eller situationer som inkluderar många olika objekt som interagerar med varandra. Dessa system har tunga beräkningar och kräver högprestanda parallella system.

7. **Datainsamlingssystem** *(Data collection systems)* System som hämtar data, genom sensorer, från omgivningen och skickar vidare denna data till ett behandlingssystem. T ex, väderballonger eller fartkameror. Ett annat exempel är det nya hypade “Internet of things”, där vardagsprylar, typ som kylskåpet, linser eller vantar är uppkopplade mot internet.

8. **System av system** *(System of systems)* System som består utav flera mjukvarusystem.

Question: Relationship to other disciplines (3 points) s. 10 i boken.

There is no universal software engineering method or technique that is applicable for all different types of software. However, there are three general issues that affect different types of software. These are *heterogeneity*, *business and social change* and *security and trust*. Explain in what way these issues provide challenge to software developers?

(Hannes)

1. **Heterogenitet** *(heterogeneity)* - I allt större utsträckning måste mjukvarusystem fungera som distribuerade system över nätverk, och både på datorer och mobiltelefoner. Man är även tvungen att utveckla nya system så att de är kompatibla med äldre system och hårdvara som kanske bygger på ett utdaterat programmeringsspråk. Utmaningen är alltså att utveckla system som är robusta och klarar ovan nämnda utmaningar, vilket kallas heterogenitet, (=likvärdigt).

2. **Ändringar inom företag och samhälle** *(business and social change)* - Företag och samhällen ändras väldigt snabbt medan utvecklingsländer kommer ikapp västvärlden och nya teknologier görs tillgängliga. Alla måste kunna ändra och utveckla den redan existerande mjukvaran i ett snabbare tempo än idag. Många traditionella mjukvaruutvecklingsmetoder är tidskrävande och leverans av nya system tar ofta längre tid än planerat. Mjukvaruutvecklingsmetoderna måste utvecklas så att tiden det tar att leverera ett system är mindre och mer flexibelt. (Tänk: Banker som fortfarande har system som togs fram för 40 år sedan, och som är för kostsamma att

byta ut eller uppgradera).

3. **Säkerhet och förtroende** *(security and trust)* - Medan mjukvaran blir en större del av våra liv måste vi kunna lita på den. Detta gäller speciellt för fjärrsystem (remote systems) som t ex inloggning till en bank genom en webläsare. Vi, ingenjörer, måste se till att skadliga användare och program inte kommer åt vår mjukvara och att informationssäkerhet bibehålls. (Tänk: Hälsoappar som håller känsliga uppgifter).

(3p)

Question: Types of software products (2 points)
There are two types of software products.
What are they? (0,5)
Provide an example of each type. (0,5)
Who is in the control of the specification of each product type? Motivate? (1 point)

(Hannes)
De två typerna: **Allmänna produkter** *(generic products)* och **Specialgjorda produkter** *(customized products).* (0,5p)

Expempel på allmänna produkter: Alla typer av mjukvara som är framtagna av en organisation eller företag som är tillverkade för massproduktion och sålda på marknaden. T ex olika program till en PC eller TV-spel.

Exemplel på specialgjorda produkter: System som är framtagna till en viss kund. Det kan vara ett system som är framtaget för att stödja produktionen hos ett visst företag eller systemet som ansvarar för flygkoordinationen på en viss flygplats.
(0.5p)

Vem som kontrollerar och bestämmer specifikationen för de två typerna: När det gäller allmänna produkter är det företaget som tillverkar produkten som bestämmer specifikationen, alltså hur produkten ska vara och fungera. När det gäller specialgjorda produkter är det å andra sidan kunden som bestämmer specifikationen för produkten. Numera har dock dessa skillnader suddats ut då fler försöker skapa produkter som kunden själv kan anpassa efter sina behov.
(1p)

Question: Professional and ethical responsibilities (4 points)
Like other engineers, software engineers must accept that their job involves wider responsibilities than simply the application of technical skills, the professional and ethical responsibilities. Some of the professional and ethical responsibilities are:

- **Confidentiality**
- **Competence**
- **Intellectual property rights**
- **Computer misuse.**

Describe briefly in what way the software engineers may misuse their responsibilities.

(Hannes)

1. **Konfidentialitet** *(Confidentialty)* - Man bör respektera konfidentialiteten hos anställda och klienter även om inget formellt sekretessavtal har skrivits under. Mjukvaruingenjörer kan missbruka detta genom att utveckla system som inte inte har tillräckligt hög sekretess, t ex hälsoappar som sparar och skickar känslig patientinformation okrypterat.

2. **Kompetens** *(Competence)* - Du bör inte misstolka eller överskatta din kompetens och heller inte ta på dig arbete som du vet ligger ovanför din kompetens. Mjukvaruutvecklare, precis som vilka anställda som helst, tar gärna på sig svåra uppdrag för att visa vad de går för, fast de egentligen inte kommer klara av det. Detta kan leda till förödande konsekvenser för produkter som inte fungerar som de ska och projekt som inte blir klara i tid.

3. **Immateriella rättigheter** *(Intellectual propery rights)* - Du ska vara medveten om de lokala reglerna och stadgarna för patent och copyright där du jobbar. Du bör se till att dina och kollegors immaterialrättsliga rättigheter hemlighålls.
   En mjukvaruutvecklare kanske publicerar kod eller design av ett system som är en företagshemlighet eller som en viss kollega har patent på.

4. **Datormissbruk** *(computer misuse)* - Du ska inte använda dina tekniska färdigheter till att missbruka andras datorer. Det kan vara alltifrån att spela spel på jobbdatorn till att använda jobbdatorn som bas för virusspridning eller olaglig aktivitet.
   (4p)

Organizational Levels

Question: **Organizational levels and strategies** (6 points) based on Mira's material in Chapter 5

Give an account of organizational levels and the strategies that apply on each level.

(Hannes)

1. **Affärsstrategiska nivån** *(Business strategic level)* - Strategier som man ansvarar för i den här nivån: Affärs-, kund-. produkt- och andra strategier. Alla dessa strategier består utav en vision (långsiktiga), mission (kortsiktiga), mål (väldigt kortsiktiga), värden, planer, beslut för strategin samt kriterier för framgång.
   Man ser även till att underhålla och skapa dessa strategier.

2. **Taktiska nivån** *(tactical level)* - Strategier som man ansvarar för i den här nivån: Produktframtagnings-, processlivscykel-, teknik-, support-, SOA- (service-oriented architecture) strategier.

3. **Implementationsnivån** *(implementation level)* - Står ingenstans om att man måste ansvara för några strategier i den här nivån. Det man hursomhelst ansvarar för är utveckling, vidareutveckling, underhåll och testning (av produkter/mjukvara).

Software Processes

Question: Process versus process model (1 point)
What is a process and process model? (1 point)

(Hannes) En process är er rad sammansatta aktiviteter som utförs för ett visst resultat, där ordningen mellan aktiviteterna är eller inte är betydande.

En processmodell är en abstrakt representation av av en process. Processmodeller kan vara framtagna ur olika synvinklar och visar de involverade aktiviteterna, artefakterna som används, begränsningar, och medverkande personers roller till de olika aktiviteterna i processen. Processmodellen kan representeras som flytande text, diagram, punktlista och otaliga andra sätt. Processmodellen kan också innehålla för- och efterhandsvillkor som listar vad som är sant respektive falskt före och efter processen. (1p)

Question: Reasons for modelling processes (4 points)
Lari Lawrence Pfleeger (see Mira's slides) lists reasons for modelling processes. List and explain them.

(Hannes)
Orsaker till att använda processmodeller:

- För att få en allmän förståelse för processen.
- För att få fram en konsekvent och strukturerad mängd aktiviteter.
- För att leda våra handlingar (för att förstå vad som ska göras).
- För att få oss att undersöka, förstå, kontrollera och förbättra aktiviteterna i processen.
  - Vi kan göra det hela effektivare genom att hitta inkonsekventa (konstiga, avstickande), överflödiga och försummade delar av processen.
  - Vi kan utvärdera hur nödvändiga olika aktiviteter av processen är genom att titta på målet med processen.
  - Processmodeller får oss att ta till vara på de erfarenheter vi har (fått) och så att vi kan föra dem vidare.

Question: Process Model (4 points)
Give an account of the waterfall model.

Vattenfallsmodellen separerar processaktiviteterna: Kravspecifikation, systemdesign, mjukvaruuutveckling, validering (testning), och vidareutveckling, underhåll osv i olika faser där den ena fasen föregår den andra i en viss ordning och faserna måste på så sätt planeras noga i förväg, och är på så vis ett exempel på en plandriven process.

Inom systemutveckling kan vattenfallsmodellen delas in i dessa delar:

1. *Analys och definition av system- och mjukvarukrav.* - Systemets tjänster, begränsningar och mål etableras i samarbete med slutanvändare (kunden). Dessa krav studeras i detalj och blir sedan kravspecifikationen.

2. *System- och mjukvarudesign* - I designprocessen samlas kraven in för att skapa en övergripande arkitektur till hårdvaru- och mjukvarusystemen.

3. *Implementation och enhetstestning* - Mjukvarudesignen har nu materialiserats i olika delprogram. Under enhetstestningen verifierar man att dessa delar möter kravspecifikationen.

4. *Integrations- och systemtestning* - De olika programdelarna, enheterna, kopplas samman och testas som ett helt system för att se att man möter kravspecifikationen. Efter testerna levereras produkten till kunden.

5. *Drift och underhåll* - Det här är vanligtvis den längsta fasen i processen. Systemet är nu installerat och används hos kunden. Underhåll handlar om att rätta till fel som inte hittades i tidigare faser. Man utvecklar systemet då nya krav dyker upp.

Fortsättning på engelska:

In principle, the result of each phase is one or more documents that are approved (‘signed off’). The following phase should not start until the previous phase has finished. In practice, 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 is not a simple linear model but involves feedback from one phase to another. Documents produced in each phase may then have to be modified to reflect the changes made.

Because of the costs of producing and approving documents, iterations can be costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored, or programmed around. This premature freezing of requirements may mean that the system won’t do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

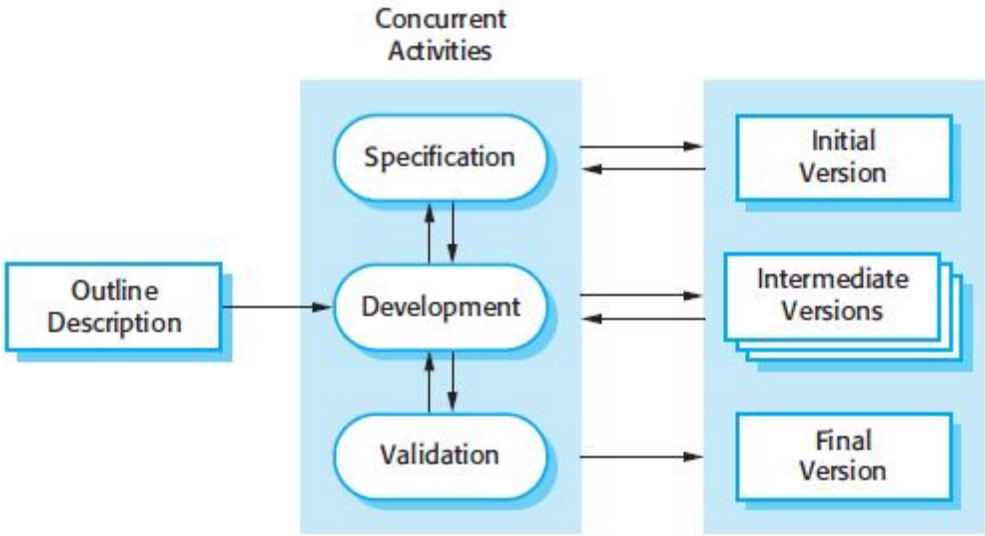
During the final life cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

The waterfall model is consistent with other engineering process models and documentation is produced at each phase. This makes the process visible so managers can

monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.
In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development.
However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.
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. Based on the assumption that your mathematical transformations are correct, you can therefore make a strong argument that a program generated in this way is consistent with its specification.
Formal development processes, such as that based on the B method (Schneider, 2001; Wordsworth, 1996) are particularly suited to the development of systems that have stringent safety, reliability, or security requirements. The formal approach simplifies the production of a safety or security case. This demonstrates to customers or regulators that the system actually meets its safety or security requirements.
Processes based on formal transformations are generally only used in the development of safety-critical or security-critical systems. They require specialized expertise. For the majority of systems this process does not offer significant costbenefits over other approaches to system development.

Question: Process Model (4 points)
Give an account of the incremental model.

(Hannes, Från boken)
Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed (Figure 2.2). Specification, development, and



validation activities are interleaved rather than separate, with rapid feedback across activities.

Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems. 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, backtracking 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 customer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer 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 for later increments.

Incremental development has three important benefits, compared to the waterfall model:

1. The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much 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. More rapid 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.

Incremental development in some form is now the most common approach for the development of application systems. This approach can be either plan-driven, agile, or, more usually, a mixture of these approaches. In a plan-driven approach, the system 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.
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. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

The problems of incremental development 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 responsibilities 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.
You can develop a system incrementally and expose it to customers for comment, without actually delivering it and deploying it in the customer's environment. Incremental delivery and deployment means that the software is used in real, operational processes. This is not always possible as experimenting with new software can disrupt normal business processes. I discuss the advantages and disadvantages of incremental delivery in Section 2.3.2.

Question: Process Model (4 points)
Give an account of the reuse-oriented model.

(Hannes, direkt från boken)

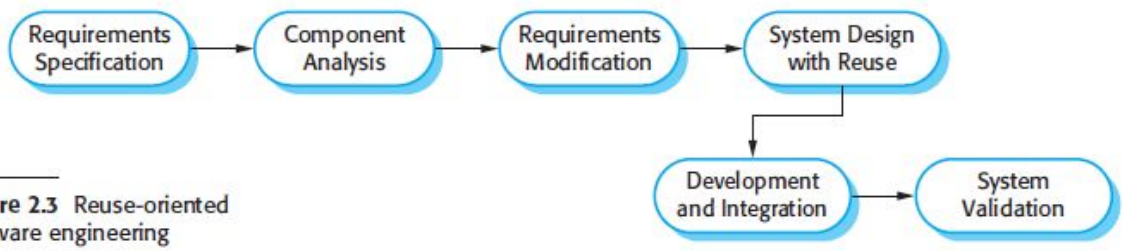


Figure 2.3 Reuse-oriented software engineering

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incorporate them into their system.

This informal reuse takes place irrespective of the development process that is used. However, in the 21st century, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a large base of reusable software components and an integrating framework for the composition of these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as word processing or a spreadsheet.

A general process model for reuse-based development is shown in Figure 2.3. Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuseoriented process are different. These stages are:

1. Component analysis. Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.
2. Requirements modification. During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.
3. System design with reuse. During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater for this. Some new software may have to be designed if reusable components are not available.
4. Development and integration. Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

There are three types of software component that may be used in a reuse-oriented process:

1. Web services that are developed according to service standards and which are available for remote invocation.
2. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
3. Stand-alone software systems that are configured for use in a particular environment.

Reuse-oriented software engineering 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, requirements compromises are inevitable 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.

Software reuse is very important and I have dedicated several chapters in the third part of the book to this topic. General issues of software reuse and COTS reuse are covered in Chapter 16, component-based software engineering in Chapters 17 and 18, and service-oriented systems in Chapter 19.

Question: Process Phases (4 points)

There are many different process models, but all must include four fundamental software engineering activities. These are:

1. Software specification
2. Software design and implementation
3. Software validation
4. Software evolution.

Give an account of software specification phase and its inclusive requirements engineering process.

(Från boken)

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. Software developers use a variety of different software tools in their work. Tools are particularly useful for supporting the editing of different types of document and for managing the immense volume of detailed information that is generated in a large software project.

The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved. How these activities are carried out depends on the type of software, people, and organizational structures involved. In extreme programming, for example, specifications are written on cards. Tests are executable and developed before the program itself. Evolution may involve substantial system restructuring or refactoring.

Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation.

The requirements engineering process (Figure 2.4) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

There are four main activities in the requirements engineering process:

1. Feasibility study. An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

2. Requirements elicitation and analysis. This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development

of one or more system models and prototypes. These help you understand the system to be specified.

3. Requirements specification. Requirements specification is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.

4. Requirements validation. This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.
Of course, the activities in the requirements process are not simply carried out in a strict sequence. Requirements analysis continues during definition and specification and new requirements come to light throughout the process. Therefore, the activities of analysis, definition, and specification are interleaved. In agile methods, such as extreme programming, requirements are developed incrementally according to user priorities and the elicitation of requirements comes from users who are part of the development team.

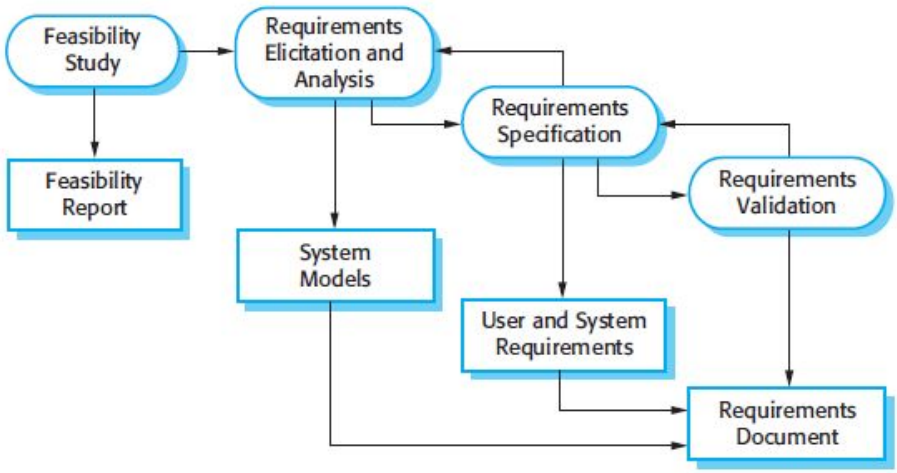


Figure 2.4 The requirements engineering process

Software design and implementation
The implementation stage of software development is the process of converting a system specification into an executable system. It always involves processes of software design and programming but, if an incremental approach to development is used, may also involve refinement of the software specification.
A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design iteratively. They add formality and detail as they develop their design with constant backtracking to correct earlier designs.
Figure 2.5 is an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.

The diagram suggests that the stages of the design process are sequential. In fact, design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.

Most software interfaces with other software systems. These include the operating system, database, middleware, and other application systems. These make up the 'software platform', the environment in which the software will execute. Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with the software's environment. The requirements specification is a description of the functionality the software must provide and its performance and dependability requirements. If the system is to process existing data, then the description of that data may be included in the platform specification; otherwise, the data description must be an input to the design process so that the system data organization to be defined.

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require timing design but may not include a database so there is no database design involved. Figure 2.5 shows four activities that may be part of the design process for information systems:

1. Architectural design, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
2. Interface design, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
3. Component design, where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
4. Database design, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

These activities lead to a set of design outputs, which are also shown in Figure 2.5. The detail and representation of these vary considerably. For critical systems, detailed design documents setting out precise and accurate descriptions of the system must be produced. If a model-driven approach is used, these outputs may mostly be diagrams. Where agile methods of development are used, the outputs of the design process may not be separate specification documents but may be represented in the code of the program.

Structured methods for design were developed in the 1970s and 1980s and were the precursor to the UML and object-oriented design (Budgen, 2003). They rely on producing graphical models of the system and, in many cases, automatically generating code from these models. Model-driven development (MDD) or model-driven engineering (Schmidt, 2006), where models of the software are created at different levels of abstraction, is an evolution of structured methods. In MDD, there is greater emphasis on architectural models with a separation between abstract implementationindependent models and implementation-specific models. The models are

developed in sufficient detail so that the executable system can be generated from them. I discuss this approach to development in Chapter 5.

The development of a program to implement the system follows naturally from the system design processes. Although some classes of program, such as safety-critical systems, are usually designed in detail before any implementation begins, it is more common for the later stages of design and program development to be interleaved. Software development tools may be used to generate a skeleton program from a design. This includes code to define and implement interfaces, and, in many cases, the developer need only add details of the operation of each program component.

Programming is a personal activity and there is no general process that is usually followed. Some programmers start with components that they understand, develop these, and then move on to less-understood components. Others take the opposite approach, leaving familiar components till last because they know how to develop them. Some developers like to define data early in the process then use this to drive the program development; others leave data unspecified for as long as possible.

Normally, programmers carry out some testing of the code they have developed. This often reveals program defects that must be removed from the program. This is called debugging. Defect testing and debugging are different processes. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects. When you are debugging, you have to generate hypotheses about the observable behavior of the program then test these hypotheses in the hope of finding the fault that caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debugging tools, which show the intermediate values of program variables and a trace of the statements executed, may be used to support the debugging process.

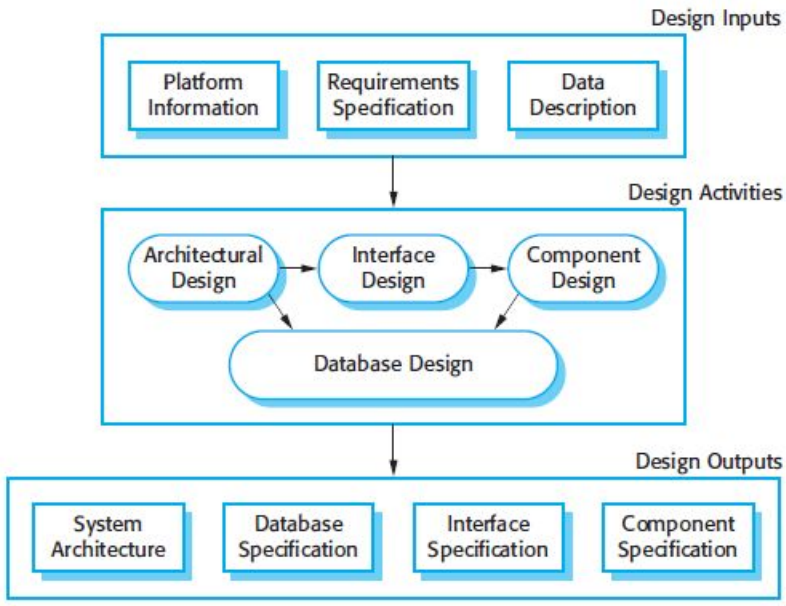


Figure 2.5 A general model of the design process

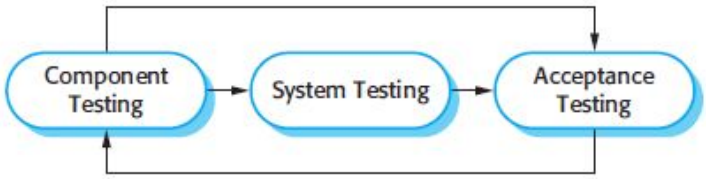


Figure 2.6 Stages of testing

Software validation

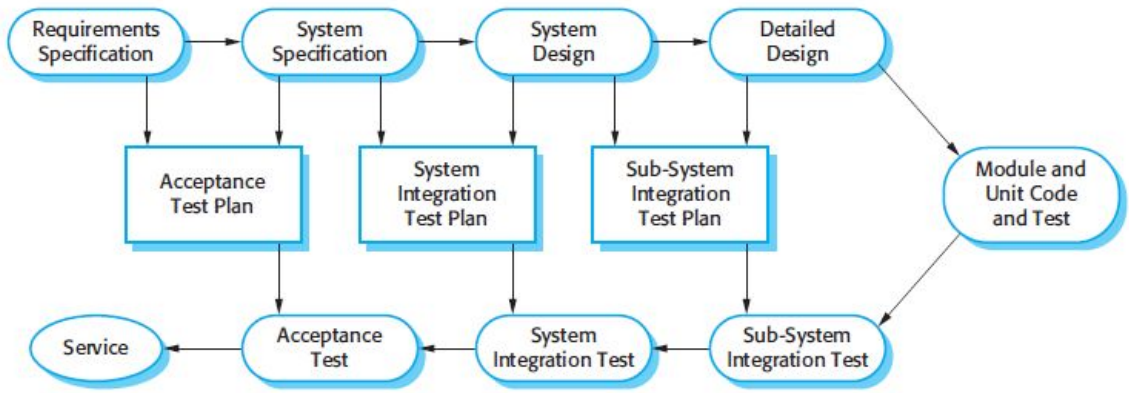
Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. Because of the predominance of testing, the majority of validation costs are incurred during and after implementation.

Except for small programs, systems should not be tested as a single, monolithic unit. Figure 2.6 shows a three-stage testing process in which system components are tested then the integrated system is tested and, finally, the system is tested with the customer's data. Ideally, component defects are discovered early in the process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

The stages in the testing process are:

1. Development testing. The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities. Test automation tools, such as JUnit (Massol and Husted, 2003), that can re-run component tests when new versions of the component are created, are commonly used.
2. System testing. System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multi-stage process where components are integrated to form subsystems that are individually tested before these sub-systems are themselves integrated to form the final system.
3. Acceptance testing. This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do

not really meet the user's needs or the system performance is unacceptable.
Normally, component development and testing processes are interleaved. Programmers make up their own test data and incrementally test the code as it is developed. This is an economically sensible approach, as the programmer knows the component and is therefore the best person to generate test cases.
If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment.
In extreme programming, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.
When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans. An independent team of testers works from these pre-formulated test plans, which have been developed from the system specification and design. Figure 2.7 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development (turn it on its side to see the V).
Acceptance testing is sometimes called 'alpha testing'. Custom systems are developed for a single client. The alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implementation of the requirements.
When a system is to be marketed as a software product, a testing process called 'beta testing' is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the system builders. After this feedback, the system is modified and released either for further beta testing or for general sale.



Software evolution

The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.
Historically, there has always been a split between the process of software development

and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. Although the costs of maintenance are often several times the initial development costs, maintenance processes are sometimes considered to be less challenging than original software development.
This distinction between development and maintenance is increasingly irrelevant. Hardly any software systems are completely new systems and it makes much more sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.8) where software is continually changed over its lifetime in response to changing requirements and customer needs.

Question: Process Phases (4 points)

Give an account of Boehm's spiral model – the risk-driven software process framework.

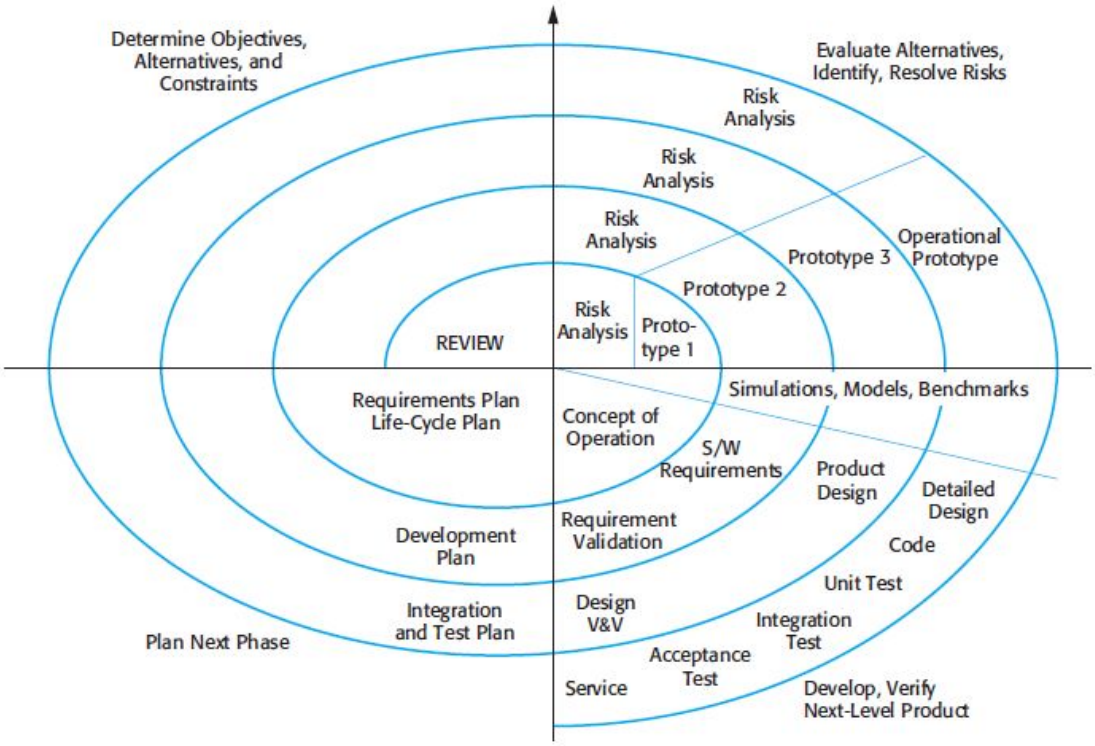
A risk-driven software process framework (the spiral model) was proposed by Boehm (1988). This is shown in Figure 2.11. Here, the software process is represented as a spiral, rather than a sequence of activities with some backtracking from one activity to another. Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.

Each loop in the spiral is split into four sectors:

1. Objective setting. Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.
2. Risk assessment and reduction. For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
3. Development and validation. After risk evaluation, a development model for the system is chosen. For example, throwaway prototyping may be the best development approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-system integration, the waterfall model may be the best development model to use.
4. Planning. The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is its explicit recognition of risk. A cycle of the spiral begins by elaborating objectives such as performance and functionality. Alternative ways of achieving these objectives, and dealing with the constraints on each of them, are then enumerated. Each

alternative is assessed against each objective and sources of project risk are identified. The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping, and simulation.
Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code. Risks lead to proposed software changes and project problems such as schedule and cost overrun, so risk minimization is a very important project management activity. Risk management, an essential part of project management, is covered in Chapter 22.



Requirements

Question: Requirement levels (3 points)
Sommerville describes two levels of requirements. What are they? (1 point) Why do you need to write requirements on different levels of detail? (1 point) How are these requirements structured into hierarchies? (1 point)

(Amina) [Sommerville Kap 4, sida 83]
Sommerville beskriver 2 nivåer av requirements.

User requirements: Hög nivå, en abstrakt beskrivning av den funktionalitet ett system ska uppfylla.
System requirements: En detaljerad formell definition av ett systems funktion.

User requirements skrivs av en kund som letar leverantör, syftet med att skriva en så abstrakt beskrivning som möjligt är för att olika leverantörer ska kunna presentera olika möjliga lösningar. När en leverantör har valts ska den valda leverantören presentera system requirements. Vidare är de olika nivåer av requirements till för olika läsare (Bild 1).

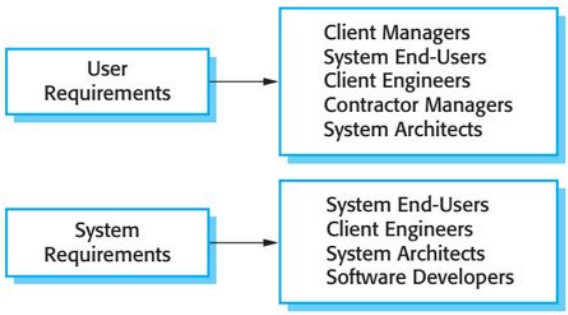


Bild 1, sommerville sida 85.

Hierarkiskt är requirements organiserade så att user requirements är överlägsen system requirements (Bild 2).

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Bild 2, sommerville sida 84.

Question: Requirement types (4 points)
Software requirements are often grouped into functional, non-functional requirements. Describe briefly each requirements type and exemplify it.

(Från boken)

Software system requirements are often classified as functional requirements or nonfunctional requirements:
1. Functional requirements. These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. Non-functional requirements. These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

In reality, the distinction between different types of requirement is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a nonfunctional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.
This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered properly.

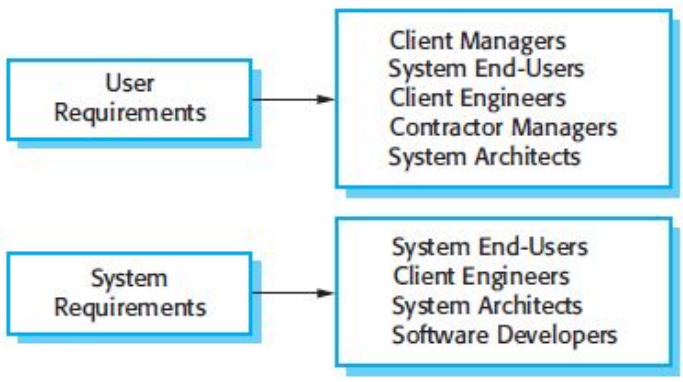


Figure 4.2 Readers of different types of requirements specification

Question: Non-functional requirements (2 points)
Describe three main different types of non-functional requirement which may be placed on a system. Give at least one example of each of the types.

1. Product requirements These requirements specify or constrain the behavior of the software. Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.

2. Organizational requirements These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization. Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.

3. External requirements This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a central bank; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

Question: Defining requirements in natural language (2 points)

Discuss the problem of using natural language for defining user and system requirements and show, using small examples, how structuring natural language into forms can help avoid some of these difficulties.

To use a structured approach to specifying system requirements, you define one or more standard templates for requirements and represent these templates as structured forms. The specification may be structured around the objects manipulated by the system, the functions performed by the system, or the events processed by the system. An example of a form-based specification, in this case, one that defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band, is shown in Figure 4.10.

When a standard form is used for specifying functional requirements, the following information should be included:

1. A description of the function or entity being specified.
2. A description of its inputs and where these come from.
3. A description of its outputs and where these go to.
4. Information about the information that is needed for the computation or other entities in the system that are used (the 'requires' part).
5. A description of the action to be taken.
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called, and a post-condition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.

Using structured specifications removes some of the problems of natural language specification. Variability in the specification is reduced and requirements are organized more effectively. However, it is still sometimes difficult to write requirements in a clear and unambiguous way, particularly when complex computations (e.g., how to calculate the insulin dose) are to be specified.

To address this problem, you can add extra information to natural language

requirements, for example, by using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system, and how sequences of actions are performed.

(Gustav)

Various problems can arise when requirements are written in natural language sentences, in a text document:
Lack of clarity. It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.
Requirements confusion. Functional requirements, non-functional requirements, system goals and design information may not be clearly distinguished.
Requirements amalgamation. Several different requirements may be expressed together as a single requirement.
Natural language is often used to write system requirements specifications as well as user requirements. However, because system requirements are more detailed than user requirements, natural language specifications can be confusing and hard to understand:
1.Natural language understanding relies on the specification readers and writers using the same words for the same concept. This leads to misunderstandings because of the ambiguity of natural language. Jackson (Jackson, 1995) gives an excellent example of this when he discusses signs displayed by an escalator. These said ‘Shoes must be worn’ and ‘Dogs must be carried’. I leave it to you to work out the conflicting interpretations of these phrases.
2.A natural language requirements specification is overflexible. You can say the same thing in completely different ways. It is up to the reader to find out when requirements are the same and when they are distinct.
3.There is no easy way to modularise natural language requirements. It may be difficult to find all related requirements. To discover the consequence of a change, you may have to look at every requirement rather than at just a group of related requirements.
Because of these problems, requirements specifications written in natural language are prone to misunderstandings. These are often not discovered until later phases of the software process and may then be very expensive to resolve.

To minimize misunderstandings when writing natural language requirements, I recommend that you follow some simple guidelines:
1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. The format I use expresses the requirement in a single sentence. I associate a statement of rationale with each user requirement to explain why the requirement has been proposed. The rationale may also include information on who proposed the requirement (the requirement source) so that you know whom to consult if the requirement has to be changed.
2. Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using ‘shall’. Desirable requirements are not essential and are written using ‘should’.

3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical software engineering language. It is easy for words like ‘architecture’ and ‘module’ to be misunderstood. You should, therefore, avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included. It is particularly useful when requirements are changed as it may help decide what changes would be undesirable.

Question: Requirements engineering process phase (4 points)

After initial feasibility studies, the next stage of the requirements engineering process is requirements elicitation and analysis.

Describe the process and each of its phases. (2 points)

Give four reasons for why the elicitation and analysis phase is difficult. (2 points)

(Hannes sammanfattning)

1. Requirements discovery - samla in alla krav
2. Requirements classification and organization - gruppera dem
3. Requirements prioritization and negotiation - prioritera dem
4. Requirements specification - skapa formell kravspecifikation
(2p)

Svårigheter

1. **Stakeholders vet inte vad de vill ha och har svårt att sätta ord på det och gör orealistiska krav.**
2. **Stakeholder uttrycker sina krav på sitt fackspråk och utvecklarna fattar inte.**
3. **Stakeholder uttrycker sina krav på olika sätt, men kan mena samma sak och tvärtom, krav-skaparen som kunna tyda dessa skillnader.**
4. **Political factors - krav kommer från folk som ställer kraven för egen vinning, typ karriärsmänniskor.**
5. **The economic and business environment - kraven ändras om de som först ställde kraven inte är med längre och nya kravställare har tagit över.**

(2p)

(Från boken)

1. Requirements discovery

This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity. There are several complementary techniques that can be used for requirements discovery, which I discuss later in this section.

2. Requirements classification and organization

This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements

is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.

3. Requirements prioritization and negotiation
Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. Requirements specification
The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced, as discussed in Section 4.3.
(2p)

1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.

2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.

3. Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.

4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.

5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.
(2p)

Question: Requirements engineering discovery process phase (4 points)
When discovering requirements, you may use different techniques such as

1. Use cases
2. Interviewing
3. Scenarios
4. Ethnography

Describe (2 points) and compare two of them (2 points).

Requirements discovery

Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information. Sources of information during the requirements discovery phase include documentation, system stakeholders, and specifications of similar systems. You interact with stakeholders through interviews and observation and you may use scenarios and prototypes to help stakeholders understand what the system will be like.

Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system. For example, system stakeholders for the mental health care patient information system include:

1. Patients whose information is recorded in the system.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Healthcare managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

In addition to system stakeholders, we have already seen that requirements may also come from the application domain and from other systems that interact with the system being specified. All of these must be considered during the requirements elicitation process.

These different requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints with each viewpoint showing a subset of the requirements for the system. Different viewpoints on a problem see the problem in different ways. However, their perspectives are not completely independent but usually overlap so that they have common requirements. You can use these viewpoints to structure both the discovery and the documentation of the system requirements.

Interviewing

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions.

Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a pre-defined set of questions.
2. Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

In practice, interviews with stakeholders are normally a mixture of both of these.
You may have to obtain the answer to certain questions but these usually lead on to other issues that are discussed in a less structured way. Completely open-ended discussions rarely work well. You usually have to ask some questions to get started and to keep the interview focused on the system to be developed.
Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems. People like talking about their work so are usually happy to get involved in interviews. However, interviews are not so helpful in understanding the requirements from the application domain.
It can be difficult to elicit domain knowledge through interviews for two reasons:
1. All application specialists use terminology and jargon that are specific to a domain. It is impossible for them to discuss domain requirements without using this terminology. They normally use terminology in a precise and subtle way that is easy for requirements engineers to misunderstand.
2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning.
For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.
Interviews are also not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power relationships between the different people in the organization. Published organizational structures rarely match the reality of decision making in an organization but interviewees may not wish to reveal the actual rather than the theoretical structure to a stranger. In general, most people are generally reluctant to discuss political and organizational issues that may affect the requirements.
Effective interviewers have two characteristics:
1. They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then they are willing to change their mind about the system.
2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.
Information from interviews supplements other information about the system from documentation describing business processes or existing systems, user observations, etc. Sometimes, apart from the information in the system documents, the interview information may be the only source of information about the system requirements. However, interviewing on its own is liable to miss essential information and so it should be used in conjunction with other requirements elicitation techniques.

Scenarios
People usually find it easier to relate to real-life examples rather than abstract descriptions. They can understand and criticize a scenario of how they might interact with a software system. Requirements engineers can use the information gained

from this discussion to formulate the actual system requirements.
Scenarios can be particularly useful for adding detail to an outline requirements description. They are descriptions of example interaction sessions. Each scenario usually covers one or a small number of possible interactions. Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system. The stories used in extreme programming, discussed in Chapter 3, are a type of requirements scenario.
A scenario starts with an outline of the interaction. During the elicitation process, details are added to this to create a complete description of that interaction. At its most general, a scenario may include:
1. A description of what the system and users expects when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how this is handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario finishes.
Scenario-based elicitation involves working with stakeholders to identify scenarios and to capture details to be included in these scenarios. Scenarios may be written as text, supplemented by diagrams, screen shots, etc. Alternatively, a more structured approach such as event scenarios or use cases may be used.
As an example of a simple text scenario, consider how the MHC-PMS may be used to enter data for a new patient (Figure 4.14). When a new patient attends a clinic, a new record is created by a medical receptionist and personal information (name, age, etc.) is added to it. A nurse then interviews the patient and collects medical history. The patient then has an initial consultation with a doctor who makes a diagnosis and, if appropriate, recommends a course of treatment. The scenario shows what happens when medical history is collected.

Use cases
Use cases are a requirements discovery technique that were first introduced in the Objectory method (Jacobson et al., 1993). They have now become a fundamental feature of the unified modeling language. In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction. This is then supplemented by additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as UML sequence or state charts.
Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements. Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse.
Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 4.15, which shows some of the use cases for the patient information system.
There is no hard and fast distinction between scenarios and use cases. Some people consider that each use case is a single scenario; others, as suggested by Stevens and Pooley (2006), encapsulate a set of scenarios in a single use case. Each scenario is a single thread through the use case. Therefore, there would be a scenario for the

normal interaction plus scenarios for each possible exception. You can, in practice, use them in either way.

Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail. For example, a brief description of the Setup Consultation use case from Figure 4.15 might be:

Setup consultation allows two or more doctors, working in different offices, to view the same record at the same time. One doctor initiates the consultation by choosing the people involved from a drop-down menu of doctors who are online.

The patient record is then displayed on their screens but only the initiating doctor can edit the record. In addition, a text chat window is created to help coordinate actions. It is assumed that a phone conference for voice communication will be separately set up.

Scenarios and use cases are effective techniques for eliciting requirements from stakeholders who interact directly with the system. Each type of interaction can be represented as a use case. However, because they focus on interactions with the system, they are not as effective for eliciting constraints or high-level business and nonfunctional requirements or for discovering domain requirements.

The UML is a de facto standard for object-oriented modeling, so use cases and use case–based elicitation are now widely used for requirements elicitation. I discuss use cases further in Chapter 5 and show how they are used alongside other system models to document a system design.

Ethnography

Software systems do not exist in isolation. They are used in a social and organizational context and software system requirements may be derived or constrained by that context. Satisfying these social and organizational requirements is often critical for the success of the system. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how the social and organizational context affects the practical operation of the system.

Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes. An analyst immerses himself or herself in the working environment where the system will be used. The day-to-day work is observed and notes made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship to other work in the organization. Social and organizational factors that affect the work, but which are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a work group may self-organize so that members know of each other's work and can cover for each other if someone is absent. This may not be mentioned during an interview as the group might not see it as an integral part of their work.

Suchman (1987) pioneered the use of ethnography to study office work. She

found that the actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity. Crabtree (2003) discusses a wide range of studies since then and describes, in general, the use of ethnography in systems design. In my own research, I have investigated methods of integrating ethnography into the software engineering process by linking it with requirements engineering methods (Viller and Sommerville, 1999; Viller and Sommerville, 2000) and documenting patterns of interaction in cooperative systems (Martin et al., 2001; Martin et al., 2002; Martin and Sommerville, 2004).

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. They deliberately put the aircraft on conflicting paths for a short time to help manage the airspace. Their control strategy is designed to ensure that these aircrafts are moved apart before problems occur and they find that the conflict alert alarm distracts them from their work.
2. Requirements that are derived from cooperation and awareness of other people's activities. For example, air traffic controllers may use an awareness of other controllers' work to predict the number of aircrafts that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.

Ethnography can be combined with prototyping (Figure 4.16). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study (Sommerville et al., 1993).

Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end-user, this approach is not always appropriate for discovering organizational or domain requirements. They cannot always identify new features that should be added to a system. Ethnography is not, therefore, a complete approach to elicitation on its own and it should be used to complement other approaches, such as use case analysis.

Question: Requirements engineering discovery process phase (2 points)
Describe(1 point) and motivate (1 point) the role of ethnography within requirements engineering process?

(Hannes)
Etnografi handlar om att de som skapar och samlar in krav gör det på plats där produkten ska användas.
(1p)
Etnografer tar in fakta om krav som annars skulle vara svåra att få in genom andra sätt. Genom att sitta bredvid kunden på arbetsplatsen kan etnografen se hur kunden arbetar och på så sätt ta in information som kunden annars skulle ha svårt att förklara med ord. Detta sätt kortar ner tiden som läggs ner under prototyp-stadiet av produktframtagningen.
(1p)

(Från boken)
ethnography
An observational technique that may be used in requirements elicitation and analysis. The ethnographer immerses him- or herself in the users' environment and observes their day-to-day work habits. Requirements for software support can be inferred from these observations.

Software systems do not exist in isolation. They are used in a social and organizational context and software system requirements may be derived or constrained by that context. Satisfying these social and organizational requirements is often critical for the success of the system. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how the social and organizational context affects the practical operation of the system.
Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes. An analyst immerses himself or herself in the working environment where the system will be used. The day-to-day work is observed and notes made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.
People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship to other work in the organization. Social and organizational factors that affect the work, but which are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a work group may self-organize so that members know of each other's work and can cover for each other if someone is absent. This may not be mentioned during an interview as the group might not see it as an integral part of their work.
Suchman (1987) pioneered the use of ethnography to study office work. She found that the actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity. Crabtree (2003) discusses a wide range of studies since then and describes, in general, the use of ethnography in systems design. In my own research, I have investigated methods of integrating ethnography into the software engineering process by linking it with

requirements engineering methods (Viller and Sommerville, 1999; Viller and Sommerville, 2000) and documenting patterns of interaction in cooperative systems (Martin et al., 2001; Martin et al., 2002; Martin and Sommerville, 2004).
Ethnography is particularly effective for discovering two types of requirements:
1. Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. They deliberately put the aircraft on conflicting paths for a short time to help manage the airspace. Their control strategy is designed to ensure that these aircrafts are moved apart before problems occur and they find that the conflict alert alarm distracts them from their work.
2. Requirements that are derived from cooperation and awareness of other people's activities. For example, air traffic controllers may use an awareness of other controllers' work to predict the number of aircrafts that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.
Ethnography can be combined with prototyping (Figure 4.16). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study (Sommerville et al., 1993).
Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end-user, this approach is not always appropriate for discovering organizational or domain requirements. They cannot always identify new features that should be added to a system. Ethnography is not, therefore, a complete approach to elicitation on its own and it should be used to complement other approaches, such as use case analysis.

Question: Requirements engineering discovery process phase (3 points)
During the requirements validation phase, different types of checks should be carried out on the requirements in the requirements document. Sommerville suggests five types of checks. List and describe three of them.

1. **Validity checks**

A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.

2. Consistency checks
Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. Completeness checks
The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. Realism checks
Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.

5. Verifiability
To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

Question: Requirements engineering discovery process phase (3 points)
There are a number of requirements validation techniques which can be used. These are requirements reviews, prototyping, and test-case generation. Describe each of them.

1. **Requirements reviews**
The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

2. **Prototyping**
In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

3. **Test-case generation**
Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered.
Developing tests from the user requirements before any code is written is an integral part of extreme programming.

Question: Requirements change management (3 points)
The requirements for software systems are always changing. Hence, one must define a process of understanding and controlling changes to system requirements. Describe the requirements management process.

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing (Figure 4.17). The system requirements must then also evolve to reflect this changed problem view.
Once a system has been installed and is regularly used, new requirements inevitably emerge. It is hard for users and system customers to anticipate what effects the new system will have on their business processes and the way that work is done. Once endusers have experience of a system, they will discover new needs and priorities. There are several reasons why change is inevitable:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.

2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

3. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements management is the process of understanding and controlling changes to system requirements. You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements. The formal process of requirements management should start as soon as a draft version of the requirements document is available. However, you should start planning how to manage changing requirements during the requirements elicitation process.

Requirements management planning
Planning is an essential first stage in the requirements management process. The planning stage establishes the level of requirements management detail that is required. During the requirements management stage, you have to decide on:

1. Requirements identification. Each requirement must be uniquely identified so

that it can be cross-referenced with other requirements and used in traceability assessments.

2. A change management process. This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.

3. Traceability policies. These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.

4. Tool support. Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements management needs automated support and the software tools for this should be chosen during the planning phase. You need tool support for:

1. Requirements storage. The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.

2. Change management. The process of change management (Figure 4.18) is simplified if active tool support is available.

3. Traceability management. As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, it may not be necessary to use specialized requirements management tools. The requirements management process may be supported using the facilities available in word processors, spreadsheets, and PC databases. However, for larger systems, more specialized tool support is required. I have included links to information about requirements management tools in the book's web pages.

Requirements change management

Requirements change management (Figure 4.18) should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way. There are three principal stages to a change management process:

1. Problem analysis and change specification.
The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During

this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. Change analysis and costing.
The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

3. Change implementation.
The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. You should try to avoid this as it almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document or to add information to the requirements document that is inconsistent with the implementation. Agile development processes, such as extreme programming, have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped.

KEY POINTS

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used. These might be product requirements, organizational requirements, or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification, requirements validation, and requirements management.
- Requirements elicitation and analysis is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.
- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism, and verifiability.
- Business, organizational, and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

Testing

Question: Verification and validation (2 points)
Verification and validation are not the same thing. What is the difference? Explain why validation is a particularly difficult process.
(Mira, s207 sommerville, Föreläsning 8) Validation testing testar om vi bygger rätt produkt medan verification testing testar om vi bygger produkten rätt. Validation testing är svårare då det handlar om att på något vis testa att produkten möter kundens förväntningar. Validation testing är nödvändig då kravspecifikationer inte alltid är tillräckliga för att uppnå vad kunden vill ha.

(Hannes, från boken)
Verification - formell testning mot functional och non-funktional krav.

Validation - se till att kunden är nöjd med produkten, att kundens förväntningar är mötta, vilket inte kan mätas med hjälp av kravspecifikationen.

Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.
The aim of verification is to check that the software meets its stated functional and non-functional requirements. Validation, however, is a more general process. The aim of validation is to ensure that the software meets the customer's expectations. It goes beyond simply checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because, as I discussed in Chapter 4, requirements specifications do not always reflect the real wishes or needs of system customers and users.
The ultimate goal of verification and validation processes is to establish confidence that the software system is 'fit for purpose'. This means that the system must be good enough for its intended use.

Question: Confidence level in verification and validation (3 points)
The ultimate goal of verification and validation is to establish confidence that the software system is "fit for purpose". However, the level of confidence depends on (1) software purpose, (2) user expectations, and (3) marketing environment. Motivate how the level of confidence varies with respect to the three above-listed reasons.

(Mira, s207 sommerville) Level of confidence för de olika punkterna menas med hur hög denna nivån behöver vara i just detta fall. Alltså systemets syfte avgör hur hög level of confidence som krävs för respektive punkt(1-3).

1. Software purpose. Level of confidence på mjukvaran avgörs såklart av vad systemet är till för, ska systemet hantera säkerhet är mjukvaran viktigare än om det är en enkel prototyp som skapas.
2. User expectations. Level of confidence för användar förväntningar stiger i och med antal versioner på mjukvaran.
3. Marketing environment. Level of confidence vad gäller marknadsföring beror på antalet liknande produkter på marknaden. Man måste dels räkna in detta och dels priset som kunden är villig att betala.

(Hannes, Från boken)

1. **Software purpose**

The more critical the software, the more important that it is reliable. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.

2. **User expectations**

Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery.
In these situations, you may not need to devote as much time to testing the software. However, as software matures, users expect it to become more reliable so more thorough testing of later versions may be required.

3. **Marketing environment**

When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. In a competitive environment, a software company may decide to release a program before it has been fully tested and debugged because they want to be the first into the market. If a software product is very cheap, users may be willing to tolerate a lower level of reliability.

Question: Program inspections (3 points)

Sommerville provides three advantages of program inspections. What are they?

(Hannes, från boken)

1. During testing, errors can mask (hide) other errors. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side

effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.

2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.

3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability, and maintainability. You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

Question: Equivalence partitioning (2 points)
Give an account of equivalence partitioning within testing.

(Hannes, från boken)
The input data and output results of a program often fall into a number of different classes with common characteristics. Examples of these classes are positive numbers, negative numbers, and menu selections. Programs normally behave in a comparable way for all members of a class. That is, if you test a program that does a computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers.
Because of this equivalent behavior, these classes are sometimes called equivalence partitions or domains (Bezier, 1990). One systematic approach to test case design is based on identifying all input and output partitions for a system or component. Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.
In Figure 8.5, the large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested. The smaller unshaded ellipses represent equivalence partitions. A program being tested should process all of the members of an input equivalence partitions in the same way. Output equivalence partitions are partitions within which all of the outputs have something in common. Sometimes there is a 1:1 mapping between input and output equivalence partitions. However, this is not always the case; you may need to define a separate input equivalence partition, where the only common characteristic of the inputs is that they generate outputs within the same output partition. The shaded area in the left ellipse represents inputs that are invalid. The shaded area in the right ellipse represents exceptions that may occur (i.e., responses to invalid inputs).
Once you have identified a set of partitions, you choose test cases from each of these partitions. A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition.

The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system. You test these by choosing the midpoint of the partition. Boundary values are often atypical (e.g., zero may behave differently from other non-negative numbers) so are sometimes overlooked by developers. Program failures often occur when processing these atypical values.

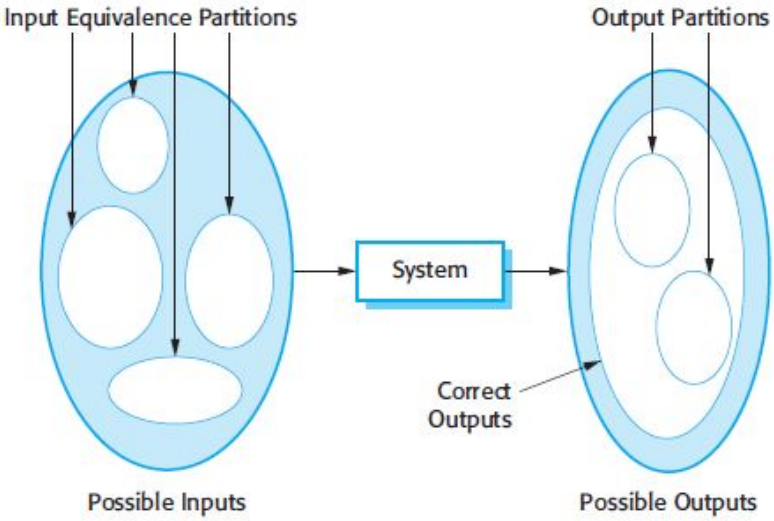


Figure 8.5 Equivalence partitioning

Question: Component testing (4 points)

There are different types of interface between program components. These are (1) parameter interfaces, (2) shared memory interfaces, (3) procedural interfaces, and (4) message passing interfaces. Explain them.

1. Parameter interfaces

These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.

2. Shared memory interfaces

These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other subsystems. This type of interface is often used in embedded systems, where sensors create data that is retrieved and processed by other system components.

3. Procedural interfaces

These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.

4. Message passing interfaces

These are interfaces in which one component requests a service from another component by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client–server systems.

Question: Component testing (4 points)
There are different classes of interface errors. Sommerville lists (1) interface misuse, (2) interface misunderstanding, and (3) timing errors. Explain them.

(Hannes, från boken)
Interface misuse
A calling component calls some other component and makes an error in the use of its interface. This type of error is common with parameter interfaces, where parameters may be of the wrong type or be passed in the wrong order, or the wrong number of parameters may be passed.

Interface misunderstanding
A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior. The called component does not behave as expected which then causes unexpected behavior in the calling component. For example, a binary search method may be called with a parameter that is an unordered array. The search would then fail.

Timing errors
These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

Question: Interface testing (2 points)
Why is interface testing difficult?

(Hannes, från boken)
Testing for interface defects is difficult because some interface faults may only manifest themselves under unusual conditions. For example, say an object implements a queue as a fixed-length data structure. A calling object may assume that the queue is implemented as an infinite data structure and may not check for queue overflow when an item is entered. This condition can only be detected during testing by designing test cases that force the queue to overflow and cause that overflow to corrupt the object behavior in some detectable way.
A further problem may arise because of interactions between faults in different modules or objects. Faults in one object may only be detected when some other object behaves in an unexpected way. For example, an object may call another object to receive some service and assume that the response is correct. If the called service is faulty in some way, the returned value may be valid but incorrect. This is not immediately detected but only becomes obvious when some later computation goes wrong.

Question: Test-driven development (4 points)
Given an account of test-driven development. What are its fundamental steps? (2 points).
Mention at least its two benefits. (2 points)

(Gustav, från boken)
Test-driven development (TDD) is an approach to program development in which you interleave testing and code development (Beck, 2002; Jeffries and Melnik, 2007). Essentially, you develop the code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test. Test-driven development was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.
The steps in the process are as follows:
1. You start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
2. You write a test for this functionality and implement this as an automated test. This means that the test can be executed and will report whether or not it has passed or failed.
3. You then run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail. This is deliberate as it shows that the test adds something to the test set.
4. You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what's already there.
5. Once all tests run successfully, you move on to implementing the next chunk of functionality.

As well as better problem understanding, other benefits of test-driven development are:
1. Code coverage.
In principle, every code segment that you write should have at least one associated test. Therefore, you can be confident that all of the code in the system has actually been executed. Code is tested as it is written so defects are discovered early in the development process.

2. Regression testing.
A test suite is developed incrementally as a program is developed. You can always run regression tests to check that changes to the program have not introduced new bugs.

3. Simplified debugging.
When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified. You do not need to use debugging tools to locate the problem. Reports of the use of test-driven development suggest that it is hardly ever necessary to use an automated debugger in test-driven development (Martin, 2007).

4. System documentation.
The tests themselves act as a form of documentation that
describe what the code should be doing. Reading the tests can make it easier to
understand the code.

Question: User testing (1,5 points)
Sommerville distinguishes three types of user testing. What are they and what do they imply?

(Hannes, från boken)
1. Alpha testing, where users of the software work with the development team to
test the software at the developer's site.

2. Beta testing, where a release of the software is made available to users to allow
them to experiment and to raise problems that they discover with the system
developers.

3. Acceptance testing, where customers test a system to decide whether or not it is
ready to be accepted from the system developers and deployed in the customer
environment.

Question: Acceptance testing (6 points)
Sommerville identifies six stages in the acceptance testing process. These are (1) define acceptance criteria, (2) plan acceptance testing, (3) derive acceptance tests, (4) run acceptance tests, (5) negotiate test results, and (6) reject/accept system. Give an account of these phases.

(Hannes, från boken)
1. Define acceptance criteria.
This stage should, ideally, take place early in the
process before the contract for the system is signed. The acceptance criteria
should be part of the system contract and be agreed between the customer and
the developer. In practice, however, it can be difficult to define criteria so early
in the process. Detailed requirements may not be available and there may be significant
requirements change during the development process.

2. Plan acceptance testing.
This involves deciding on the resources, time, and
budget for acceptance testing and establishing a testing schedule. The acceptance
test plan should also discuss the required coverage of the requirements and
the order in which system features are tested. It should define risks to the testing
process, such as system crashes and inadequate performance, and discuss how

these risks can be mitigated.
3. Derive acceptance tests.
Once acceptance criteria have been established, tests
have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system. They should, ideally, provide complete coverage of the system requirements. In practice, it is difficult to establish completely objective acceptance criteria. There is often scope for argument about whether or not a test shows that a criterion has definitely been met.

4. Run acceptance tests.
The agreed acceptance tests are executed on the system.
Ideally, this should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system. Some training of end-users may be required.

5. Negotiate test results.
It is very unlikely that all of the defined acceptance tests will
pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be put into use. They must also agree on the developer's response to identified problems.

6. Reject/accept system.
This stage involves a meeting between the developers
and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.

Project Management

Question: Differences with other engineering projects (3 points)

Most projects have important goals such as

1. Deliver the software to the customer at the agreed time.
2. Keep overall costs within budget.

3. Deliver software that meets the customer's expectations.
4. Maintain happy and well-functioning development team.

These goals are not unique to other engineering projects. However, software engineering differs in a number of ways from other engineering projects. List and explain three such differences.

(Hannes, från boken)
1. The product is intangible.
A manager of a shipbuilding or a civil engineering
project can see the product being developed. If a schedule slips, the effect on
the product is visible—parts of the structure are obviously unfinished.
Software is intangible. It cannot be seen or touched. Software project managers
cannot see progress by simply looking at the artifact that is being constructed.
Rather, they rely on others to produce evidence that they can use to review the
progress of the work.

2. Large software projects are often 'one-off ' projects.
Large software projects are usually different in some ways from previous projects. Therefore, even managers who have a large body of previous experience may find it difficult to anticipate problems. Furthermore, rapid technological changes in computers and communications can make a manager's experience obsolete. Lessons learned from previous projects may not be transferable to new projects.

3. Software processes are variable and organization-specific.
The engineering process for some types of system, such as bridges and buildings, is well understood. However, software processes vary quite significantly from one organization to another. Although there has been significant progress in process standardization and improvement, we still cannot reliably predict when a particular software process is likely to lead to development problems. This is especially true when the software project is part of a wider systems engineering project.

Question: Risk management (4 points)
Give an account of a risk management process and its phases.

(Hannes, från boken)
Risk management is particularly important for software projects because of the inherent uncertainties that most projects face. These 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. You have to anticipate risks; understand the impact of these risks on the project, the product, and the business; and take steps to avoid these risks. You may need to draw up contingency plans so that, if the risks do occur, you

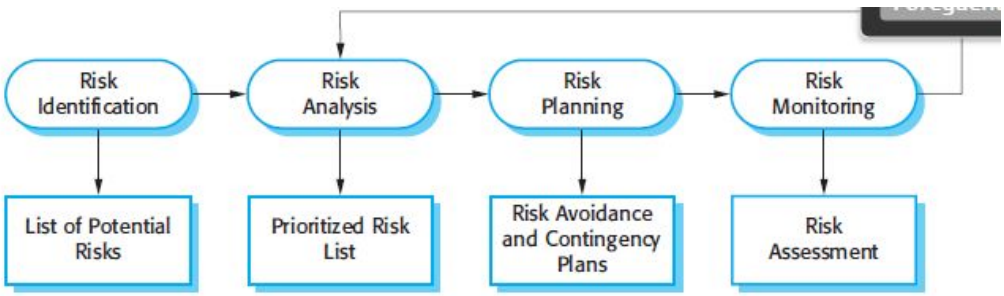
can take immediate recovery action.
An outline of the process of risk management is illustrated in Figure 22.2. It involves several stages:
1. Risk identification.
You should identify possible project, product, and business risks.

2. Risk analysis.
You should assess the likelihood and consequences of these risks.

3. Risk planning.
You should make plans to address the risk, either by avoiding it or minimizing its effects on the project.

4. Risk monitoring.
You should regularly assess the risk and your plans for risk mitigation and revise these when you learn more about the risk.

You should document the outcomes of the risk management process in a risk management plan. This should include a discussion of the risks faced by the project, an analysis of these risks, and information on how you propose to manage the risk if it seems likely to be a problem.
The risk management process is an iterative process that continues throughout the project. Once you have drawn up an initial risk management plan, you monitor the situation to detect emerging risks. As more information about the risks becomes available, you have to reanalyze the risks and decide if the risk priority has changed. You may then have to change your plans for risk avoidance and contingency management.



Question: Risk types (4 points)
Sommerville categorizes risks into three groups:
1. Project risks
2. Product risks
3. Business risks.

Explain each risk category and provide an example (3 points). Why do the risk types overlap sometimes? (1 point)

1. Project risks

Risks that affect the project schedule or resources. An example of a project risk is the loss of an experienced designer. Finding a replacement designer with appropriate skills and experience may take a long time and, consequently, the software design will take longer to complete.

2. Product risks
Risks that affect the quality or performance of the software being developed. An example of a product risk is the failure of a purchased component to perform as expected. This may affect the overall performance of the system so that it is slower than expected.

3. Business risks
Risks that affect the organization developing or procuring the software. For example, a competitor introducing a new product is a business risk. The introduction of a competitive product may mean that the assumptions made about sales of existing software products may be unduly optimistic.

Of course, these risk types overlap. If an experienced programmer leaves a project this can be a project risk because, even if they are immediately replaced, the schedule will be affected. It inevitably takes time for a new project member to understand the work that has been done, so they cannot be immediately productive. Consequently, the delivery of the system may be delayed. The loss of a team member can also be a product risk because a replacement may not be as experienced and so could make programming errors. Finally, it can be a business risk because that programmer's experience may be crucial in winning new contracts.

Question: Risk strategies (3 points)

The risk planning process considers each of the key risks that have been identified and develops strategies for managing these risks. Sommerville identifies three strategies. What are they? Provide an example for each strategy.

1. Avoidance strategies.
Following these strategies means that the probability that the risk will arise will be reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components shown in Figure 22.5.

2. Minimization strategies.
Following these strategies means that the impact of the risk will be reduced. An example of a risk minimization strategy is the strategy for staff illness shown in Figure 22.5.

3. Contingency plans.

Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial problems that I have shown in Figure 22.5.

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

Question: Planning stages (3 points)

Project planning takes place at three stages in a project lifecycle. Describe these stages.

1. At the **proposal stage**, when you are bidding for a contract to develop or provide a software system. You need a plan at this stage to help you decide if you have the resources to complete the work and to work out the price that you should quote to a customer.

2. During the project **startup phase**, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc. Here, you have more information than at the proposal stage, and can therefore refine the initial effort estimates that you have prepared.

3. Periodically **throughout the project**, when you modify your plan in light of experience gained and information from monitoring the progress of the work. You learn more about the system being implemented and capabilities of your development team. This information allows you to make more accurate estimates of how long the work will take. Furthermore, the software requirements are likely to change and this usually means that the work breakdown has to be altered and the schedule extended. For traditional development projects, this means that the plan created during the startup phase has to be modified. However, when an agile approach is used, plans are shorter term and continually change as the software evolves.

Question: Project scheduling (5 points)
Give an account of a project scheduling process.

Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed. You estimate the calendar time needed to complete each task, the effort required, and who will work on the tasks that have been identified. You also have to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be. In terms of the planning stages that I discussed in the introduction of this chapter, an initial project schedule is usually created during the project startup phase. This schedule is then refined and modified during development planning.

Both plan-based and agile processes need an initial project schedule, although the level of detail may be less in an agile project plan. This initial schedule is used to plan how people will be allocated to projects and to check the progress of the project against its contractual commitments. In traditional development processes, the complete schedule is initially developed and then modified as the project progresses. In agile processes, there has to be an overall schedule that identifies when the major phases of the project will be completed. An iterative approach to scheduling is then used to plan each phase.

Scheduling in plan-driven projects (Figure 23.4) involves breaking down the total work involved in a project into separate tasks and estimating the time required to complete each task. Tasks should normally last at least a week, and no longer than 2 months. Finer subdivision means that a disproportionate amount of time must be spent on replanning and updating the project plan. The maximum amount of time for any task should be around 8 to 10 weeks. If it takes longer than this, the task should be subdivided for project planning and scheduling.

Some of these tasks are carried out in parallel, with different people working on different components of the system. You have to coordinate these parallel tasks and organize the work so that the workforce is used optimally and you don't introduce unnecessary dependencies between the tasks. It is important to avoid a situation where the whole project is delayed because a critical task is unfinished.

If a project is technically advanced, initial estimates will almost certainly be optimistic even when you try to consider all eventualities. In this respect, software scheduling is no different from scheduling any other type of large advanced project. New aircraft, bridges, and even new models of cars are frequently late because of unanticipated problems. Schedules, therefore, must be continually updated as better progress information becomes available. If the project being scheduled is similar to a previous project, previous estimates may be reused. However, projects may use different design methods and implementation languages, so experience from previous projects may not be applicable in the planning of a new project.

As I have already suggested, when you are estimating schedules, you must take into account the possibility that things will go wrong. People working on a project may fall ill or leave, hardware may fail, and essential support software or hardware may be delivered late. If the project is new and technically advanced, parts of it may

turn out to be more difficult and take longer than originally anticipated.
A good rule of thumb is to estimate as if nothing will go wrong, then increase your estimate to cover anticipated problems. A further contingency factor to cover unanticipated problems may also be added to the estimate. This extra contingency factor depends on the type of project, the process parameters (deadline, standards, etc.), and the quality and experience of the software engineers working on the project. Contingency estimates may add 30% to 50% to the effort and time required for the project.

Question: Agile planning (5 points)

Most of the agile approaches such as Scrum and extreme programming have a two-stage approach to planning. What are the two stages (2 points) and how do they correspond to traditional planning (2 points)?

The most commonly used agile approaches such as Scrum and extreme programming have a two-stage approach to planning, corresponding to the startup phase in plan-driven development and development planning:

1. **Release planning**, which looks ahead for several months and decides on the features that should be included in a release of a system.

2. **Iteration planning**, which has a shorter-term outlook, and focuses on planning the next increment of a system. This is typically 2 to 4 weeks of work for the team.

Release planning involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented. The customer has to be involved in this process. A release date is then chosen and the stories are examined to see if the effort estimate is consistent with that date. If not, stories are added or removed from the list.

Iteration planning is the first stage into the iteration development process. Stories to be implemented for that iteration are chosen, with the number of stories reflecting the time to deliver an iteration (usually 2 or 3 weeks) and the team's velocity. When the iteration delivery date is reached, that iteration is complete, even if all of the stories have not been implemented. The team considers the stories that have been implemented and adds up their effort points. The velocity can then be recalculated and this is used in planning the next release of the system.

People Management

(Adin) Jag lade till sidnumret var jag hittade svaren ifall någon vill läsa på lite mer. jag svarade på frågorna innan jag hade tillgång till detta dokument så mina svar är baserade på 8th. Samma innehåll men sidnumren kanske skiljer sig lite.

Question: Managing people (5 points)

The people working are the greatest assets in a software organization. Therefore, to ensure that the organization gets the best possible return on investment, it is important to manage people with respect. Explain and motivate why it may not always be optimal to assign software engineers to the roles of project managers? (2 points)

(Aleks, p.602) It is important that software project managers understand the technical issues that influence the work of software development. Unfortunately, however, good software engineers are not necessarily good people managers. Software engineers often have strong technical skills but may lack the softer skills that enable them to motivate and lead a project development team. As a project manager, you should be aware of the potential problems of people management and should try to develop people management skills.

There are four critical factors that must be considered while managing people. These are *consistency, respect, inclusion* and *honesty*. (3 points). Explain and motivate three of them.

- (Adin) Honesty: as a manager you must be honest otherwise you will lose the respect of the group. The project will not succeed if you lie about your technical skills/knowledge.
- Consistency: people should be evaluated by their knowledge. Compare people and show that there contributes is valuable.
- Respect: people have different skills, the manager should respect that and give persons a chance to make contribution

P617

(Aleks, från boken, s.603)

1. *Consistency.* People in a project team should all be treated in a comparable way. No one expects all rewards to be identical but people should not feel that their contribution to the organization is undervalued.
2. *Respect.* Different people have different skills and managers should respect these differences. All members of the team should be given an opportunity to make a contribution. In some cases, of course, you will find that people simply don't fit into a team and they cannot continue, but it is important not to jump to conclusions about this at an early stage in the project.

3. *Inclusion.* People contribute effectively when they feel that others listen to them and take account of their proposals. It is important to develop a working environment where all views, even those of the most junior staff, are considered.
4. *Honesty.* As a manager, you should always be honest about what is going well and what is going badly in the team. You should also be honest about your level of technical knowledge and willing to defer to staff with more knowledge when necessary. If you try to cover up ignorance or problems you will eventually be found out and will lose the respect of the group.

Question: Motivating people (5 points)

Project managers need to motivate their people. Using Maslow's hierarchy of human needs, explain what employee needs project managers should focus on and how they may be satisfied.

- (Adin) The first level consist of the physical level that contains stuff as food, sleep and fundamental things. The second level consist of safety needs, the human needs to feel safe in his environment. The third level consist of social needs, such as being part of an group. And the fourth and last one is esteem needs which are basically being respected and having confidence. People need to satisfy lower-level needs like hunger before the more abstract, higher-level needs.

p622

(Aleks, p. 604) These needs are arranged in a series of levels, as shown in Figure 22.7. The lower levels of this hierarchy represent fundamental needs for food, sleep, and so on, and the need to feel secure in an environment. Social needs are concerned with the need to feel part of a social grouping. Esteem needs represent the need to feel respected by others, and self-realization needs are concerned with personal development. People need to satisfy lower-level needs like hunger before the more abstract, higher-level needs.

Figure 22.7 Human needs hierarchy

People working in software development organizations are not usually hungry or thirsty or physically threatened by their environment. Therefore, making sure that people's social, esteem, and self-realization needs are satisfied is most important from a management point of view.

1. To satisfy **social** needs, you need to give people time to meet their co-workers and provide places for them to meet. This is relatively easy when all of the members of a development team work in the same place but, increasingly, team members are not located in the same building or even the same town or state. They may work for different organizations or from home most of the time.
2. To satisfy **esteem** needs, you need to show people that they are valued by the organization. Public recognition of achievements is a simple yet effective way of doing this. Obviously, people must also feel that they are paid at a level that reflects their skills and experience.
3. Finally, to satisfy **self-realization** needs, you need to give people responsibility for their work, assign them demanding (but not impossible) tasks, and provide a training programme where people can develop their skills. Training is an important motivating influence as people like to gain new knowledge and learn new skills.

Question: Personality types (5 points)

There are three different personality types that must be considered when choosing team members.

- **What are they? (1 point)**

- (Adin) The personality types are task-oriented, self-oriented and interaction oriented. P624

- **Explain how each personality type is motivated while doing the work? (1 point)**

- Task-oriented are motivated by the task they are doing, self-oriented are motivated by personal success and recognition and the interaction-oriented is motivated by presence and actions of coworkers. P624

(Aleks, s.606) Personality type also influences motivation. Bass and Dunteman (1963) classify professionals into three types:

1. Task-oriented people, who are motivated by the work they do. In software engineering, these are people who are motivated by the intellectual challenge of software development.

2. Self-oriented people, who are principally motivated by personal success and recognition. They are interested in software development as a means of achieving their own goals. This does not mean that these people are selfish and think only of their own concerns. Rather, they often have longer-term goals, such as career progression, that motivate them and they wish to be successful in their work to help realize these goals.

3. Interaction-oriented people, who are motivated by the presence and actions of co-workers. As software development becomes more user-centered, interaction- oriented individuals are becoming more involved in software engineering.

- **What personality types would you choose when creating a team? (1 point). Motivate why? (1 point).**

- (Adin) I would choose interaction-oriented because people do a better job when they are happy and social. A team consist of more than people and someone that likes actions of coworkers is a great contribute to the team.

- **In case it is impossible to choose people with the right personality types, how should you manage people so that the organizational and group objectives are met? (1 point)**

- I would control the group so that individual goals doesn't hindrance organisational and group objectives. P626

(Aleks, s.610) It is sometimes impossible to choose a group with complementary personalities. If this is the case, the project manager has to control the group so that individual goals do not take precedence over organizational and group objectives. This control is easier to achieve if all group members participate in each stage of the project. Individual initiative is most likely when group members are given instructions without being aware of the part that their task plays in the overall project.
For example, say a software engineer is given a program design for coding and notices what appears to be possible improvements that could be made to the design. If he or she implements these improvements without understanding the rationale for the original design, any changes, though well intentioned, might have adverse implications for other parts of the system. If all the members of the group are involved in the design from the start, they will understand why design decisions have been made. They may then identify with these decisions rather than oppose them.

Question: Cohesive groups (5 points)
It is important to create a team that has the right balance of personalities, experience and skills. This however does may not always lead to successful and cohesive groups. What does it mean that the group is cohesive? (2 points). What are the benefits of a cohesive group? (3 points)

(Aleks, p.607) In a cohesive group, members think of the group as more important than the individuals who are group members. Members of a well-led, cohesive group are loyal to the group. They identify with group goals and other group members. They attempt to protect the group, as an entity, from outside interference. This makes the group robust and able to cope with problems and unexpected situations.

The benefits of creating a cohesive group are:

1. *The group can establish its own quality standards.* Because these standards are established by consensus, they are more likely to be observed than external standards imposed on the group.

2. *Individuals learn from and support each other.* People in the group learn from each other. Inhibitions caused by ignorance are minimized as mutual learning is encouraged.

3. *Knowledge is shared.* Continuity can be maintained if a group member leaves. Others in the group can take over critical tasks and ensure that the project is not unduly disrupted.

4. *Refactoring and continual improvement is encouraged.* Group members work collectively to deliver high-quality results and fix problems, irrespective of the individuals who originally created the design or program.

Question: Group communications (5 points)
It is essential that group members communicate effectively and efficiently with each other and other project stakeholders. However, the effectiveness and efficiency is influenced by (1) group size, (2) group structure, (3) group composition, (4) the physical work environment, and (5) the available communication channels. Motivate why these issues impact group communication?

- (Adin, Aleks) Group size: when there are many people there is a chance that some of them will never communicate with each other. And there can also be a problem in status. Lower status members tend to find it harder to start a conversation with higher status members.

- Group structure: people in informally structured groups communicate more effectively than people in groups with a formal, hierarchical structure. In hierarchical groups, communications tend to flow up and down the hierarchy. People at the same level may not talk to each other. This is a particular problem in a large project with several development groups.

-Group composition: people with the same personality types may clash and, as a result, communications can be inhibited. Communication is also usually better in mixed-sex groups than in single-sex groups. Women are often more interaction-oriented than men and may act as interaction controllers and facilitators for the group.

- The physical work environment: The room has an impact on people, the color and brightness etc plays an important role. If people are unhappy about their working place they will not do a good work.

- The available communication channels: there are many different forms of communication—face-to-face, e-mail messages, formal documents, telephone, and Web 2.0 technologies such as social networking and wikis. As project teams become increasingly distributed, with team members working remotely, you need to make use of a range of technologies to facilitate communications.

Quality Management

s653 osv

Question: Quality in manufacturing (3 points)
What are the fundamentals of quality management in manufacturing industry (1 point) and why are they not directly comparable with software quality (2 points)?

(Aleks, p. 655) The fundamentals of quality management were established by manufacturing industry in a drive to improve the quality of the products that were being made. As part of this, they developed a definition of 'quality', which was based on conformance with a detailed product specification (Crosby, 1979) and the notion of tolerances. The underlying assumption was that products could be completely specified and procedures could be established that could check a manufactured product against its specification. Of course, products will never exactly meet a specification so some tolerance was allowed. If the product was 'almost right', it was classed as acceptable.
Software quality is not directly comparable with quality in manufacturing. The idea of tolerances is not applicable to digital systems and, for the following reasons, it may be impossible to come to an objective conclusion about whether or not a software system meets its specification:

1. It is difficult to write complete and unambiguous software specifications. Software developers and customers may interpret the requirements in different ways and it may be impossible to reach agreement on whether or not software conforms to its specification.

2. Specifications usually integrate requirements from several classes of stakeholders. These requirements are inevitably a compromise and may not include the requirements of all stakeholder groups. The excluded stakeholders may therefore perceive the system as a poor quality system, even though it implements the agreed requirements.

3. It is impossible to measure certain quality characteristics (e.g., maintainability) directly and so they cannot be specified in an unambiguous way.

Because of these problems, the assessment of software quality is a subjective process where the quality management team has to use their judgment to decide if an acceptable level of quality has been achieved. The quality management team has to consider whether or not the software is fit for its intended purpose.

Question: Process-based quality (3 points)

Give an account of process-based quality and its phases in the manufacturing context (2 points). Is there any clear link between process and product quality in manufacturing? Motivate! (1 point).

(Aleks, pp. 656-657) An assumption that underlies software quality management is that the quality of software is directly related to the quality of the software development process. This again comes from manufacturing systems where product quality is intimately related to the production process. A manufacturing process involves configuring, setting up, and operating the machines involved in the process. Once the machines are operating correctly, product quality naturally follows. You measure the quality of the product and change the process until you achieve the quality level that you need. Figure 24.3 illustrates this process-based approach to achieving product quality.

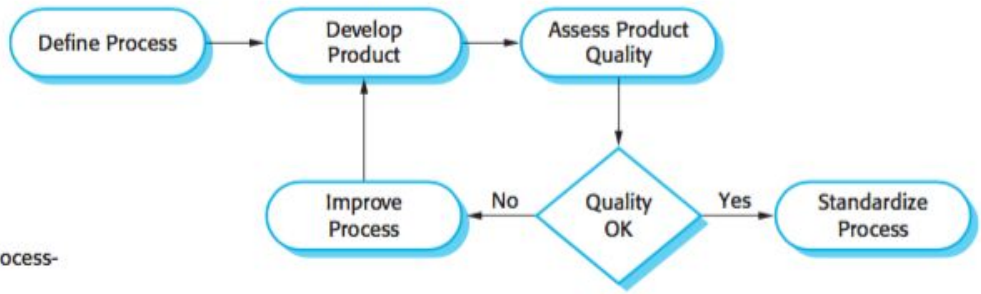


Figure 24.3 Process-based quality

There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor. Once manufacturing systems are calibrated, they can be run again and again to output high-quality products. However, software is not manufactured—it is designed. In software development, therefore, the relationship between process quality and product quality is more complex. Software development is a creative rather than a mechanical process, so the influence of individual skills and experience is significant. External factors, such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.

Question: Software standards (3 points)
There are three reasons why software standards are important. What are they? (3 points)

(Aleks, p. 658) Software standards are important for three reasons:

1. **Standards capture wisdom that is of value to the organization**. They are based on knowledge about the best or most appropriate practice for the company. This knowledge is often only acquired after a great deal of trial and error. Building it into a standard helps the company reuse this experience and avoid previous mistakes.

2. **Standards provide a framework for defining what 'quality' means in a particular setting**. As I have discussed, software quality is subjective, and by using standards you establish a basis for deciding if a required level of quality has been achieved. Of course, this depends on setting standards that reflect user expectations for software dependability, usability, and performance.

3. **Standards assist continuity when work carried out by one person is taken up and continued by another.** Standards ensure that all engineers within an organization adopt the same practices. Consequently, the learning effort required when starting new work is reduced.

Question: Quality management & software development (2 points)

Should or should not quality management and software development be separated? Motivate your answer! (1 point) When is it practically impossible to separate them in smaller companies? Motivate your answer! (1 point)

(Aleks, p.653) Ideally, the quality management team should not be associated with any particular development group, but should rather have organization-wide responsibility for quality management. They should be independent and report to management above the project manager level. The reason for this is that project managers have to maintain the project budget and schedule. If problems arise, they may be tempted to compromise on product quality so that they meet their schedule. An independent quality management team ensures that the organizational goals of quality are not compromised by short-term budget and schedule considerations. In smaller companies, however, this is practically impossible. Quality management and software development are inevitably intertwined with people having both development and quality responsibilities.

Question: Reviews (5 points)

Reviews are quality assurance activities that check the quality of project deliverables. Give an account of a software review process and its phases in a traditional software development context (3 points). What does the review process look like in agile development (1 point)? In what context (traditional or agile) is it more appropriate to use reviews? Motivate your answer (1 point)

(Aleks, pp.664-665) Although there are many variations in the details of reviews, the review process is normally structured into three phases:

1. ***Pre-review activities.*** These are preparatory activities that are essential for the review to be effective. Typically, pre-review activities are concerned with review planning and review preparation. Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed. During review preparation, the team may meet to get an overview of the software to be reviewed. Individual review team members read and understand the software or documents and relevant standards. They work independently to find errors, omissions, and departures from standards. Reviewers may supply written comments on the software if they cannot attend the review meeting.

2. ***The review meeting****.* During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team. The review itself should be relatively short—two hours at most. One team member should chair the review and another should formally record all review decisions and actions to be taken. During the review, the chair is responsible for ensuring that all written comments are considered. The review chair should sign a record of comments and actions agreed during the review.

3. ***Post-review activities****.* After a review meeting has finished, the issues and problems raised during the review must be addressed. This may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents. Sometimes, the problems discovered in a quality review are such that a management review is also necessary to decide if more resources should be made available to correct them. After changes have been made, the review chair may check that the review comments have all been taken into account. Sometimes, a further review will be required to check that the changes made cover all of the previous review comments.

The review process in agile software development is usually informal. In Scrum, for example, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed. In extreme programming, as I discuss in the next section, pair programming ensures that code is constantly being examined and reviewed by another team member. General quality issues are also considered at daily team meetings but XP relies on individuals taking the initiative to improve and refactor code. Agile approaches are not usually standards-driven, so issues of standards compliance are not usually considered.

The lack of formal quality procedures in agile methods means that there can be problems in using agile approaches in companies that have developed detailed quality management procedures. Quality reviews can slow down the pace of software development and they are best used within a plan-driven development process. In a plan-driven process, reviews can be planned and other work scheduled in parallel with them. This is impractical in agile approaches that focus single-mindedly on code development.

Question: Inspections (2 points)

Despite of the well-known cost effectiveness, many software development companies are reluctant to use inspections or peer reviews. Motivate why inspections are effective (1 points) and explain why companies are unwilling to have inspections (1 point).

(Aleks, p.666) Program inspections involve team members from different backgrounds who make a careful, line-by-line review of the program source code. They look for defects and problems and describe these at an inspection meeting.

Most companies that have introduced inspections have found that they are very effective in finding bugs. Fagan (1986) reported that more than 60 percent of the errors in a program can be detected using informal program inspections. Mills et al. (1987) suggest that a more formal approach to inspection, based on correctness arguments, can detect more than 90% of the errors in a program. McConnell (2004) compares unit testing, where the defect detection rate is about 25%, with inspections, where the defect detection rate was 60%. He also describes a number of case studies including an example where the introduction of peer reviews led to a 14% increase in productivity and a 90% decrease in program defects.

In spite of their well-publicized cost effectiveness, many software development companies are reluctant to use inspections or peer reviews. Software engineers with experience of program testing are sometimes unwilling to accept that inspections can be more effective for defect detection than testing. Managers may be suspicious because inspections require additional costs during design and development. They may not wish to take the risk that there will be no corresponding savings in program testing costs.

Configuration and Version Management

Question CM defintion (2 points)

What is configuration management and why is it important?

(Aleks, p.682) Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
Configuration management (CM) is concerned with the policies, processes, and tools for managing changing software systems. You need to manage evolving systems because it is easy to lose track of what changes and component versions have been incorporated into each system version. Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems. There may be several versions under development and in use at the same time. If you don’t have effective configuration management procedures in place, you may waste effort modifying the wrong version of a system, deliver the wrong version of a system to customers, or forget where the software source code for a particular version of the system or component is stored.

Configuration management is useful for individual projects as it is easy for one person to forget what changes have been made. It is essential for team projects where several developers are working at the same time on a software system. Sometimes these developers are all working in the same place but, increasingly, development teams are distributed with members in different locations across the world. The use of a configuration management system ensures that teams have access to information about a system that is under development and do not interfere with each other's work.

Question CM activities (6 points)

There are four principal configuration management activities:

- Change management
- Version management
- System building
- Release management

Choose and describe two of these activities.

(Aleks, p.681) The configuration management of a software system product involves four closely related activities (Figure 25.1):

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

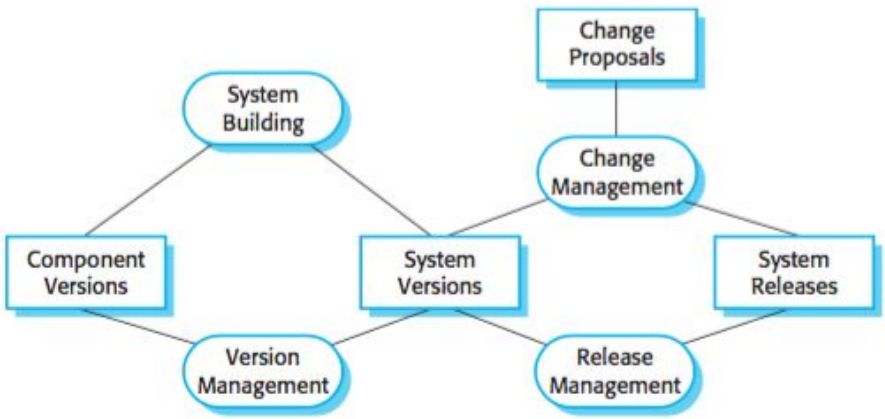


Figure 25.1 Configuration management activities

Question Change management process (5 points)

Describe the change management process.

(Gustav)

The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile, and tracking which components in the system have been changed. Figure 25.3 is a model of a change management process that shows the principal change management activities. There are many variants of this process in use but, to be effective, change management processes should always have a means of checking, costing, and approving changes. This process should come into effect when the software is handed over for release to customers or for deployment within an organization.

The change management process is initiated when a 'customer' completes and submits a change request describing the change required to the system. This could be a bug report, where the symptoms of the bug are described, or a request for additional functionality to be added to the system. Some companies handle bug reports and new requirements separately but, in principle, these are both simply change requests. Change requests may be submitted using a change request form (CRF). I use the term 'customer' here to include any stakeholder who is not part of the development team, so changes may be suggested, for example, by the marketing department in a company.

After a change request has been submitted, it is checked to ensure that it is valid. The checker may be from a customer or application support team or, for internal requests, may be a member of the development team. Checking is necessary because not all change requests require action. If the change request is a bug report, the bug may have already been reported. Sometimes, what people believe to be problems are actually misunderstandings of what the system is expected to do. On occasions, people request features that have already been implemented but which they don't know about. If any of these are true, the change request is closed and the form is updated with the reason for closure. If it is a valid change request, it is then logged as an outstanding request for subsequent analysis.

For valid change requests, the next stage of the process is change assessment and costing. This is usually the responsibility of the development or maintenance team as they can work out what is involved in implementing the change. The impact of the change on the rest of the system must be checked. To do this, you have to identify all of the components affected by the change. If making the change means that further changes elsewhere in the system are needed, this will obviously increase the cost of change implementation. Next, the required changes to the system modules are assessed. Finally, the cost of making the change is estimated, taking into account the costs of changing related components.

Following this analysis, a separate group should then decide if it is costeffective from a business perspective to make the change to the software. For military and government systems, this group is often called the change control board (CCB). In industry, it may be called something like a 'product development group', who is responsible for making decisions about how a software system

should evolve. This group should review and approve all change requests, unless the changes simply involve correcting minor errors on screen displays, webpages, or documents. These small requests should be passed to the development team without detailed analysis, as such an analysis could cost more than implementing the change.
The CCB or product development group considers the impact of the change from a strategic and organizational rather than a technical point of view. It decides whether the change in question is economically justified and prioritizes accepted changes for implementation. Accepted changes are passed back to the development group; rejected change requests are closed and no further action is taken.

Question Change impact (3 points)

Sommerville lists five factors that should be taken into account in deciding whether or not a change should be approved. List and describe three of them.

(Gustav)
Significant factors that should be taken into account in deciding whether or not a change should be approved are:

1. **The consequences of not making the change**
When assessing a change request, you have to consider what will happen if the change is not implemented. If the change is associated with a reported system failure, the seriousness of that failure has to be taken into account. If the system failure causes the system to crash, this is very serious and failure to make the change may disrupt the operational use of the system. On the other hand if the failure has a minor effect, such as incorrect colors on a display, then it is not important to fix the problem quickly, so the change should have a low priority.

2. **The benefits of the change**
Is the change something that will benefit many users of the system or is it simply a proposal that will primarily be of benefit to the change proposer?

3. **The number of users affected by the change**
If only a few users are affected, then the change may be assigned a low priority. In fact, making the change may be inadvisable if it could have adverse effects on the majority of system users.

4. **The costs of making the change**
If making the change affects many system components (hence increasing the chances of introducing new bugs) and/or takes a lot of time to implement, then the change may be rejected, given the elevated costs involved.

5. **The product release cycle**

If a new version of the software has just been released to customers, it may make sense to delay the implementation of the change until the next planned release (see Section 25.3).

Question (1 point)

What is Change Control Board and what is its function?

(Gustav, samma som står två frågor upp)
Following this analysis, a separate group should then decide if it is costeffective from a business perspective to make the change to the software. For military and government systems, this group is often called the change control board (CCB). In industry, it may be called something like a 'product development group', who is responsible for making decisions about how a software system should evolve. This group should review and approve all change requests, unless the changes simply involve correcting minor errors on screen displays, webpages, or documents. These small requests should be passed to the development team without detailed analysis, as such an analysis could cost more than implementing the change.
The CCB or product development group considers the impact of the change from a strategic and organizational rather than a technical point of view. It decides whether the change in question is economically justified and prioritizes accepted changes for implementation. Accepted changes are passed back to the development group; rejected change requests are closed and no further action is taken.

Question (2 points)

Is it ethical for a company to quote a low price for a software contract knowing that the requirements are ambiguous and that they can charge a high price for subsequent changes requested by the customers? Motivate your answer.

(Gustav)
As the cost of a project is only loosely related to the price quoted to a customer, 'pricing to win' is a commonly used strategy. Pricing to win means that a company has some idea of the price that the customer expects to pay and makes a bid for the contract based on the customer's expected price. This may seem unethical and unbusinesslike, but it does have advantages for both the customer and the system provider.
A project cost is agreed on the basis of an outline proposal. Negotiations then take place between client and customer to establish the detailed project specification. This specification is constrained by the agreed cost. The buyer and seller must agree on what is acceptable system functionality. The fixed factor in many projects is not the project requirements but the cost. The requirements may be changed so that the cost is not exceeded.
For example, say a company (OilSoft) is bidding for a contract to develop a fuel delivery system for an oil company that schedules deliveries of fuel to its service stations. There is no detailed requirements document for this system, so OilSoft

estimates that a price of $900,000 is likely to be competitive and within the oil company's budget. After they are granted the contract, OilSoft then negotiates the detailed requirements of the system so that basic functionality is delivered. They then estimate the additional costs for other requirements. The oil company does not necessarily lose here because it has awarded the contract to a company that it can trust. The additional requirements may be funded from a future budget, so that the oil company's budgeting is not disrupted by a high initial software cost.

Question : (8 points) Den här frågan kommer att ställas bara på första ordinarietentan.

During the course project, you have been assigned a specific role, such as for instance, tester, developer, business manager, etc. Your task was to follow the responsibilities assigned to your role. Among many tasks, you were responsible for one or several process models.

From the perspective of your role and from the perspective of **only one** process, do the following:

- Present what role and responsibilities you have had (0 points).
- Describe the process model for which you have been responsible (1 points).
- List and describe the good sides of your process model (1 point).
  - at least 2 good sides
- Motivate why these good sides have been achieved (1 point).
- List and describe the bad sides of your process model (1 point).
  - at least 2 bad sides
- Motivate why these bad sides have been encountered (1 point)!
- Describe and evaluate the solutions that you have used for remedying the bad sides (1 point).
- Evaluate the book from the perspective of the process model you have been responsible for (2 points).
  - Does the book cover the necessary details of the process? Motivate your answer!
  - Does any other literature cover the necessary details of the process? Motivate your answer!
  - Does the book correctly describe the process? Motivate your answer!
  - Does any other literature describe the process? Motivate your answer!
  - What suggestions would you provide to Ian Sommerville to improve his process descriptions? Motivate your suggestions!
  - Finally, does the book properly describe the relationship between your process and other processes? Motivate your answer!

(Hannes, project manager)

s.1-13 i föreläsning 9.
I boken, s 593-617

Present what role and responsibilities you have had (0 points).
Jag har varit Project Manager. I den rollen har jag ansvarat för bland annat:
Planering, schemaläggning, kostnad för, övervakning, granskning, personal, rapportering och riskanalysering av projektet.

Utöver ovanstående ansvar har det varit min uppgift att se till att projektet når deadline inom budget och i tid, samt om möjligt se till att projektet blev lyckat, alltså att produkten mötte kravspecifikationen.

Describe the process model for which you have been responsible (1 points).

Project managerns processmodell kan delas upp i fem övergripande aktiviteter:

Projektplanering
Som nämndes ovanför ansvarar project manager för planering, uppskattning, schemaläggning av projektet och ser till att delegera uppgifter och människor. Den ser även till att projektet görs inom tids och budgetramar.

Rapportering
De ansvarar för rapportering till kunder, ledning och medverkande i projektet, alltså som kommunikationsrör. De måste kunna rapportera om avancerade tekniska saker samt om alla projektrelaterade detaljer, och omvandla abstrakt information till korta och koncisa rapporter.

Riskhantering
De måste kunna bedöma och förutsäga risker, övervaka dessa och sätta in åtgärder där det behövs. Detta görs förstås i nära samarbete med risk manager, om en sådan finns.

Personalledning
De måste välja ut, leda, och hantera de personer som är med i projektteamet. Valet av medlemmar i projektet måste göras med avseende på balans mellan olika personlighetstyper och kompetens.

Skriva förslag/en plan
De ansvar för att skriva en plan för som beskriver hur projektet ska lyckas. Denna plan innefattar detaljer som projektplan, kostnader, risker, affärsfördelar osv, som tillsammans används i början av eller innan ett projekt, till exempel under tender-fasen för att övertyga ledningen eller potentiella kunden om varför de ska utföra projektet.

List and describe the good sides of your process model (1 point).
- **at least 2 good sides**

Projektschemat
People management

- **Motivate why these good sides have been achieved (1 point).**

Genom att ha skapat ett tydligt schema med tasks och dependancies har vi-...

Genom god kommunikation böa bla

- **List and describe the bad sides of your process model (1 point).**
  - **at least 2 bad sides**

Prioritering vid agila har varit svårt att fullfölja

Rapporteing till kunden

- **Motivate why these bad sides have been encountered (1 point)!**

Iterativa har lett till otydliga övergånger mellan iternationerna

Som projektledare har det varit mitts ansvar att rapportera till ledning och till kund..

- **Describe and evaluate the solutions that you have used for remedying the bad sides (1 point).**

Revidera planen

Försökt hålla consistens kommunikation

- **Evaluate the book from the perspective of the process model you have been responsible for (2 points).**
  - **Does the book cover the necessary details of the process? Motivate your answer!**

Ej riskhantering

  - **Does any other literature cover the necessary details of the process? Motivate your answer!**

attbeta i projekt sven eklund

- **Does the book correctly describe the process? Motivate your answer!**

- **Does any other literature describe the process? Motivate your answer!**

  sven eklunds tar endast upp vattenfallsmodellen och inte så mycket agilt

- **What suggestions would you provide to Ian Sommerville to improve his process descriptions? Motivate your suggestions!**

- **Finally, does the book properly describe the relationship between your process and other processes? Motivate your answer!**

  ja, det nära samarbetet med risk- kommunitation quality manager osv...

(Slut på Hannes svar om sin roll)

(Mira beskrivning av sin roll)

- **Presentera vilken roll och ansvarsområden du har haft (0 poäng).**
  Marketing manager

- **Beskriv den processmodell för vilken du har varit ansvarig (1 poäng).**
  Definera en marketing process model. Marketing process model beskriver processen att nå nya kunder men även bevara gamla. Samt att marknadsföra genom att ha koll på konkurrensen och och efterfrågan.

- **Lista och beskriv de goda sidorna av din processmodell (1 poäng).**
  - Känna till konkurrensen. Alltså veta vilka andra företag som finns på marknaden och som vårt företag konkurrerar om kunder med.
  - Förstå kundens form av affärer. Alltså förstå vad kunden ska använda produkten till genom att förstå vad kunden arbetar med och vilka problem som kan uppstå.

- **Motivera varför dessa goda sidor har uppnåtts (1 poäng).**
  Dessa goda sidor har uppnåtts genom en noga utformad kravspecifikation och kundkontakt samt eftersökningar kring vilka liknande företag som finns på marknaden.
- **Lista och beskriv de dåliga sidorna av din processmodell (1 poäng).**
  - Marknadsundersökningen. Undersöka vad marknaden vill ha för produkt.
  - Kund relationer.

- **Motivera varför dessa dåliga sidor har uppstått (1 poäng)!**
  Marknadsundersökningen var svår att genomföra då marknaden bestod av enbart ett annat företag. Kundrelationen var inte dålig, men att vårda kundrelationen kanske kom i lite i andra hand då det bara fanns ett annat företag samt att alla hade lika mycket kontakt med dem.
- **Beskriv och utvärdera de lösningar som du har använt för att åtgärda de dåliga sidorna (1 poäng).**

För att genomföra en trovärdig marknadsundersökning gjorde jag en marknadsundersökning utanför "vår marknad". För att se vad kunder i "riktiga världen vill ha". Kundrelationen löstes genom att veckomöten planerades in.

- **Utvärdera kursboken utifrån den process du har varit ansvarig för (2 poäng).**

Marketing beskrivs på flera ställen i Sommerville men inte just i formen för marketing manager utan mer för marketing group.

- **Har boken täckt de nödvändiga detaljerna i processen? Motivera ditt svar!**

Ja det har den gjort på flera ställen tas det upp olika processer för marketing och vad man ska tänka på och hur marketing påverkar andra processer.

- **Har annan literatur täckt de nödvändiga detaljerna i processen? Motivera ditt svar!**

Jag har inte använt annan litteratur för att ta reda på detaljerna i marketing processen utan använt sommerville och föreläsningsanteckningarna.

- **Har boken korrekt beskrivit processen? Motivera ditt svar!**

I jämförelse med vad som gåtts igenom på föreläsningar och vad jag läst mig till i boken så har processen beskrivits korrekt. Exempelvis hur marketing management måste hantera förändringar på produkten.

- **Har en annan literatur korrekt beskrivit processen? Motivera ditt svar!**
- **Vilka förslag skulle du ge till Ian Sommerville för att förbättra sina processbeskrivningar? Motivera dina förslag!**

Det finns noga beskrivningar om många processer, just marketing är inte lika noga beskrivet utan uppkommer i och med andra processer så det kan vara svårt att få en tydlig bild. Men ska man titta på processbeskrivningarna som helhet så är de tydliga.

- **Slutligen, har boken beskrivit relationen mellan din process och andra processer ordentligt? Motivera ditt svar!**

Ja det beskrivs hur olika processer måste samarbeta. Exempelvis mellan quality management och marketing. Även processerna marketing och test manager för att testa och se om och hur man kan marknadsföra produkten.

(Alexander, forstättning projectM, bullshitande):

List and describe the good sides of your process model (1 point).

- **at least 2 good sides**

Projekplanering: att i god tid innan projektets start planera, schemalägga det kommande projektet.

Personalledning: Att välja ut de personer som är med i projektteamet med hänsyn till deras kompetens och personlighetstyper. Samt vad de är motiverade till att jobba med.

Motivate why these good sides have been achieved (1 point).
Genom att ha gjort en god projektplanering, hade vi en överblick på vad som skulle göras under de dagar vi jobbade med projektet. Detta underlättade enormt mkt så att vi kunde hålla oss inom vissa tidsramar och deadlines.

Att välja ut de personer med hänsyn till deras kompetens och personlighetstyper gjorde att vi fick en mycket trevligare arbetsmiljö. Alla var nöjda med sina uppgifter och bidrog till att vi hjälpte varandra mer. Vi fick också chansen att kunna ta lärdom av varandras kompetenser.

List and describe the bad sides of your process model (1 point).
- **at least 2 bad sides**

Motivate why these bad sides have been encountered (1 point)!
Describe and evaluate the solutions that you have used for remedying the bad sides (1 point).
En dålig sida av processen för vårt projekt var att man skulle delegera uppgifter till alla personer inom projektgruppen. Eftersom vi var så få som medverkande under projektet så kändes det onödigt att ha sådan formell struktur på delegering av uppgifter. Istället kom vi fram till att vi skulle ha friare roller och öppnare diskussioner när vi skulle delegera uppgifter.

En annan dålig sida var att vi ibland hade bristande kommunikation när vi blev klara med systemdelar innan tidsplaneringen. Detta uppkom eftersom vi inte hade diskuterat vad som skulle göras när detta uppstod. Vi löste detta genom att komma överens om hur vi skulle gå tillväga varje gång vi blev klara innan tidsplanen. Ska vi ta ledigt? börja på nästa projekt osv. Innan så satt vi kanske onödigt lång tid med samma systemdel.

If you happen to be CEO, then you are welcome to choose any role.

Under kursprojektet, har du tilldelats en specifik roll, till exempel testare, utvecklare, affärschef osv. Din uppgift var att följa de ansvarsområden som givits till din roll. Bland många uppgifter, var du ansvarig för en eller flera processmodeller.

Ur din rolls och utifrån endast en process perspektiv, gör följande:
- Presentera vilken roll och ansvarsområden du har haft (0 poäng).
- Beskriv den processmodell för vilken du har varit ansvarig (1 poäng).
- Lista och beskriv de goda sidorna av din processmodell (1 poäng).
    - minst 2 goda sidor
- Motivera varför dessa goda sidor har uppnåtts (1 poäng).
- Lista och beskriv de dåliga sidorna av din processmodell (1 poäng).
    - minst 2 dåliga sidor
- Motivera varför dessa dåliga sidor har uppstått (1 poäng)!

- Beskriv och utvärdera de lösningar som du har använt för att åtgärda de dåliga sidorna (1 poäng).
- Utvärdera kursboken utifrån den process du har varit ansvarig för (2 poäng).
- Har boken täckt de nödvändiga detaljerna i processen? Motivera ditt svar!
- Har annan literatur täckt de nödvändiga detaljerna i processen? Motivera ditt svar!
- Har boken korrekt beskrivit processen? Motivera ditt svar!
- Har en annan literatur korrekt beskrivit processen? Motivera ditt svar!
- Vilka förslag skulle du ge till Ian Sommerville för att förbättra sina processbeskrivningar? Motivera dina förslag!
- Slutligen, har boken beskrivit relationen mellan din process och andra processer ordentligt? Motivera ditt svar!

Om du råkar vara VD, så är du välkommen att välja vilken roll som helst.