Software Engineering
CSE 307

Md Awsaf Alam

August 17, 2018

Contents

Chapter 1

CH-2 Software Processes

1.1 Topics covered

- Software process models
- Process activities
- Coping with change
- The Rational Unified Process(*An example of a modern software process*).

1.2 The software process

- A structured set of activities required to develop a software system.
- Many different software processes but all involve:
  1. Specification − defining what the system should do;
  2. Design and implementation − defining the organization of the system and implementing the system;
  3. Validation − checking that it does what the customer wants;
  4. Evolution − changing the system in response to changing customer needs.
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

1.2.1 Software process descriptions

- When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.

- Process descriptions may also include:
    1. Products, which are the outcomes of a process activity;
    2. Roles, which reflect the responsibilities of the people involved in the process;
    3. Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.

1.3 Plan-driven and agile processes

- Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- In practice, most practical processes include elements of both plan-driven and agile approaches.
- There are no right or wrong software processes.

1.4 Software process models

1. **The waterfall model** Plan-driven model. Separate and distinct phases of specification and development.
2. **Incremental development** Specification, development and validation are interleaved. May be plan-driven or agile.
3. **Reuse-oriented software engineering** The system is assembled from existing components. May be plan-driven or agile.
4. In practice, most large systems are developed using a process that incorporates elements from all of these models.

1.4.1 The Waterfall Model

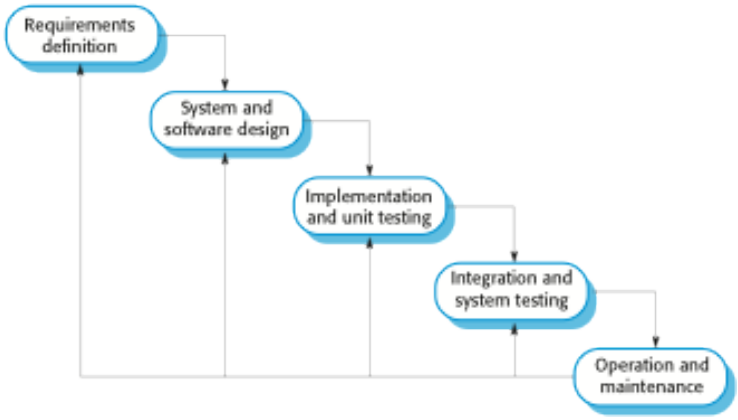

Figure 1.1:

Waterfall model phases

- There are separate identified phases in the waterfall model:
  1. Requirements analysis and definition
  2. System and software design
  3. Implementation and unit testing
  4. Integration and system testing
  5. Operation and maintenance
- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

Waterfall model problems

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
  - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
  - Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
  In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work. Incremental development

1.4.2 Incremental development

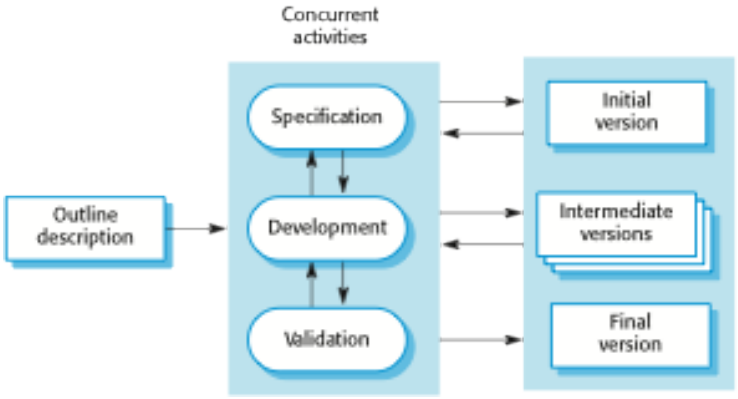

Figure 1.2:

Incremental development benefits

- 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.

- 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.

- More rapid delivery and deployment of useful software to the customer is possible.
  Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental development problems

- 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.

- 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.

1.4.3 Reuse-oriented software engineering

- Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- Process stages
  1. Component analysis
  2. Requirements modification
  3. System design with reuse
  4. Development and integration.
- Reuse is now the standard approach for building many types of business system
  Reuse covered in more depth in Chapter 16.

Reuse-oriented software engineering

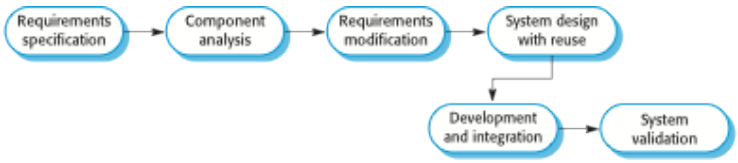

Figure 1.3:

1.5 Types of software component

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

1.6 Process activities

- Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.

- 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 inter-leaved.

1.7 Software specification

- The process of establishing what services are required and the constraints on the system's operation and development.
- Requirements engineering process

    1. Feasibility study
       -Is it technically and financially feasible to build the system?

    2. Requirements elicitation and analysis -What do the system stakeholders require or expect from the system?

    3. Requirements specification -Defining the requirements in detail

    4. Requirements validation -Checking the validity of the requirements

1.8 The requirements engineering process

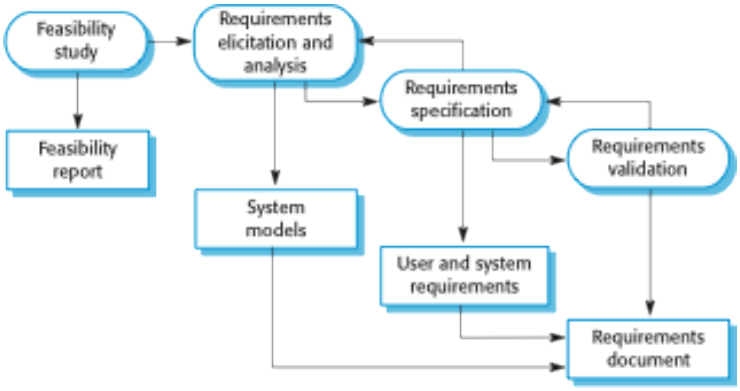

Figure 1.4:

1.9 Software design and implementation

- The process of converting the system specification into an executable system.
- Software design
  −Design a software structure that realises the specification;
- Implementation
  −Translate this structure into an executable program;
- The activities of design and implementation are closely related and may be inter-leaved.

1.10 A general model of the design process

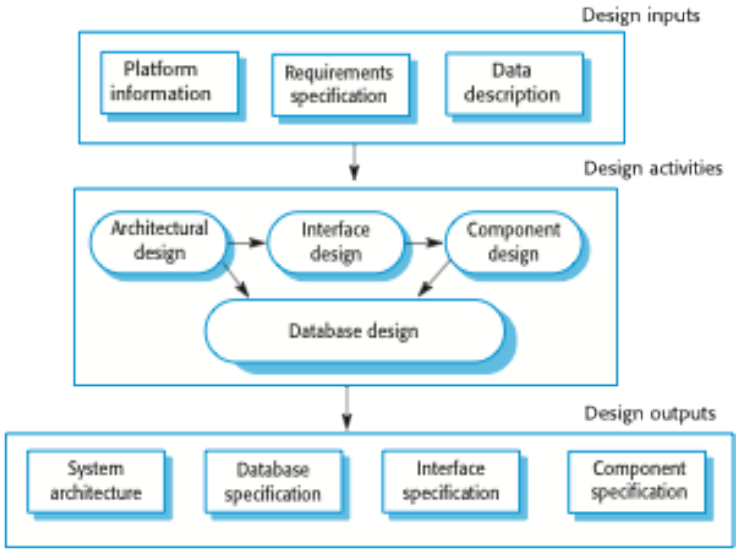

Figure 1.5:

1.11 Design activities

- 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.
- Interface design, where you define the interfaces between system components.
- Component design, where you take each system component and design how it will operate.
- Database design, where you design the system data structures and how these are to be represented in a database.

1.12 Software validation

- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking and review processes and system testing.

- System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- Testing is the most commonly used V & V activity.

1.13 Stages of testing

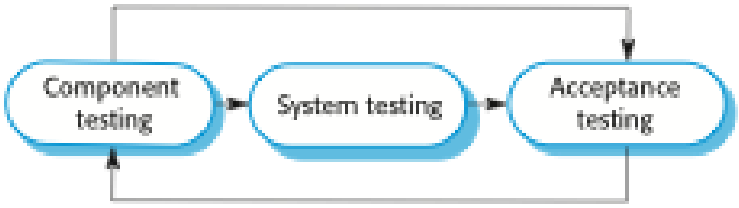

Figure 1.6:

1.13.1 Testing stages

- Development or component testing
- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.
- System testing
- Testing of the system as a whole. Testing of emergent properties is particularly important.
- Acceptance testing
- Testing with customer data to check that the system meets the customer's needs.

1.13.2 Testing phases in a plan-driven software process

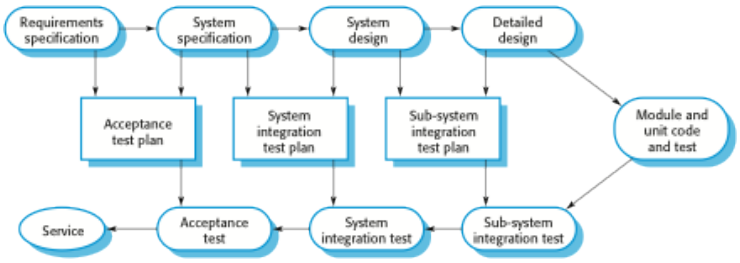

Figure 1.7:

1.14 Software evolution

- Software is inherently flexible and can change.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

1.15 System evolution

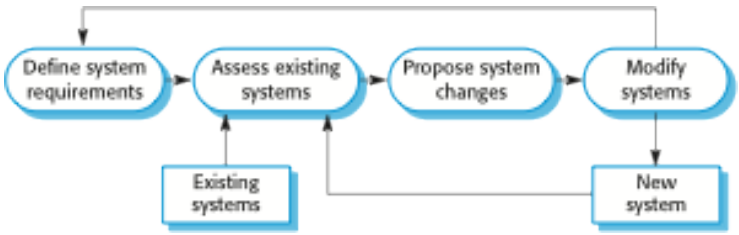

Figure 1.8:

1.16 Key points

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- General process models describe the organization of software processes. Examples of these general models include the 'waterfall' model, incremental development, and reuse-oriented development.
- Requirements engineering is the process of developing a software specification.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.

1.17 Coping with change

- Change is inevitable in all large software projects.
- Business changes lead to new and changed system requirements
- New technologies open up new possibilities for improving implementations
- Changing platforms require application changes
- Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality

1.18 Reducing the costs of rework

- Change avoidance, where the software process includes activities that can anticipate possible changes before significant rework is required.
- For example, a prototype system may be developed to show some key features of the system to customers.
- Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost.
- This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have be altered to incorporate the change.

1.19 Software prototyping

- A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- A prototype can be used in:
- The requirements engineering process to help with requirements elicitation and validation;
- In design processes to explore options and develop a UI design;
- In the testing process to run back-to-back tests.

1.19.1 Benefits of prototyping

- Improved system usability.
- A closer match to users' real needs.
- Improved design quality.
- Improved maintainability.
- Reduced development effort.

1.19.2 The process of prototype development

1.19.3 Prototype development

- May be based on rapid prototyping languages or tools
- May involve leaving out functionality
- Prototype should focus on areas of the product that are not well-understood;

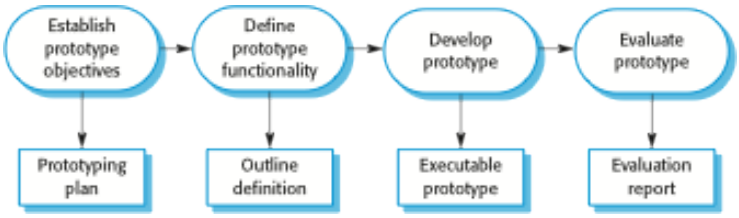

Figure 1.9:

- Error checking and recovery may not be included in the prototype;
- Focus on functional rather than non-functional requirements such as reliability and security

1.19.4 Throw-away prototypes

- Prototypes should be discarded after development as they are not a good basis for a production system:
- It may be impossible to tune the system to meet non-functional requirements;
- Prototypes are normally undocumented;
- The prototype structure is usually degraded through rapid change;
- The prototype probably will not meet normal organisational quality standards.

1.20 Incremental delivery

- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- User requirements are prioritised and the highest priority requirements are included in early increments.
- Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

1.21 Incremental development and delivery

- Incremental development
- Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;

- Normal approach used in agile methods;
- Evaluation done by user/customer proxy.
- Incremental delivery
- Deploy an increment for use by end-users;
- More realistic evaluation about practical use of software;
- Difficult to implement for replacement systems as increments have less functionality than the system being replaced.

1.22 Incremental delivery

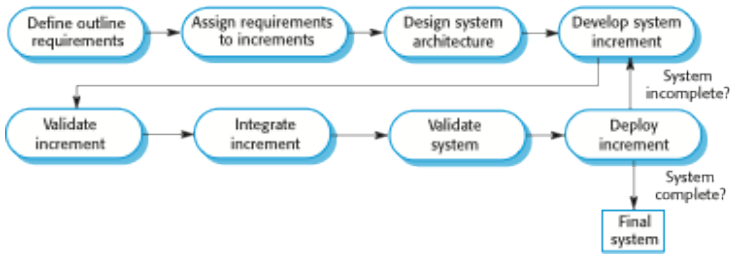

Figure 1.10:

1.23 Incremental delivery advantages

- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.

1.24 Incremental delivery problems

- Most systems require a set of basic facilities that are used by different parts of the system.
- As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.

- The essence of iterative processes is that the specification is developed in conjunction with the software.
- However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

1.25 Boehm's spiral model

- Process is represented as a spiral rather than as a sequence of activities with backtracking.
- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- Risks are explicitly assessed and resolved throughout the process.

1.26 Boehm's spiral model of the software process

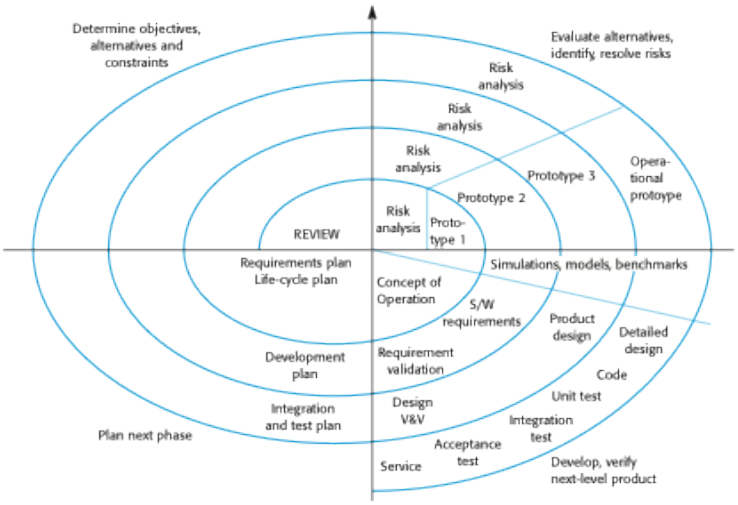

Figure 1.11:

1.27 Spiral model sectors

- Objective setting

- Specific objectives for the phase are identified.
- Risk assessment and reduction
- Risks are assessed and activities put in place to reduce the key risks.
- Development and validation
- A development model for the system is chosen which can be any of the generic models.
- Planning
- The project is reviewed and the next phase of the spiral is planned.

1.28 Spiral model usage

- Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development.
- In practice, however, the model is rarely used as published for practical software development.

1.29 The Rational Unified Process

- A modern generic process derived from the work on the UML and associated process.
- Brings together aspects of the 3 generic process models discussed previously.
- Normally described from 3 perspectives
- A dynamic perspective that shows phases over time;
- A static perspective that shows process activities;
- A practive perspective that suggests good practice.

1.30 Phases in the Rational Unified Process

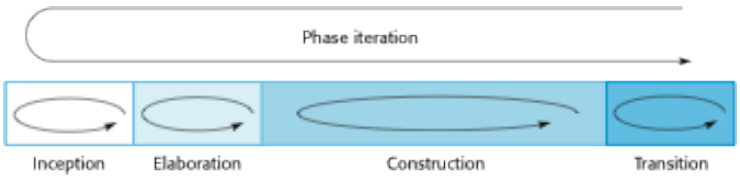

Figure 1.12:

1.31 RUP phases

- Inception
- Establish the business case for the system.
- Elaboration
- Develop an understanding of the problem domain and the system architecture.
- Construction
- System design, programming and testing.
- Transition
- Deploy the system in its operating environment.

1.32 RUP iteration

- In-phase iteration
- Each phase is iterative with results developed incrementally.
- Cross-phase iteration
- As shown by the loop in the RUP model, the whole set of phases may be enacted incrementally.

1.33 Static workflows in the Rational Unified Process

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.

1.34 Static workflows in the Rational Unified Process

Workflow	Description
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow managed changes to the system (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

1.35 RUP good practice

- Develop software iteratively
- Plan increments based on customer priorities and deliver highest priority increments first.
- Manage requirements
- Use component-based architectures
- Organize the system architecture as a set of reusable components.

1.36 RUP good practice

- Visually model software
- Use graphical UML models to present static and dynamic views of the software.
- Verify software quality

- Ensure that the software meet's organizational quality standards.
- Control changes to software
- Manage software changes using a change management system and configuration management tools.

1.37 Key points

- Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design.
- Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- The Rational Unified Process is a modern generic process model that is organized into phases (inception, elaboration, construction and transition) but separates activities (requirements, analysis and design, etc.) from these phases.

Chapter 2

CH-3 Agile Software Development

2.1 Topics covered

- Agile methods
- Plan-driven and agile development
- Extreme programming
- Agile project management
- Scaling agile methods

2.2 Rapid software development

- Rapid development and delivery is now often the most important requirement for software systems
- Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
- Software has to evolve quickly to reflect changing business needs.
- Rapid software development
- Specification, design and implementation are inter-leaved
- System is developed as a series of versions with stakeholders involved in version evaluation
- User interfaces are often developed using an IDE and graphical toolset.

2.3 Agile methods

- Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
- Focus on the code rather than the design
- Are based on an iterative approach to software development
- Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

2.4 Agile manifesto

- We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
- Individuals and interactions over processes and tools Working software over comprehensive documentation Customer collaboration over contract negotiation Responding to change over following a plan
- That is, whilethere is value in the items on value the items on the left more.

2.5 The Principles of agile methods

2.6 Agile method applicability

- Product development where a software company is developing a small or medium-sized product for sale.
- Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.
- Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems.

Principles	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

2.7 Problems with agile methods

- It can be difficult to keep the interest of customers who are involved in the process.
- Team members may be unsuited to the intense involvement that characterises agile methods.
- Prioritising changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.

2.8 Agile methods and software maintenance

- Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- Two key issues:

- Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
- Can agile methods be used effectively for evolving a system in response to customer change requests?
- Problems may arise if original development team cannot be maintained.

2.9 Plan-driven and agile development

- Plan-driven development
- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.
- Agile development
- Specification, design, implementation and testing are inter-leaved and the outputs from the development process are decided through a process of negotiation during the software development process.

2.10 Plan-driven and agile specification

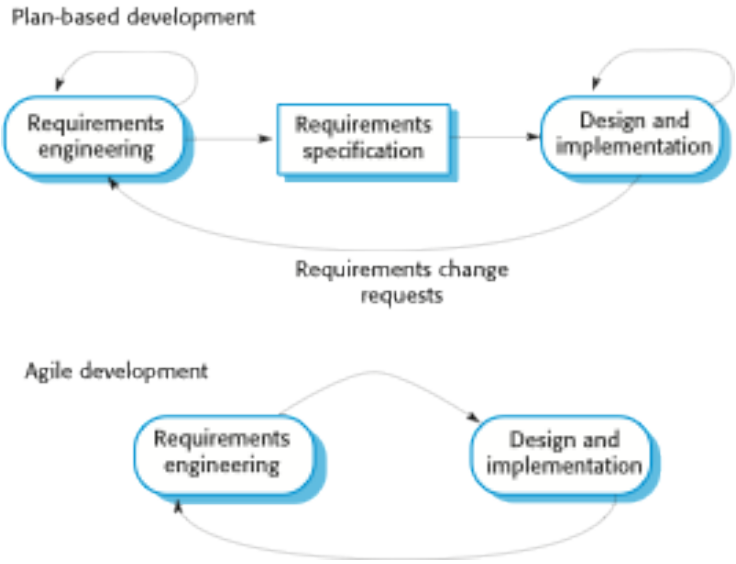

Figure 2.1:

2.11 Technical, human, organizational issues

- Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:

- Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.

- Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.

- How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

2.12 Technical, human, organizational issues

- What type of system is being developed?
  −Plan-driven approaches may be required for systems that require a lot of analysis before implementation (e.g. real-time system with complex timing requirements).
- What is the expected system lifetime?
  −Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team.
- What technologies are available to support system development?
  −Agile methods rely on good tools to keep track of an evolving design
- How is the development team organized?
  −If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.

2.13 Technical, human, organizational issues

- Are there cultural or organizational issues that may affect the system development?
  −Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- How good are the designers and programmers in the development team?
  −It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code
- Is the system subject to external regulation?
  −If a system has to be approved by an external regulator (e.g. the FAA approve software that is critical to the operation of an aircraft) then you will probably be required to produce detailed documentation as part of the system safety case.

2.14 Extreme programming

- Perhaps the best-known and most widely used agile method.
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
- New versions may be built several times per day;
- Increments are delivered to customers every 2 weeks;

- All tests must be run for every build and the build is only accepted if tests run successfully.

2.15 XP and agile principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- People not process through pair programming, collective ownership and a process that avoids long working hours.
- Change supported through regular system releases.
- Maintaining simplicity through constant refactoring of code.

2.16 The extreme programming release cycle

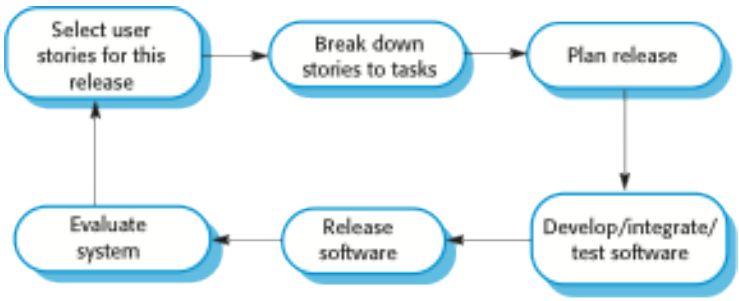

Figure 2.2:

2.17 Extreme programming practices (a)

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more. Test-first development & An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

2.18 Extreme programming practices (b)

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

2.19 Requirements scenarios

- In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- User requirements are expressed as scenarios or user stories.
- These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

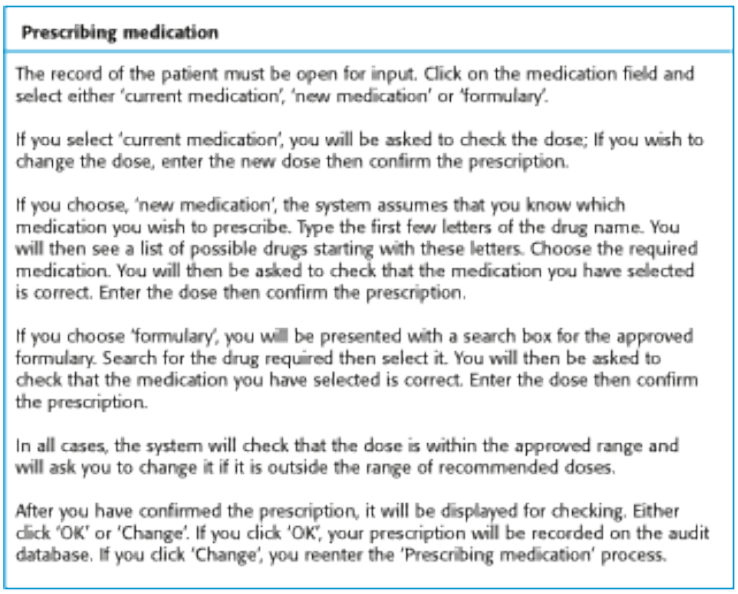

Figure 2.3:

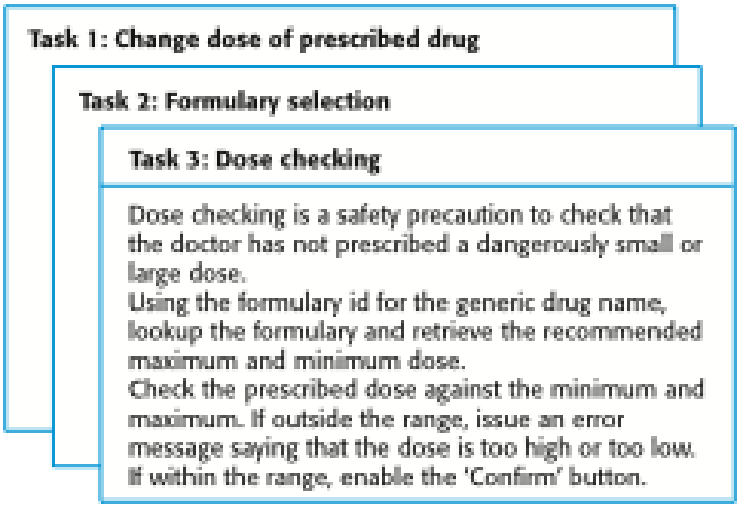

Figure 2.4:

2.20 A 'prescribing medication' story

2.21 Examples of task cards for prescribing medication

2.22 XP and change

- Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs

later in the life cycle.

- XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.

2.23 Refactoring

- Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- This improves the understandability of the software and so reduces the need for documentation.
- Changes are easier to make because the code is well-structured and clear.
- However, some changes requires architecture refactoring and this is much more expensive.

2.24 Examples of refactoring

- Re-organization of a class hierarchy to remove duplicate code.
- Tidying up and renaming attributes and methods to make them easier to understand.
- The replacement of inline code with calls to methods that have been included in a program library.

2.25 Key points

- Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads and producing high-quality code. They involve the customer directly in the development process.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team and the culture of the company developing the system.
- Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement and customer participation in the development team.

2.26 Testing in XP

- Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- XP testing features:
- Test-first development.
- Incremental test development from scenarios.
- User involvement in test development and validation.
- Automated test harnesses are used to run all component tests each time that a new release is built.

2.27 Test-first development

- Writing tests before code clarifies the requirements to be implemented.
- Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
- Usually relies on a testing framework such as Junit.
- All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors. Customer involvement
- The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

2.28 Test case description for dose checking

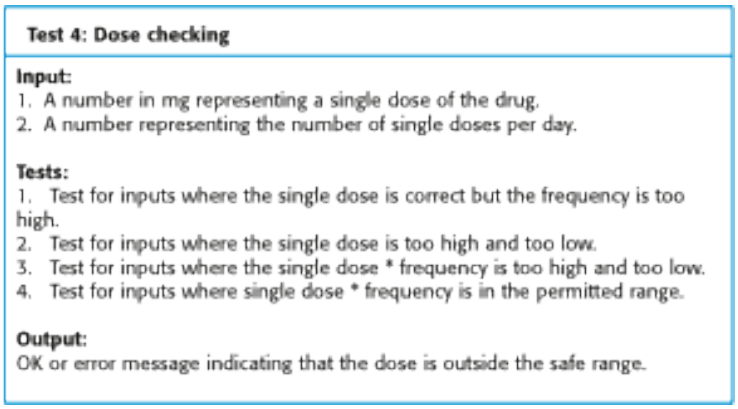

Figure 2.5:

2.29 Test automation

- Test automation means that tests are written as executable components before the task is implemented
- These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- As testing is automated, there is always a set of tests that can be quickly and easily executed
- Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

2.30 XP testing difficulties

- Programmers prefer programming to testing and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.

- It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.

2.31 Pair programming

- In XP, programmers work in pairs, sitting together to develop code.
- This helps develop common ownership of code and spreads knowledge across the team.
- It serves as an informal review process as each line of code is looked at by more than 1 person.
- It encourages refactoring as the whole team can benefit from this.
- Measurements suggest that development productivity with pair programming is similar to that of two people working independently.

2.32 Pair programming

- In pair programming, programmers sit together at the same workstation to develop the software.
- Pairs are created dynamically so that all team members work with each other during the development process.
- The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- Pair programming is not necessarily inefficient and there is evidence that a pair working together is more efficient than 2 programmers working separately.

2.33 Advantages of pair programming

- It supports the idea of collective ownership and responsibility for the system.
- Individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
- It acts as an informal review process because each line of code is looked at by at least two people.
- It helps support refactoring, which is a process of software improvement.
- Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

2.34 Agile project management

- The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.
- The standard approach to project management is plan-driven. Managers draw up a plan for the project showing what should be delivered, when it should be delivered and who will work on the development of the project deliverables.
- Agile project management requires a different approach, which is adapted to incremental development and the particular strengths of agile methods.

2.35 Scrum

- The Scrum approach is a general agile method but its focus is on managing iterative development rather than specific agile practices.
- There are three phases in Scrum.
- The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
- This is followed by a series of sprint cycles, where each cycle develops an increment of the system.
- The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.

2.36 The Scrum process

Figure 2.6:

2.37 The Sprint cycle

- Sprints are fixed length, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
- The starting point for planning is the product backlog, which is the list of work to be done on the project.
- The selection phase involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.
- Once these are agreed, the team organize themselves to develop the software.During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called 'Scrum master'.
- The role of the Scrum master is to protect the development team from external distractions.
- At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

2.38 Teamwork in Scrum

- The 'Scrum master' is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- The whole team attends short daily meetings where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
- This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

2.39 Scrum benefits

- The product is broken down into a set of manageable and understandable chunks.
- Unstable requirements do not hold up progress.
- The whole team have visibility of everything and consequently team communication is improved.
- Customers see on-time delivery of increments and gain feedback on how the product works.

- Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

2.40 Scaling agile methods

- Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.

- It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.

- Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

2.41 Large systems development

- Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.

- Large systems are 'brownfield systems', that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development.

- Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.

2.42 Large system development

- Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.

- Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.

- Large systems usually have a diverse set of stakeholders.It is practically impossible to involve all of these different stakeholders in the development process.

2.43 Scaling out and scaling up

- 'Scaling up' is concerned with using agile methods for developing large software systems that cannot be developed by a small team.
- 'Scaling out' is concerned with how agile methods can be introduced across a large organization with many years of software development experience.
- When scaling agile methods it is essential to maintain agile fundamentals
- Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.

2.44 Scaling up to large systems

- For large systems development, it is not possible to focus only on the code of the system. You need to do more up-front design and system documentation
- Cross-team communication mechanisms have to be designed and used. This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress.
- Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.

2.45 Scaling out to large companies

- Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.
- There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

2.46 Key points

- A particular strength of extreme programming is the development of automated tests before a program feature is created. All tests must successfully execute when an increment is integrated into a system.
- The Scrum method is an agile method that provides a project management framework. It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- Scaling agile methods for large systems is difficult. Large systems need up-front design and some documentation.

Chapter 3

CH-6 Architectural Design

3.1 Topics covered

- Architectural design decisions
- Architectural views
- Architectural patterns
- Application architectures

3.2 Software architecture

- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is architectural design.
- The output of this design process is a description of the software architecture.

3.3 Architectural design

- An early stage of the system design process.
- Represents the link between specification and design processes.
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.

3.4 The architecture of a packing robot control system

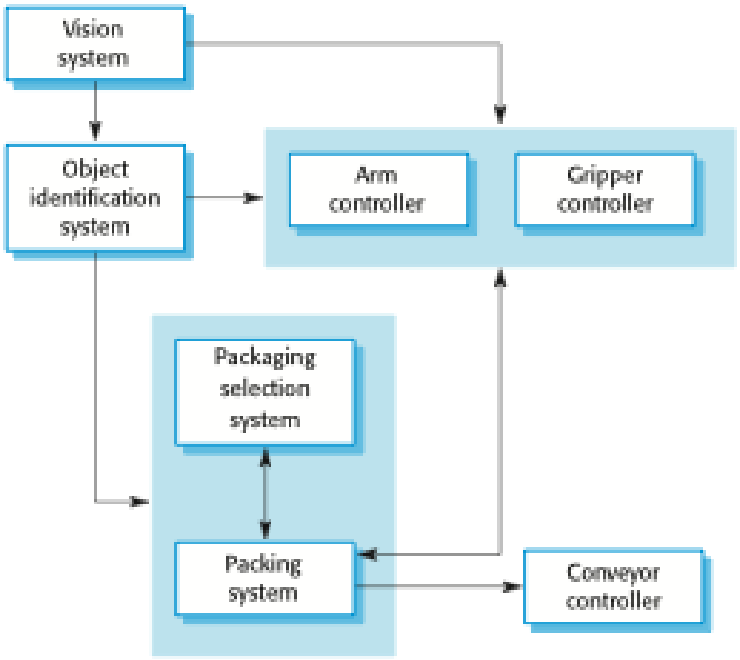

Figure 3.1:

3.5 Architectural abstraction

- Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

3.6 Advantages of explicit architecture

- Stakeholder communication

- Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
- Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
- The architecture may be reusable across a range of systems
- Product-line architectures may be developed. Architectural representations
- Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- Depends on the use of architectural models.The requirements for model semantics depends on how the models are used. Box and line diagrams
- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

3.7 Use of architectural models

- As a way of facilitating discussion about the system design
- A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- As a way of documenting an architecture that has been designed
- The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

3.8 Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

3.9 Architectural design decisions

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

3.10 Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- The architecture of a system may be designed around one of more architectural patterns or 'styles'.
- These capture the essence of an architecture and can be instantiated in different ways.
- Discussed later in this lecture.

3.11 Architecture and system characteristics

- Performance
- Localise critical operations and minimise communications. Use large rather than fine-grain components.

- Security
- Use a layered architecture with critical assets in the inner layers.
- Safety
- Localise safety-critical features in a small number of sub-systems.
- Availability
- Include redundant components and mechanisms for fault tolerance.
- Maintainability
- Use fine-grain, replaceable components.

3.12 Architectural views

- What views or perspectives are useful when designing and documenting a system's architecture?
- What notations should be used for describing architectural models?
- Each architectural model only shows one view or perspective of the system.
- It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

3.13 4 + 1 view model of software architecture

- A logical view, which shows the key abstractions in the system as objects or object classes.
- A process view, which shows how, at run-time, the system is composed of interacting processes.
- A development view, which shows how the software is decomposed for development.
- A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
- Related using use cases or scenarios (+1)

3.14 Architectural patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.

3.15 The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

3.16 The organization of the Model-View-Controller

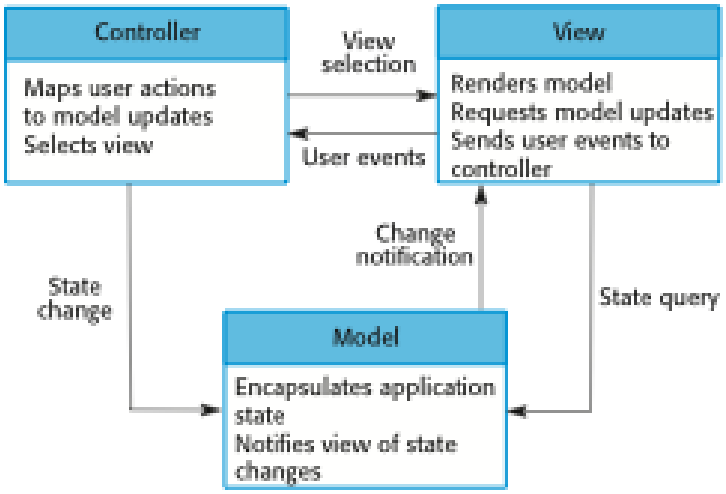

Figure 3.2:

3.17 Web application architecture using the MVC pattern

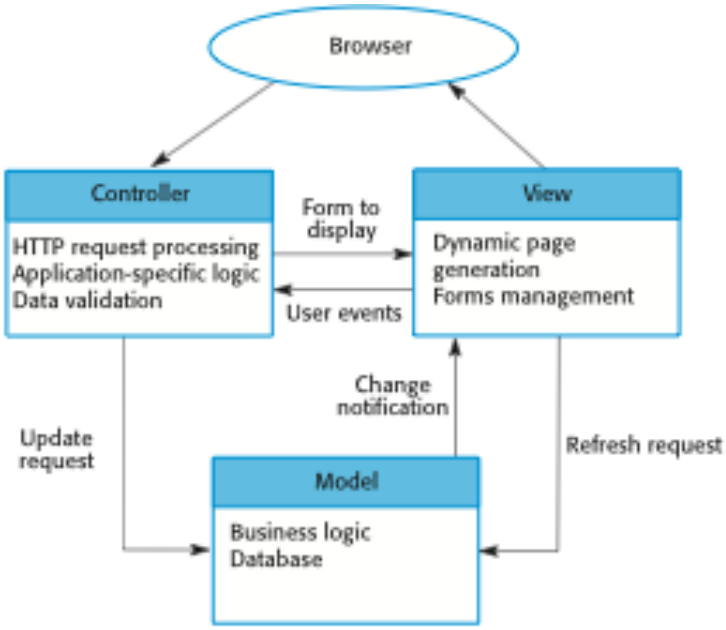

Figure 3.3:

3.18 Layered architecture

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

3.19 The Layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

3.20 A generic layered architecture

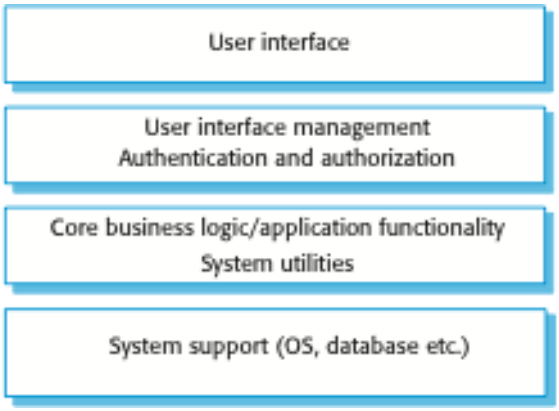

Figure 3.4:

3.21 The architecture of the LIBSYS system

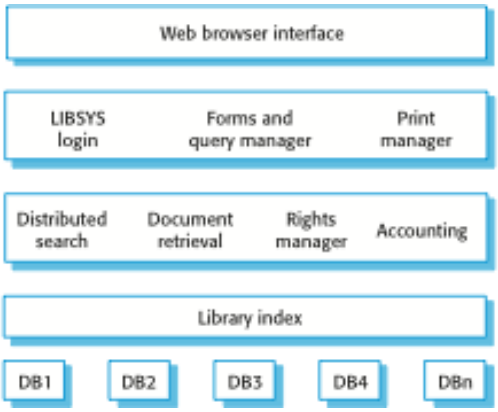

Figure 3.5:

3.22 Key points

- A software architecture is a description of how a software system is organized.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.

- Architectures may be documented from several different perspectives or viewssuch as a conceptual view, a logical view, a process view, and a development view.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.

3.23 Repository architecture

- Sub-systems must exchange data. This may be done in two ways:
- Shared data is held in a central database or repository and may be accessed by all sub-systems;
- Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

3.24 A repository architecture for an IDE

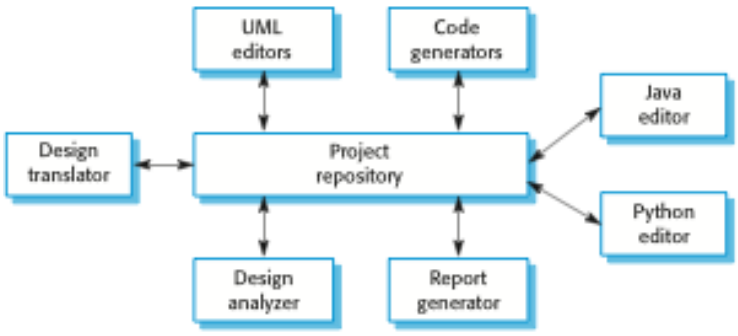

Figure 3.6:

3.25 The Repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

3.26 Client-server architecture

- Distributed system model which shows how data and processing is distributed across a range of components.
- Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.

- Network which allows clients to access servers.

3.27 The Client–server pattern

Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

3.28 A client–server architecture for a film library

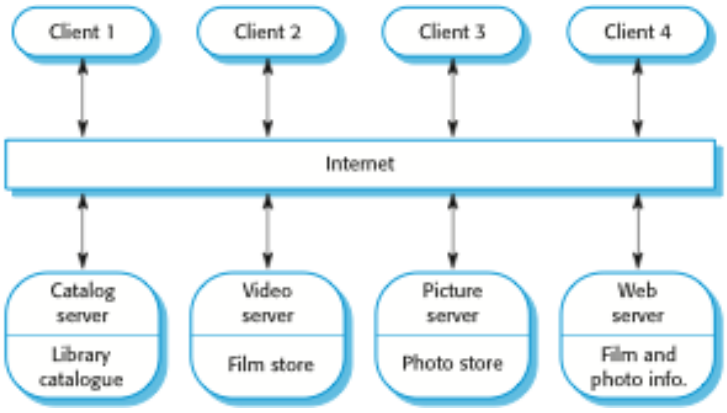

Figure 3.7:

3.29 Pipe and filter architecture

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

3.30 The pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

3.31 An example of the pipe and filter architecture

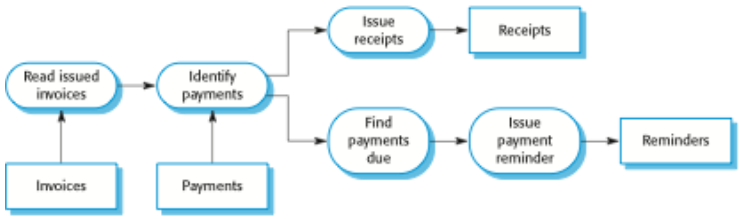

Figure 3.8:

3.32 Application architectures

- Application systems are designed to meet an organisational need.
- As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

3.33 Use of application architectures

- As a starting point for architectural design.
- As a design checklist.
- As a way of organising the work of the development team.
- As a means of assessing components for reuse.
- As a vocabulary for talking about application types.

3.34 Examples of application types

- Data processing applications
- Data driven applications that process data in batches without explicit user intervention during the processing.
- Transaction processing applications
- Data-centred applications that process user requests and update information in a system database.
- Event processing systems
- Applications where system actions depend on interpreting events from the system's environment.
- Language processing systems
- Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.

3.35 Application type examples

- Focus here is on transaction processing and language processing systems.
- Transaction processing systems
- E-commerce systems;
- Reservation systems.
- Language processing systems
- Compilers;
- Command interpreters.

3.36 Transaction processing systems

- Process user requests for information from a database or requests to update the database.
- From a user perspective a transaction is:
- Any coherent sequence of operations that satisfies a goal;
- For example - find the times of flights from London to Paris.
- Users make asynchronous requests for service which are then processed by a transaction manager.

3.37 The structure of transaction processing applications

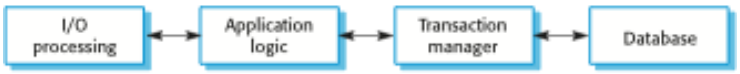

Figure 3.9:

3.38 The software architecture of an ATM system

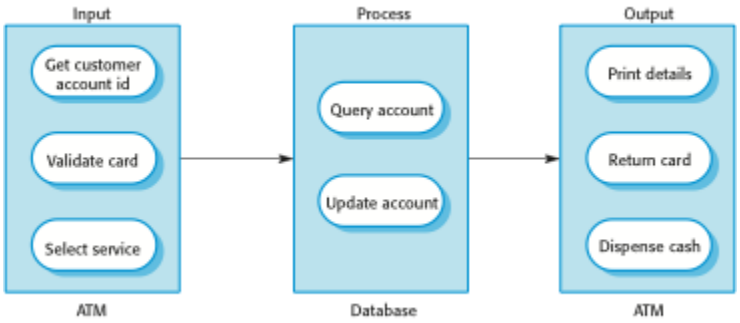

Figure 3.10:

3.39 Information systems architecture

- Information systems have a generic architecture that can be organised as a layered architecture.
- These are transaction-based systems as interaction with these systems generally involves database transactions.
- Layers include:
- The user interface
- User communications
- Information retrieval
- System database

3.40 Layered information system architecture

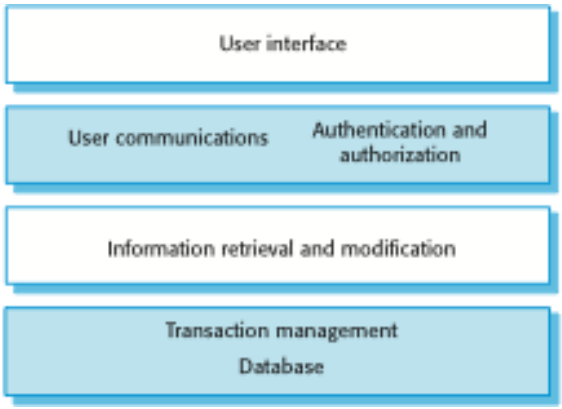

Figure 3.11:

3.41 The architecture of the MHC-PMS

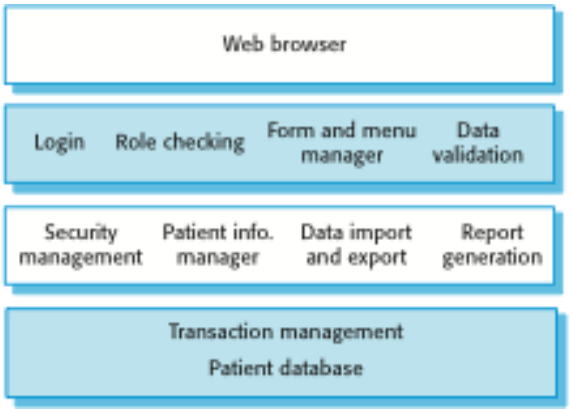

Figure 3.12:

3.42 Web-based information systems

- Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
- For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
- In an e-commerce system, the application-specific layer includes additional functionality supporting a 'shopping cart' in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

3.43 Server implementation

- These systems are often implemented as multi-tier client server/architectures (discussed in Chapter 18)
- The web server is responsible for all user communications, with the user interface implemented using a web browser;
- The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
- The database server moves information to and from the database and handles transaction management.

3.44 Language processing systems

- Accept a natural or artificial language as input and generate some other representation of that language.
- May include an interpreter to act on the instructions in the language that is being processed.
- Used in situations where the easiest way to solve a problem is to describe an algorithm or describe the system data
- Meta-case tools process tool descriptions, method rules, etc and generate tools.

3.45 The architecture of a language processing system

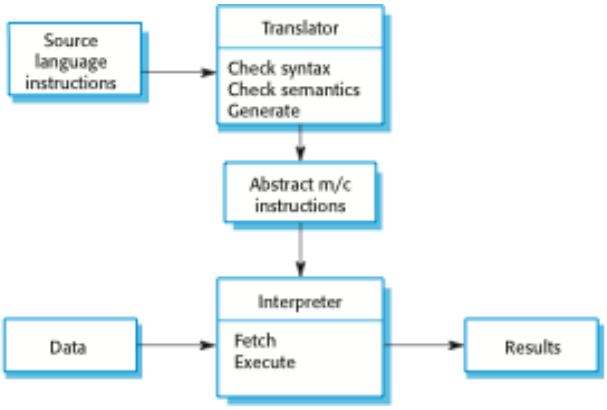

Figure 3.13:

3.46 Compiler components

- A lexical analyzer, which takes input language tokens and converts them to an internal form.
- A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- A syntax analyzer, which checks the syntax of the language being translated.

- A syntax tree, which is an internal structure representing the program being compiled.

  Compiler components

- A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- A code generator that 'walks' the syntax tree and generates abstract machine code.

3.47 A pipe and filter compiler architecture

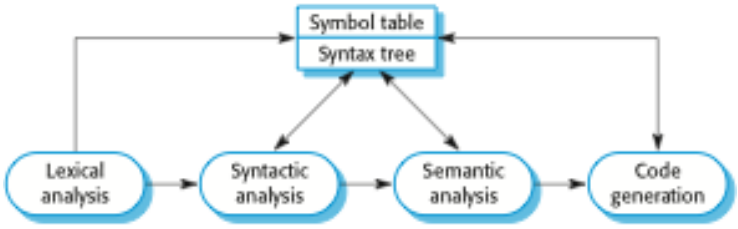

Figure 3.14:

3.48 A repository architecture for a language processing system

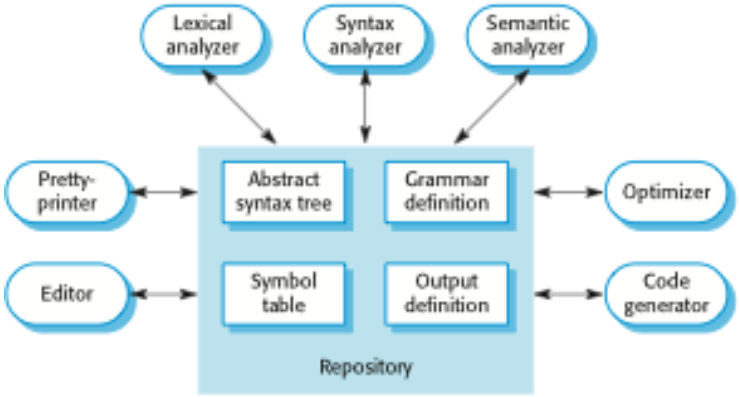

Figure 3.15:

3.49 Key points

- Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

Chapter 4

CH-11 Security and Dependability

4.1 Topics covered

- Dependability properties
  −The system attributes that lead to dependability.

- Availability and reliability
  −Systems should be available to deliver service and perform as expected.

- Safety
  −Systems should not behave in an unsafe way.

- Security
  −Systems should protect themselves and their data from external interference.

4.2 System dependability

- For many computer-based systems, the most important system property is the dependability of the system.

- The dependability of a system reflects the user's degree of trust in that system. It reflects the extent of the user's confidence that it will operate as users expect and that it will not 'fail' in normal use.

- Dependability covers the related systems attributes of reliability, availability and security. These are all inter-dependent.

4.3 Importance of dependability

- System failures may have widespread effects with large numbers of people affected by the failure.
- Systems that are not dependable and are unreliable, unsafe or insecure may be rejected by their users.
- The costs of system failure may be very high if the failure leads to economic losses or physical damage.
- Undependable systems may cause information loss with a high consequent recovery cost.

4.4 Causes of failure

1. **Hardware failure** – Hardware fails because of design and manufacturing errors or because components have reached the end of their natural life.
2. **Software failure** – Software fails due to errors in its specification, design or implementation.
3. **Operational failure** – Human operators make mistakes. Now perhaps the largest single cause of system failures in socio-technical systems.

4.5 Principal dependability properties

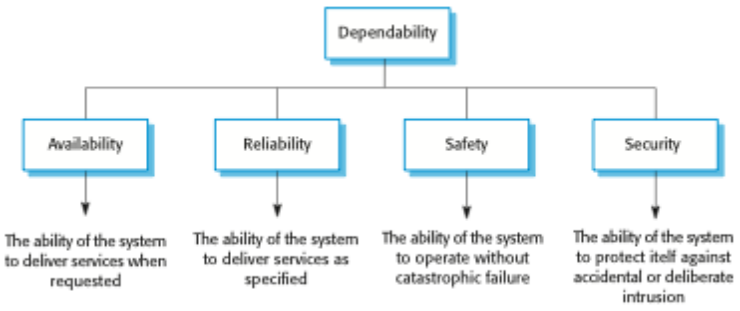

Figure 4.1:

4.6 Principal properties

- **Availability** – The probability that the system will be up and running and able to deliver useful services to users.
- **Reliability** – The probability that the system will correctly deliver services as expected by users.
- **Safety** – A judgment of how likely it is that the system will cause damage to people or its environment.
- **Security** – A judgment of how likely it is that the system can resist accidental or deliberate intrusions.

4.7 Other dependability properties

- **Repairability** – Reflects the extent to which the system can be repaired in the event of a failure
- **Maintainability** – Reflects the extent to which the system can be adapted to new requirements;
- **Survivability** – Reflects the extent to which the system can deliver services whilst under hostile attack;
- **Error tolerance** – Reflects the extent to which user input errors can be avoided and tolerated.

4.8 Repairability

- The disruption caused by system failure can be minimized if the system can be repaired quickly.
- This requires problem diagnosis, access to the failed component(s) and making changes to fix the problems.
- Repairability is a judgment of how easy it is to repair the software to correct the faults that led to a system failure.
- Repairability is affected by the operating environment so is hard to assess before system deployment.

4.9 Maintainability

- A system attribute that is concerned with the ease of repairing the system after a failure has been discovered or changing the system to include new features.

- Repairability − short-term perspective to get the system back into service; Maintainability − long-term perspective.
- Very important for critical systems as faults are often introduced into a system because of maintenance problems. If a system is maintainable, there is a lower probability that these faults will be introduced or undetected.

4.10 Survivability

- The ability of a system to continue to deliver its services to users in the face of deliberate or accidental attack
- This is an increasingly important attribute for distributed systems whose security can be compromised
- Survivability subsumes the notion of resilience - the ability of a system to continue in operation in spite of component failures

4.11 Error tolerance

- Part of a more general usability property and reflects the extent to which user errors are avoided, detected or tolerated.
- User errors should, as far as possible, be detected and corrected automatically and should not be passed on to the system and cause failures.

4.12 Dependability attribute examples

- Safe system operation depends on the system being available and operating reliably.
- A system may be unreliable because its data has been corrupted by an external attack.
- Denial of service attacks on a system are intended to make it unavailable.
- If a system is infected with a virus, you cannot be confident in its reliability or safety.

4.13 Dependability achievement

- Avoid the introduction of accidental errors when developing the system.
- Design V & V processes that are effective in discovering residual errors in the system.

- Design protection mechanisms that guard against external attacks.
- Configure the system correctly for its operating environment.
- Include recovery mechanisms to help restore normal system service after a failure.

4.14 Dependability costs

- Dependability costs tend to increase exponentially as increasing levels of dependability are required.
- There are two reasons for this
- The use of more expensive development techniques and hardware that are required to achieve the higher levels of dependability.
- The increased testing and system validation that is required to convince the system client and regulators that the required levels of dependability have been achieved.

4.15 Cost/dependability curve

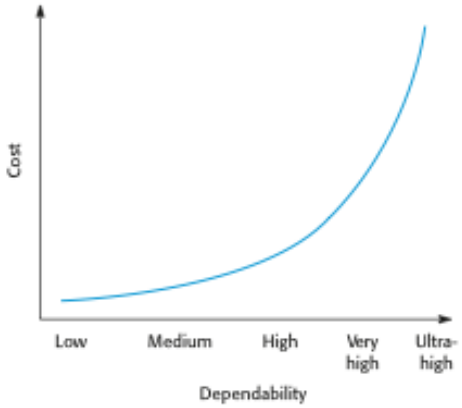

Figure 4.2:

4.16 Dependability economics

- Because of very high costs of dependability achievement, it may be more cost effective to accept untrustworthy systems and pay for failure costs

- However, this depends on social and political factors. A reputation for products that can't be trusted may lose future business
- Depends on system type - for business systems in particular, modest levels of dependability may be adequate

4.17 Availability and reliability

- Reliability
- The probability of failure-free system operation over a specified time in a given environment for a given purpose
- Availability
- The probability that a system, at a point in time, will be operational and able to deliver the requested services
- Both of these attributes can be expressed quantitatively e.g. availability of 0.999 means that the system is up and running for 99.9

4.18 Availability and reliability

- It is sometimes possible to subsume system availability under system reliability
- Obviously if a system is unavailable it is not delivering the specified system services.
- However, it is possible to have systems with low reliability that must be available.
- So long as system failures can be repaired quickly and does not damage data, some system failures may not be a problem.
- Availability is therefore best considered as a separate attribute reflecting whether or not the system can deliver its services.
- Availability takes repair time into account, if the system has to be taken out of service to repair faults.

4.19 Perceptions of reliability

- The formal definition of reliability does not always reflect the user's perception of a system's reliability

- The assumptions that are made about the environment where a system will be used may be incorrect
  −Usage of a system in an office environment is likely to be quite different from usage of the same system in a university environment
- The consequences of system failures affects the perception of reliability
  −Unreliable windscreen wipers in a car may be irrelevant in a dry climate
  −Failures that have serious consequences (such as an engine breakdown in a car) are given greater weight by users than failures that are inconvenient

4.20 Reliability and specifications

- Reliability can only be defined formally with respect to a system specification i.e. a failure is a deviation from a specification.
- However, many specifications are incomplete or incorrect – hence, a system that conforms to its specification may 'fail' from the perspective of system users.
- Furthermore, users don't read specifications so don't know how the system is supposed to behave.
- Therefore perceived reliability is more important in practice.

4.21 Availability perception

- Availability is usually expressed as a percentage of the time that the system is available to deliver services e.g. 99.95
- However, this does not take into account two factors:
- The number of users affected by the service outage. Loss of service in the middle of the night is less important for many systems than loss of service during peak usage periods.
- The length of the outage. The longer the outage, the more the disruption. Several short outages are less likely to be disruptive than 1 long outage. Long repair times are a particular problem.

4.22 Key points

- The dependability in a system reflects the user's trust in that system.
- Dependability is a term used to describe a set of related 'non-functional' system attributes – availability, reliability, safety and security.
- The availability of a system is the probability that it will be available to deliver services when requested.

- The reliability of a system is the probability that system services will be delivered as specified.

4.23 Reliability terminology

Term	Description
Human error or mistake	Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
System fault	A characteristic of a software system that can lead to a system error. The fault is the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23.00.
System error	An erroneous system state that can lead to system behavior that is unexpected by system users. The value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed.
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid.

4.24 Faults and failures

- Failures are a usually a result of system errors that are derived from faults in the system
- However, faults do not necessarily result in system errors
- The erroneous system state resulting from the fault may be transient and 'corrected' before an error arises.
- The faulty code may never be executed.

- Errors do not necessarily lead to system failures
- The error can be corrected by built-in error detection and recovery
- The failure can be protected against by built-in protection facilities. These may, for example, protect system resources from system errors

4.25 A system as an input/output mapping

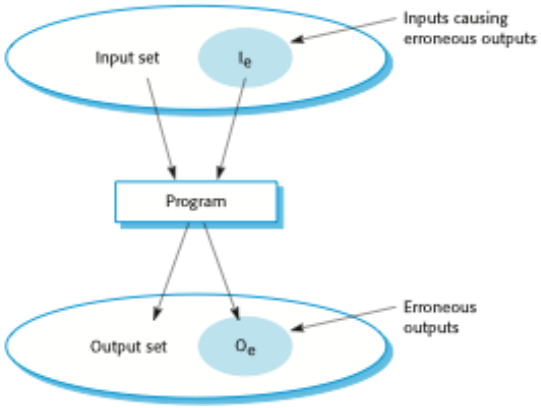

Figure 4.3:

4.26 Software usage patterns

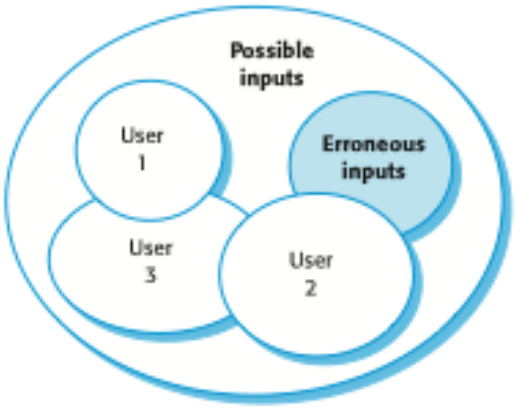

Figure 4.4:

4.27 Reliability in use

- Removing X% of the faults in a system will not necessarily improve the reliability by X%. A study at IBM showed that removing 60% of product defects resulted in a 3% improvement in reliability.
- Program defects may be in rarely executed sections of the code so may never be encountered by users. Removing these does not affect the perceived reliability.
- Users adapt their behaviour to avoid system features that may fail for them.
- A program with known faults may therefore still be perceived as reliable by its users.

4.28 Reliability achievement

- Fault avoidance
- Development technique are used that either minimise the possibility of mistakes or trap mistakes before they result in the introduction of system faults.
- Fault detection and removal
- Verification and validation techniques that increase the probability of detecting and correcting errors before the system goes into service are used.
- Fault tolerance
- Run-time techniques are used to ensure that system faults do not result in system errors and/or that system errors do not lead to system failures.

4.29 Safety

- Safety is a property of a system that reflects the system's ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system's environment.
- It is important to consider software safety as most devices whose failure is critical now incorporate software -based control systems.
- Safety requirements are often exclusive requirements i.e. they exclude undesirable situations rather than specify required system services. These generate functional safety requirements.

4.30 Safety criticality

- Primary safety-critical systems
- Embedded software systems whose failure can cause the associated hardware to fail and directly threaten people. Example is the insulin pump control system.
- Secondary safety-critical systems
- Systems whose failure results in faults in other (socio-technical)systems, which can then have safety consequences. For example, the MHC-PMS is safety-critical as failure may lead to inappropriate treatment being prescribed.

4.31 Safety and reliability

- Safety and reliability are related but distinct
- In general, reliability and availability are necessary but not sufficient conditions for system safety
- Reliability is concerned with conformance to a given specification and delivery of service
- Safety is concerned with ensuring system cannot cause damage irrespective of whether or not it conforms to its specification

4.32 Unsafe reliable systems

- There may be dormant faults in a system that are undetected for many years and only rarely arise.
- Specification errors
- If the system specification is incorrect then the system can behave as specified but still cause an accident.
- Hardware failures generating spurious inputs
- Hard to anticipate in the specification.
- Context-sensitive commands i.e. issuing the right command at the wrong time
- Often the result of operator error.

4.33 Safety terminology

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events which results in human death or injury, damage to property, or to the environment. An overdose of insulin is an example of an accident.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that measures blood glucose is an example of a hazard.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. Damage resulting from an overdose of insulin could be serious injury or the death of the user of the insulin pump.
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is 'very high'.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from 'probable' (say 1/100 chance of a hazard occurring) to 'implausible' (no conceivable situations are likely in which the hazard could occur). The probability of a sensor failure in the insulin pump that results in an overdose is probably low.
Risk	This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident. The risk of an insulin overdose is probably medium to low.

4.34 Safety achievement

- Hazard avoidance
- The system is designed so that some classes of hazard simply cannot arise.
- Hazard detection and removal
- The system is designed so that hazards are detected and removed before they result in an accident.
- Damage limitation
- The system includes protection features that minimise the damage that may result from an accident.

4.35 Normal accidents

- Accidents in complex systems rarely have a single cause as these systems are designed to be resilient to a single point of failure
- Designing systems so that a single point of failure does not cause an accident is a fundamental principle of safe systems design.
- Almost all accidents are a result of combinations of malfunctions rather than single failures.
- It is probably the case that anticipating all problem combinations, especially, in software controlled systems is impossible so achieving complete safety is impossible. Accidents are inevitable.

4.36 Software safety benefits

- Although software failures can be safety-critical, the use of software control systems contributes to increased system safety
- Software monitoring and control allows a wider range of conditions to be monitored and controlled than is possible using electro-mechanical safety systems.
- Software control allows safety strategies to be adopted that reduce the amount of time people spend in hazardous environments.
- Software can detect and correct safety-critical operator errors.

4.37 Security

- The security of a system is a system property that reflects the system's ability to protect itself from accidental or deliberate external attack.
- Security is essential as most systems are networked so that external access to the system through the Internet is possible.
- Security is an essential pre-requisite for availability, reliability and safety.

4.38 Fundamental security

- If a system is a networked system and is insecure then statements about its reliability and its safety are unreliable.
- These statements depend on the executing system and the developed system being the same. However, intrusion can change the executing system and/or its data.
- Therefore, the reliability and safety assurance is no longer valid.

4.39 Security terminology

Term	Definition
Asset	Something of value which has to be protected. The asset may be the software system itself or data used by that system.
Exposure	Possible loss or harm to a computing system. This can be loss or damage to data, or can be a loss of time and effort if recovery is necessary after a security breach.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.
Attack	An exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
Threats	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack.
Control	A protective measure that reduces a system's vulnerability. Encryption is an example of a control that reduces a vulnerability of a weak access control system.

4.40 Threat classes

- Threats to the confidentiality of the system and its data
- Can disclose information to people or programs that do not have authorization to access that information.
- Threats to the integrity of the system and its data
- Can damage or corrupt the software or its data.
- Threats to the availability of the system and its data
- Can restrict access to the system and data for authorized users.

4.41 Examples of security terminology (MHC-PMS)

Term	Example
Asset	The records of each patient that is receiving or has received treatment.
Exposure	Potential financial loss from future patients who do not seek treatment because they do not trust the clinic to maintain their data. Financial loss from legal action by the sports star. Loss of reputation.
Vulnerability	A weak password system which makes it easy for users to set guessable passwords. User ids that are the same as names.
Attack	An impersonation of an authorized user.
Threat	An unauthorized user will gain access to the system by guessing the credentials (login name and password) of an authorized user.
Control	A password checking system that disallows user passwords that are proper names or words that are normally included in a dictionary.

4.42 Damage from insecurity

- Denial of service
- The system is forced into a state where normal services are unavailable or where service provision is significantly degraded
- Corruption of programs or data
- The programs or data in the system may be modified in an unauthorised way
- Disclosure of confidential information
- Information that is managed by the system may be exposed to people who are not authorised to read or use that information

4.43 Security assurance

- Vulnerability avoidance
- The system is designed so that vulnerabilities do not occur. For example, if there is no external network connection then external attack is impossible
- Attack detection and elimination
- The system is designed so that attacks on vulnerabilities are detected and neutralised before they result in an exposure. For example, virus checkers find and remove viruses before they infect a system
- Exposure limitation and recovery
- The system is designed so that the adverse consequences of a successful attack are minimised. For example, a backup policy allows damaged information to be restored

4.44 Key points

- Reliability is related to the probability of an error occurring in operational use. A system with known faults may be reliable.
- Safety is a system attribute that reflects the system's ability to operate without threatening people or the environment.
- Security is a system attribute that reflects the system's ability to protect itself from external attack.
- Dependability is compromised if a system is insecure as the code or data may be corrupted.

Chapter 5

CH-22 Project Management

5.1 Topics covered

- Risk management
- Managing people
- Teamwork

5.2 Software project management

- Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software.
- Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software.

5.2.1 Success criteria

- Deliver the software to the customer at the agreed time.
- Keep overall costs within budget.
- Deliver software that meets the customer's expectations.
- Maintain a happy and well-functioning development team.

5.2.2 Software management distinctions

- **The product is intangible**.
  −Software cannot be seen or touched. Software project managers cannot see progress by simply looking at the artefact that is being constructed.

- **Many software projects are 'one-off' projects**.
  − Large software projects are usually different in some ways from previous projects. Even managers who have lots of previous experience may find it difficult to anticipate problems.

- **Software processes are variable and organization specific**.
  − We still cannot reliably predict when a particular software process is likely to lead to development problems.

5.2.3 Management activities

1. **Project planning**
   −Project managers are responsible for planning. estimating and scheduling project development and assigning people to tasks.

2. **Reporting**
   −Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software.

3. **Risk management**
   −Project managers assess the risks that may affect a project, monitor these risks and take action when problems arise.

4. **People management**
   − Project managers have to choose people for their team and establish ways of working that leads to effective team performance

5. **Proposal writing**
   − The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out.

5.3 Risk management

Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project. A risk is a probability that some adverse circumstance will occur

- Project risks affect schedule or resources;

- Product risks affect the quality or performance of the software being developed;

- Business risks affect the organisation developing or procuring the software.

5.3.1 Examples of common project, product, and business risks

Risk	Affects	Description
Staff Turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project & Product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project & Product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project & Product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

5.4 The risk management process

- Risk identification
- Identify project, product and business risks;
- Risk analysis
- Assess the likelihood and consequences of these risks;
- Risk planning

- Draw up plans to avoid or minimise the effects of the risk;
- Risk monitoring
- Monitor the risks throughout the project;

5.5 The risk management process

Figure 5.1:

5.6 Risk identification

- May be a team activities or based on the individual project manager's experience.
- A checklist of common risks may be used to identify risks in a project
- Technology risks.
- People risks.
- Organisational risks.
- Requirements risks.
- Estimation risks.

5.7 Examples of different risk types

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)
Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Estimation	The time required to develop the software is underestimated. (12) The rate of defect repair is underestimated. (13) The size of the software is underestimated. (14)

5.8 Risk analysis

- Assess probability and seriousness of each risk.
- Probability may be very low, low, moderate, high or very high.
- Risk consequences might be catastrophic, serious, tolerable or insignificant.

5.9 Risk types and examples

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (7).	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project (3).	High	atastrophic
Key staff are ill at critical times in the project (4).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused. (2).	Moderate	Serious
Changes to requirements that require major design rework are proposed (10).	Moderate	Serious
The organization is restructured so that different management are responsible for the project (6).	High	Serious
The database used in the system cannot process as many transactions per second as expected (1).	Moderate	Serious
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	Tolerable
The rate of defect repair is underestimated (13).	Moderate	Tolerable
The size of the software is underestimated (14).	High	Tolerable
Code generated by code generation tools is inefficient (8).	Moderate	Insignificant

5.10 Risk planning

- Consider each risk and develop a strategy to manage that risk.
- Avoidance strategies
- The probability that the risk will arise is reduced;
- Minimisation strategies
- The impact of the risk on the project or product will be reduced;
- Contingency plans
- If the risk arises, contingency plans are plans to deal with that risk;

5.11 Risk monitoring

- Assess each identified risks regularly to decide whether or not it is becoming less or more probable.
- Also assess whether the effects of the risk have changed.
- Each key risk should be discussed at management progress meetings.

5.12 Key points

- Good project management is essential if software engineering projects are to be developed on schedule and within budget.
- Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative with no body of experience to guide their management. Software processes are not as mature as traditional engineering processes.
- Risk management is now recognized as one of the most important project management tasks.
- Risk management involves identifying and assessing project risks to establish the probability that they will occur and the consequences for the project if that risk does arise. You should make plans to avoid, manage or deal with likely risks if or when they arise.

5.13 Strategies to help manage risk

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.

5.14 Risk indicators

Risk type	Potential indicators
Technology	Late delivery of hardware or support software; many reported technology problems.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Organizational	Organizational gossip; lack of action by senior management.
Tools	Reluctance by team members to use tools; complaints about CASE tools; demands for higher-powered workstations.
Requirements	Many requirements change requests; customer complaints.
Estimation	Failure to meet agreed schedule; failure to clear reported defects.

5.15 Managing people

- People are an organisation's most important assets.
- The tasks of a manager are essentially people-oriented. Unless there is some understanding of people, management will be unsuccessful.
- Poor people management is an important contributor to project failure.

5.16 People management factors

- Consistency
- Team members should all be treated in a comparable way without favourites or discrimination.
- Respect
- Different team members have different skills and these differences should be respected.
- Inclusion

- Involve all team members and make sure that people's views are considered.
- Honesty
- You should always be honest about what is going well and what is going badly in a project.

5.17 Motivating people

- An important role of a manager is to motivate the people working on a project.
- Motivation means organizing the work and the working environment to encourage people to work effectively.
- If people are not motivated, they will not be interested in the work they are doing. They will work slowly, be more likely to make mistakes and will not contribute to the broader goals of the team or the organization.
- Motivation is a complex issue but it appears that their are different types of motivation based on:
- Basic needs (e.g. food, sleep, etc.);
- Personal needs (e.g. respect, self-esteem);
- Social needs (e.g. to be accepted as part of a group).

5.18 Human needs hierarchy

Figure 5.2:

5.19 Need satisfaction

- In software development groups, basic physiological and safety needs are not an issue.
- Social
- Provide communal facilities;
- Allow informal communications e.g. via social networking
- Esteem
- Recognition of achievements;
- Appropriate rewards.
- Self-realization
- Training - people want to learn more;
- Assigning Responsibility.

5.20 Individual motivation

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of 6 developers than can develop new products based around the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team and creative new ideas are developed. The team decides to develop a peer-to-peer messaging system using digital televisions linked to the alarm network for communications. However, some months into the project, Alice notices that Dorothy, a hardware design expert, starts coming into work late, the quality of her work deteriorates and, increasingly, that she does not appear to be communicating with other members of the team. Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed, and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

After some initial denials that there is a problem, Dorothy admits that she has lost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity for this. Basically, she is working as a C programmer with other team members.

Although she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that

involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.

5.20.1 Solution to Dorithy's Problem

- Give her autonomy in designing
- Give her training on software engineering
- She will be motivated in the new work

5.21 Different Motivations for Different Personality types

- The needs hierarchy is almost certainly an over-simplification of motivation in practice.
- Motivation should also take into account different personality types: (Different People are Motivated in Different ways)
- Task-oriented;
- Self-oriented;
- Interaction-oriented.

5.22 Different Motivations for Different Personality types

- Task-oriented.
- The motivation for doing the work is the work itself;
- Acts as individuals
- Self-oriented.
- The work is a means to an end which is the achievement of individual goals - e.g. to get rich, to play tennis, to travel etc.;
- Acts as individuals
- Interaction-oriented
- The principal motivation is the presence and actions of co-workers. People go to work because they like to go to work.
- Mostly female workers

5.23 Motivation balance

- Individual motivations are made up of elements of each class.
- One personality type must be dominant
- The personality can change depending on personal circumstances and external events.
- However, people are not just motivated by personal factors but also by being part of a group and culture.
- People go to work because they are motivated by the people that they work with.

5.24 Teamwork

- Most software engineering is a group activity
- The development schedule for most non-trivial software projects is such that they cannot be completed by one person working alone.
- A good group is cohesive and has a team spirit. The people involved are motivated by the success of the group as well as by their own personal goals.
- Group interaction is a key determinant of group performance.
- Flexibility in group composition is limited
- Managers must do the best they can with available people.

5.25 Group cohesiveness

- In a cohesive group, members consider the group to be more important than any individual in it.
- The advantages of a cohesive group are:
- Group quality standards can be developed by the group members.
- Team members learn from each other and get to know each other's work; Inhibitions caused by ignorance are reduced.
- Knowledge is shared. Continuity can be maintained if a group member leaves.
- 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.

5.26 Team spirit

Alice, an experienced project manager, understands the importance of creating a cohesive group. As they are developing a new product, she takes the opportunity of involving all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She also encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an 'away day' for the group where the team spends two days on 'technology updating'. Each team member prepares an update on a relevant technology and presents it to the group. This is an off-site meeting in a good hotel and plenty of time is scheduled for discussion and social interaction.

5.27 The effectiveness of a team

- The people in the group
- You need a mix of people in a project group as software development involves diverse activities such as negotiating with clients, programming, testing and documentation.
- The group organization
- A group should be organized so that individuals can contribute to the best of their abilities and tasks can be completed as expected.
- Technical and managerial communications
- Good communications between group members, and between the software engineering team and other project stakeholders, is essential.

5.28 Selecting group members

- A manager or team leader's job is to create a cohesive group and organize their group so that they can work together effectively.
- This involves creating a group with the right balance of technical skills and personalities, and organizing that group so that the members work together effectively.

5.29 Assembling a team

- May not be possible to appoint the ideal people to work on a project
- Project budget may not allow for the use of highly-paid staff;
- Staff with the appropriate experience may not be available;
- An organisation may wish to develop employee skills on a software project.
- Managers have to work within these constraints especially when there are shortages of trained staff.

5.30 Group composition

- Group composed of members who share the same motivation can be problematic
- Task-oriented - everyone wants to do their own thing;
- Self-oriented - everyone wants to be the boss;
- Interaction-oriented - too much chatting, not enough work.
- An effective group has a balance of all types.
- This can be difficult to achieve software engineers are often task-oriented.
- Interaction-oriented people are very important as they can detect and defuse tensions that arise.

5.31 Group composition

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

Alice-self-oriented Brian—task-oriented Bob-task-oriented Carol-interaction-oriented Dorothy-self-oriented Ed-interaction-oriented Fred-task-oriented

5.32 Group organization

- The way that a group is organized affects the decisions that are made by that group, the ways that information is exchanged and the interactions between the development group and external project stakeholders.

- Key questions include:
  −Should the project manager be the technical leader of the group?
  −Who will be involved in making critical technical decisions, and how will these be made?
  −How will interactions with external stakeholders and senior company management be handled?
  −How can groups integrate people who are not co-located? • How can knowledge be shared across the group?

- Small software engineering groups are usually organised informally without a rigid structure.

- For large projects, there may be a hierarchical structure where different groups are responsible for different sub-projects.

- Agile development is always based around an informal group on the principle that formal structure inhibits information exchange

5.33 Informal groups

- The group acts as a whole and comes to a consensus on decisions affecting the system.

- The group leader serves as the external interface of the group but does not allocate specific work items.

- Rather, work is discussed by the group as a whole and tasks are allocated according to ability and experience.

- This approach is successful for groups where all members are experienced and competent.

5.34 Group communications

- Good communications are essential for effective group working.

- Information must be exchanged on the status of work, design decisions and changes to previous decisions.

- Good communications also strengthens group cohesion as it promotes understanding.

5.35 Group communications

- Group size
- The larger the group, the harder it is for people to communicate with other group members.
- Group structure
- Communication is better in informally structured groups than in hierarchically structured groups.
- Group composition
- Communication is better when there are different personality types in a group and when groups are mixed rather than single sex.
- The physical work environment
- Good workplace organisation can help encourage communications.

5.36 Key points

- People are motivated by interaction with other people, the recognition of management and their peers, and by being given opportunities for personal development.
- Software development groups should be fairly small and cohesive. The key factors that influence the effectiveness of a group are the people in that group, the way that it is organized and the communication between group members.
- Communications within a group are influenced by factors such as the status of group members, the size of the group, the gender composition of the group, personalities and available communication channels.

Chapter 6

CH-23 Project Planning

6.1 Topics covered

- Software pricing
- Plan-driven development
- Project scheduling
- Agile planning
- Estimation techniques

6.2 Project planning

- Project planning involves breaking down the work into parts and assign these to project team members, anticipate problems that might arise and prepare tentative solutions to those problems.
- The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.

6.3 Planning stages

- At the proposal stage, when you are bidding for a contract to develop or provide a software system.
- 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.

- Periodically throughout the project, when you modify your plan in the light of experience gained and information from monitoring the progress of the work.

6.4 Proposal planning

- Planning may be necessary with only outline software requirements.
- The aim of planning at this stage is to provide information that will be used in setting a price for the system to customers.

6.5 Software pricing

- Estimates are made to discover the cost, to the developer, of producing a software system.
- You take into account, hardware, software, travel, training and effort costs.
- There is not a simple relationship between the development cost and the price charged to the customer.
- Broader organisational, economic, political and business considerations influence the price charged.

6.6 Plan-driven development

- Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail.
- Plan-driven development is based on engineering project management techniques and is the 'traditional' way of managing large software development projects.
- A project plan is created that records the work to be done, who will do it, the development schedule and the work products.
- Managers use the plan to support project decision making and as a way of measuring progress.

6.7 Factors affecting software pricing

Factor	Description
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.

6.8 Plan-driven development – pros and cons

- The arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be closely taken into account, and that potential problems and dependencies are discovered before the project starts, rather than once the project is underway.

- The principal argument against plan-driven development is that many early decisions have to be revised because of changes to the environment in which the software is to be developed and used.

6.9 Project plans

- In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.
- Plan sections
- Introduction
- Project organization
- Risk analysis
- Hardware and software resource requirements
- Work breakdown
- Project schedule
- Monitoring and reporting mechanisms

6.10 Project plan supplements

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

6.11 The planning process

- Project planning is an iterative process that starts when you create an initial project plan during the project startup phase.

- Plan changes are inevitable.
- As more information about the system and the project team becomes available during the project, you should regularly revise the plan to reflect requirements, schedule and risk changes.
- Changing business goals also leads to changes in project plans. As business goals change, this could affect all projects, which may then have to be re-planned.

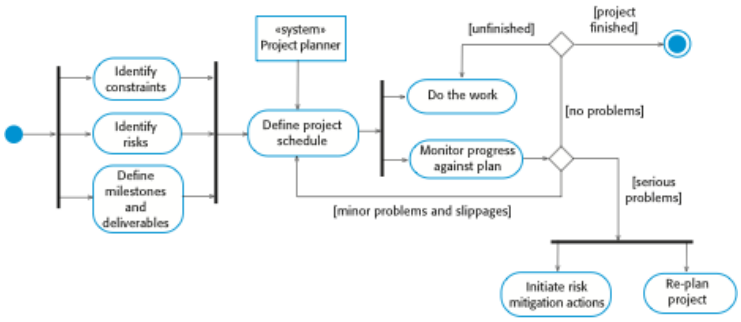

Figure 6.1:

6.12 Project scheduling

- 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.

6.13 Project scheduling activities

- Split project into tasks and estimate time and resources required to complete each task.
- Organize tasks concurrently to make optimal use of workforce.

- Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- Dependent on project managers intuition and experience.

6.14 Milestones and deliverables

- Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing.
- Deliverables are work products that are delivered to the customer, e.g. a requirements document for the system.

6.15 The project scheduling process

Figure 6.2:

6.16 Scheduling problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- Productivity is not proportional to the number of people working on a task.
- Adding people to a late project makes it later because of communication overheads.
- The unexpected always happens. Always allow contingency in planning.

6.17 Schedule representation

- Graphical notations are normally used to illustrate the project schedule.
- These show the project breakdown into tasks. Tasks should not be too small. They should take about a week or two.
- Bar charts are the most commonly used representation for project schedules. They show the schedule as activities or resources against time.

6.18 Tasks, durations, and dependencies

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

6.19 Activity bar chart

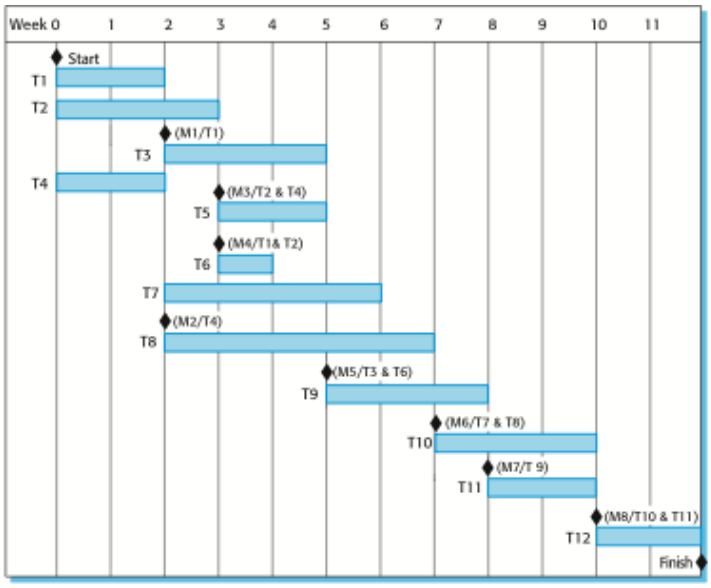

Figure 6.3:

6.20 Staff allocation chart

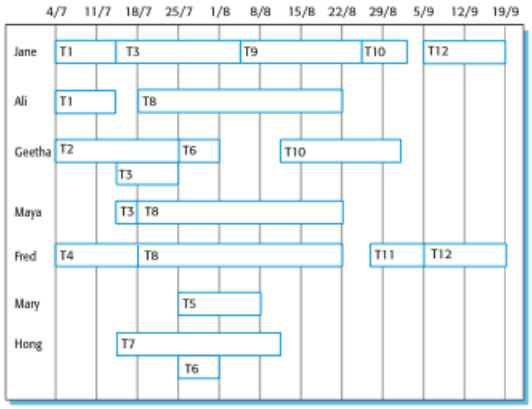

Figure 6.4:

6.21 Agile planning

- Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments.
- Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development.
- The decision on what to include in an increment depends on progress and on the customer's priorities.
- The customer's priorities and requirements change so it makes sense to have a flexible plan that can accommodate these changes.

6.22 Agile planning stages

- Release planning, which looks ahead for several months and decides on the features that should be included in a release of a system.
- Iteration planning, which has a shorter term outlook, and focuses on planning the next increment of a system. This is typically 2-4 weeks of work for the team.

6.23 Planning in XP

Figure 6.5:

6.24 Story-based planning

- The system specification in XP is based on user stories that reflect the features that should be included in the system.

- The project team read and discuss the stories and rank them in order of the amount of time they think it will take to implement the story.

- 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.

- Stories to be implemented in each iteration are chosen, with the number of stories reflecting the time to deliver an iteration (usually 2 or 3 weeks).

6.25 Key points

- The price charged for a system does not just depend on its estimated development costs; it may be adjusted depending on the market and organizational priorities.

- Plan-driven development is organized around a complete project plan that defines the project activities, the planned effort, the activity schedule and who is responsible for each activity.

- Project scheduling involves the creation of graphical representations the project plan. Bar chartsshow the activity duration and staffing timelines, are the most commonly used schedule representations.

- The XP planning game involves the whole team in project planning. The plan is developed incrementally and, if problems arise, is adjusted. Software functionality is reduced instead of delaying delivery of an increment.

6.26 Estimation techniques

- Organizations need to make software effort and cost estimates. There are two types of technique that can be used to do this:

- Experience-based techniques The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.

- Algorithmic cost modeling In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.

6.27 Experience-based approaches

- Experience-based techniques rely on judgments based on experience of past projects and the effort expended in these projects on software development activities.
- Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be developed.
- You document these in a spreadsheet, estimate them individually and compute the total effort required.
- It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate.

6.28 Algorithmic cost modelling

- Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:
- $Effort = A * Size^B * M$
- A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes.
- The most commonly used product attribute for cost estimation is code size.
- Most models are similar but they use different values for A, B and M.

6.29 Estimation accuracy

- The size of a software system can only be known accurately when it is finished.
- Several factors influence the final size
- Use of COTS and components;
- Programming language;
- Distribution of system.
- As the development process progresses then the size estimate becomes more accurate.
- The estimates of the factors contributing to B and M are subjective and vary according to the judgment of the estimator.

6.30 Estimate uncertainty

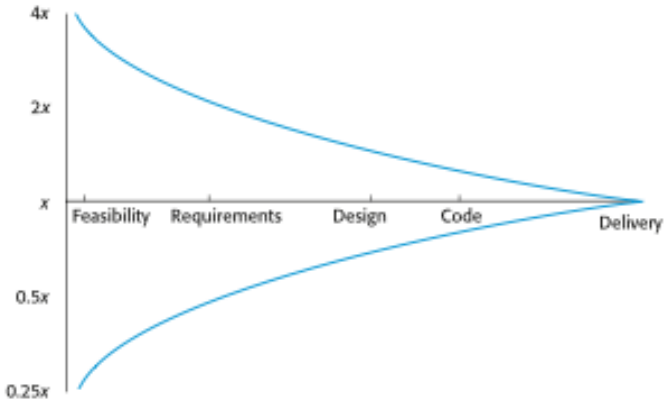

Figure 6.6:

6.31 The COCOMO 2 model

- An empirical model based on project experience.
- Well-documented, 'independent' model which is not tied to a specific software vendor.
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- COCOMO 2 takes into account different approaches to software development, reuse, etc.

6.32 COCOMO 2 models

- COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.
- The sub-models in COCOMO 2 are:
- Application composition model. Used when software is composed from existing parts.
- Early design model. Used when requirements are available but design has not yet started.
- Reuse model. Used to compute the effort of integrating reusable components.

- Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

6.33 COCOMO estimation models

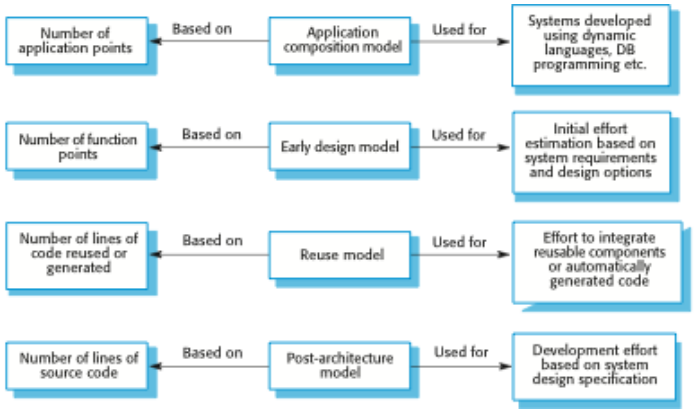

Figure 6.7:

6.34 Application composition model

- Supports prototyping projects and projects where there is extensive reuse.
- Based on standard estimates of developer productivity in application (object) points/month.
- Takes CASE tool use into account.
- Formula is
- $PM = \frac{(NAP*(1-\%\frac{reuse}{100}))}{PROD}$
- PM is the effort in person-months, NAP is the number of application points and PROD is the productivity.

6.35 Application-point productivity

Developer's experience and capability	Very low	Low	Nominal	High	Very High
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NAP/-month)	4	7	13	25	50

6.36 Early design model

- Estimates can be made after the requirements have been agreed.
- Based on a standard formula for algorithmic models
- $PM = A * SizeB * M$ where
- $M = PERS * RCPX * RUSE * PDIF * PREX * FCIL * SCED$;
- $A = 2.94$ in initial calibration, Size in KLOC, B varies from 1.1 to 1.24 depending on novelty of the project, development flexibility, risk management approaches and the process maturity.

6.37 Multipliers

- Multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc.
- RCPX - product reliability and complexity;
- RUSE - the reuse required;
- PDIF - platform difficulty;
- PREX - personnel experience;
- PERS - personnel capability;
- SCED - required schedule;
- FCIL - the team support facilities.

6.38 The reuse model

- Takes into account black-box code that is reused without change and code that has to be adapted to integrate it with new code.
- There are two versions:
- Black-box reuse where code is not modified. An effort estimate (PM) is computed.
- White-box reuse where code is modified. A size estimate equivalent to the number of lines of new source code is computed. This then adjusts the size estimate for new code.

6.39 Reuse model estimates 1

- For generated code:
- $PM = \frac{(ASLOC * \frac{AT}{100})}{ATPROD}$
- ASLOC is the number of lines of generated code
- AT is the percentage of code automatically generated.
- ATPROD is the productivity of engineers in integrating this code.

6.40 Reuse model estimates 2

- When code has to be understood and integrated:
- $ESLOC = ASLOC * (1 - \frac{AT}{100}) * AAM$
- ASLOC and AT as before.
- AAM is the adaptation adjustment multiplier computed from the costs of changing the reused code, the costs of understanding how to integrate the code and the costs of reuse decision making.

6.41 Post-architecture level

- Uses the same formula as the early design model but with 17 rather than 7 associated multipliers.
- The code size is estimated as:
- Number of lines of new code to be developed;
- Estimate of equivalent number of lines of new code computed using the reuse model;

- An estimate of the number of lines of code that have to be modified according to requirements changes.

6.42 The exponent term

- This depends on 5 scale factors (see next slide). Their $\frac{sum}{100}$ is added to 1.01
- A company takes on a project in a new domain. The client has not defined the process to be used and has not allowed time for risk analysis.

6.43 The company has a CMM level 2 rating.

- Precedenteness - new project (4)
- Development flexibility - no client involvement - Very high (1)
- Architecture/risk resolution - No risk analysis - V. Low .(5)
- Team cohesion - new team - nominal (3)
- Process maturity - some control - nominal (3)
- Scale factor is therefore 1.17.

6.44 Multipliers

- Product attributes
- Concerned with required characteristics of the software product being developed.
- Computer attributes
- Constraints imposed on the software by the hardware platform.
- Personnel attributes
- Multipliers that take the experience and capabilities of the people working on the project into account.
- Project attributes
- Concerned with the particular characteristics of the software development project.

6.45 Scale factors used in the exponent computation in the post-architecture model

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the client sets only general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis.
Team cohesion	Reflects how well the development team knows each other and work together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organization. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.

6.46 The effect of cost drivers on effort estimates

Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128,000 DSI
Initial COCOMO estimate without cost drivers	730 person-months
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	2,306 person-months
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	295 person-months

6.47 Project duration and staffing

- As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required.

- Calendar time can be estimated using a COCOMO 2 formula
- TDEV = 3 * (PM)(0.33+0.2*(B-1.01))
- PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model). This computation predicts the nominal schedule for the project.
- The time required is independent of the number of people working on the project.

6.48 Staffing requirements

- Staff required can't be computed by diving the development time by the required schedule.
- The number of people working on a project varies depending on the phase of the project.
- The more people who work on the project, the more total effort is usually required.
- A very rapid build-up of people often correlates with schedule slippage.

6.49 Key points

- Estimation techniques for software may be experience-based, where managers judge the effort required, or algorithmic, where the effort required is computed from other estimated project parameters.
- The COCOMO II costing model is an algorithmic cost model that uses project, product, hardware and personnel attributes as well as product size and complexity attributes to derive a cost estimate.

Chapter 7

CH-24 Quality Management

7.1 Topics covered

- Software quality
- Software standards
- Reviews and inspections
- Software measurement and metrics

7.2 Software quality management

- Concerned with ensuring that the required level of quality is achieved in a software product.
- Three principal concerns:
- At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
- At the project level, quality management involves the application of specific quality processes and checking that these planned processes have been followed.
- At the project level, quality management is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

7.3 Quality management activities

- Quality management provides an independent check on the software development process.
- The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals
- The quality team should be independent from the development team so that they can take an objective view of the software. This allows them to report on software quality without being influenced by software development issues.

7.4 Quality management and software development

Figure 7.1:

7.5 Quality planning

- A quality plan sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.
- The quality plan should define the quality assessment process.
- It should set out which organisational standards should be applied and, where necessary, define new standards to be used.

7.6 Quality plans

- Quality plan structure
- Product introduction;
- Product plans;
- Process descriptions;
- Quality goals;
- Risks and risk management.
- Quality plans should be short, succinct documents
- If they are too long, no-one will read them.

7.7 Scope of quality management

- Quality management is particularly important for large, complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes.
- For smaller systems, quality management needs less documentation and should focus on establishing a quality culture.

7.8 Software quality

- Quality, simplistically, means that a product should meet its specification.
- This is problematical for software systems
- There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
- Some quality requirements are difficult to specify in an unambiguous way;
- Software specifications are usually incomplete and often inconsistent.
- The focus may be 'fitness for purpose' rather than specification conformance.

7.9 Software fitness for purpose

- Have programming and documentation standards been followed in the development process?
- Has the software been properly tested?
- Is the software sufficiently dependable to be put into use?
- Is the performance of the software acceptable for normal use?
- Is the software usable?
- Is the software well-structured and understandable?

7.10 Software quality attributes

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

7.11 Quality conflicts

- It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.
- The quality plan should therefore define the most important quality attributes for the software that is being developed.
- The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

7.12 Process and product quality

- The quality of a developed product is influenced by the quality of the production process.

- This is important in software development as some product quality attributes are hard to assess.
- However, there is a very complex and poorly understood relationship between software processes and product quality.
- The application of individual skills and experience is particularly important in software development;
- External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

7.13 Process-based quality

Figure 7.2:

7.14 Software standards

- Standards define the required attributes of a product or process. They play an important role in quality management.
- Standards may be international, national, organizational or project standards.
- Product standards define characteristics that all software components should exhibit e.g. a common programming style.
- Process standards define how the software process should be enacted.

7.15 Importance of standards

- Encapsulation of best practice- avoids repetition of past mistakes.
- They are a framework for defining what quality means in a particular setting i.e. that organization's view of quality.
- They provide continuity - new staff can understand the organisation by understanding the standards that are used.

7.16 Product and process standards

Product standards	Process standards
Design review form Requirements document structure	Design review conduct Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

7.17 Problems with standards

- They may not be seen as relevant and up-to-date by software engineers.
- They often involve too much bureaucratic form filling.
- If they are unsupported by software tools, tedious form filling work is often involved to maintain the documentation associated with the standards.

7.18 Standards development

- Involve practitioners in development. Engineers should understand the rationale underlying a standard.
- Review standards and their usage regularly. Standards can quickly become outdated and this reduces their credibility amongst practitioners.

- Detailed standards should have specialized tool support. Excessive clerical work is the most significant complaint against standards.
- Web-based forms are not good enough.

7.19 ISO 9001 standards framework

- An international set of standards that can be used as a basis for developing quality management systems.
- ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.
- The ISO 9001 standard is a framework for developing software standards.
- It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

7.20 ISO 9001 core processes

Figure 7.3:

7.21 ISO 9001 and quality management

Figure 7.4:

7.22 ISO 9001 certification

- Quality standards and procedures should be documented in an organisational quality manual.
- An external body may certify that an organisation's quality manual conforms to ISO 9000 standards.
- Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

7.23 Key points

- Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability and so on.
- SQM includes defining standards for processes and products and establishing processes to check that these standards have been followed.
- Software standards are important for quality assurance as they represent an identification of 'best practice'.
- Quality management procedures may be documented in an organizational quality manual, based on the generic model for a quality manual suggested in the ISO 9001 standard.

7.24 Reviews and inspections

- A group examines part or all of a process or system and its documentation to find potential problems.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- There are different types of review with different objectives
- Inspections for defect removal (product);
- Reviews for progress assessment (product and process);
- Quality reviews (product and standards).

7.25 Quality reviews

- A group of people carefully examine part or all of a software system and its associated documentation.
- Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

7.26 The software review process

Figure 7.5:

7.27 Reviews and agile methods

- 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, pair programming ensures that code is constantly being examined and reviewed by another team member.
- 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.
- These are peer reviews where engineers examine the source of a system with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design,configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

7.28 Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

7.29 An inspection checklist

Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?

Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

Table 7.1:

7.30 Agile methods and inspections

- Agile processes rarely use formal inspection or peer review processes.
- Rather, they rely on team members cooperating to check each other's code, and informal guidelines, such as 'check before check-in', which suggest that programmers should check their own code.
- Extreme programming practitioners argue that pair programming is an effective substitute for inspection as this is, in effect, a continual inspection process.
- Two people look at every line of code and check it before it is accepted.

7.31 Software measurement and metrics

- Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- This allows for objective comparisons between techniques and processes.
- Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.
- There are few established standards in this area.

7.32 Software metric

- Any type of measurement which relates to a software system, process or related documentation
- Lines of code in a program, the Fog index, number of person-days required to develop a component.
- Allow the software and the software process to be quantified.
- May be used to predict product attributes or to control the software process.
- Product metrics can be used for general predictions or to identify anomalous components.

7.33 Predictor and control measurements

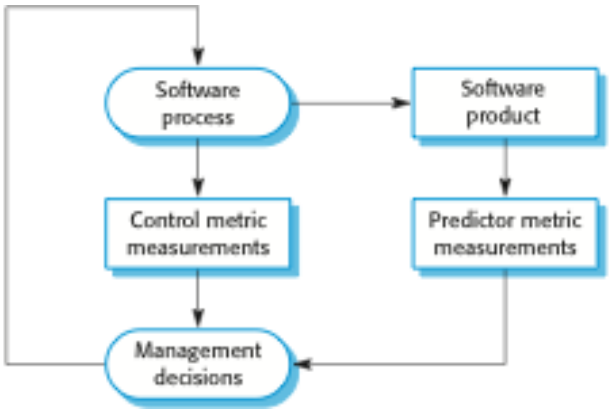

Figure 7.6:

7.34 Use of measurements

- To assign a value to system quality attributes
- By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
- To identify the system components whose quality is sub-standard
- Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

7.35 Metrics assumptions

- A software property can be measured.
- The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- This relationship has been formalised and validated.
- It may be difficult to relate what can be measured to desirable external quality attributes.

7.36 Relationships between internal and external software

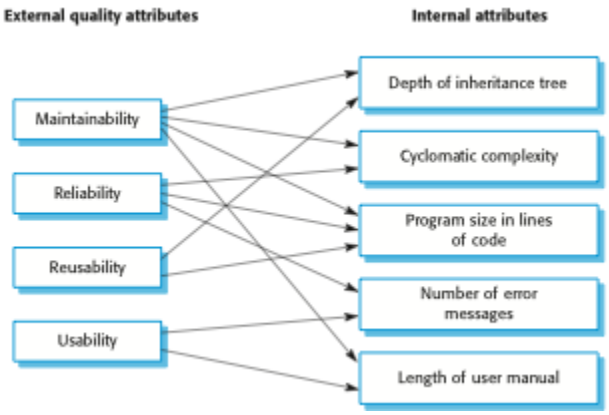

Figure 7.7:

7.37 Problems with measurement in industry

- It is impossible to quantify the return on investment of introducing an organizational metrics program.
- There are no standards for software metrics or standardized processes for measurement and analysis.
- In many companies, software processes are not standardized and are poorly defined and controlled.
- Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- Introducing measurement adds additional overhead to processes.

7.38 Product metrics

- A quality metric should be a predictor of product quality.
- Classes of product metric
- Dynamic metrics which are collected by measurements made of a program in execution;

- Static metrics which are collected by measurements made of the system representations;
- Dynamic metrics help assess efficiency and reliability
- Static metrics help assess complexity, understandability and maintainability.

7.39 Dynamic and static metrics

- Dynamic metrics are closely related to software quality attributes
- It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).
- Static metrics have an indirect relationship with quality attributes
- You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

7.40 Static software product metrics

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

7.41 Software component analysis

- System component can be analyzed separately using a range of metrics.
- The values of these metrics may then compared for different components and, perhaps, with historical measurement data collected on previous projects.
- Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

7.42 The process of product measurement

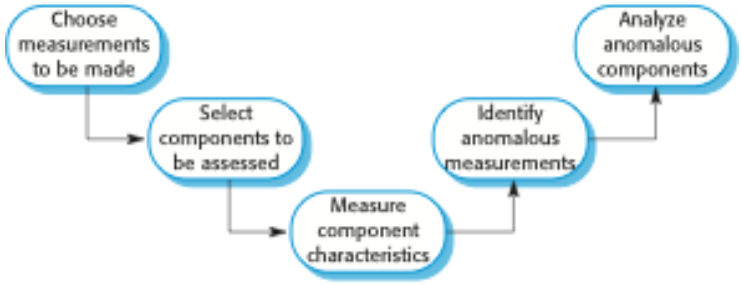

Figure 7.8:

7.43 Measurement surprises

- Reducing the number of faults in a program leads to an increased number of help desk calls
- The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;
- A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

7.44 Key points

- Reviews of the software process deliverables involve a team of people who check that quality standards are being followed.

- In a program inspection or peer review, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions
- Software measurement can be used to gather data about software and software processes.
- Product quality metrics are particularly useful for highlighting anomalous components that may have quality problems.

7.45 The CK object-oriented metrics suite

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Chapter 8

CH-25 Configuration Management

8.1 Topics covered

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

8.2 Configuration management

- Because software changes frequently, systems, can be thought of as a set of versions, each of which has to be maintained and managed.
- Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems.
- Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems. You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.

8.3 CM activities

- Change management
- Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.

- Version management
- 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.
- System building
- The process of assembling program components, data and libraries, then compiling these to create an executable system.
- Release management
- Preparing software for external release and keeping track of the system versions that have been released for customer use.

8.4 Configuration management activities

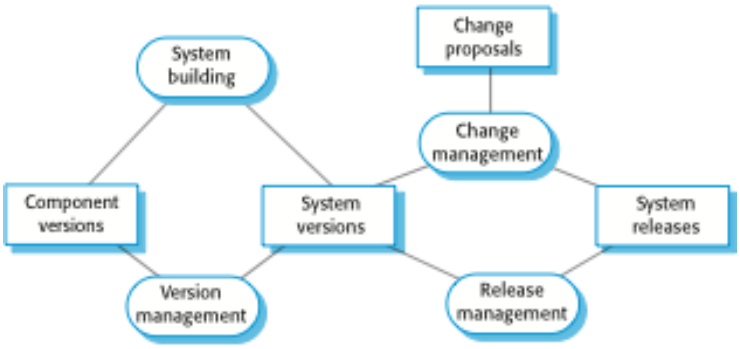

Figure 8.1:

8.5 CM terminology

Term	Explanation
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Configuration control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier, which is often composed of the configuration item name plus a version number.
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it should always be possible to recreate a baseline from its constituent components.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Mainline	A sequence of baselines representing different versions of a system.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.

System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.

Table 8.1:

8.6 Change management

- Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
- 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.

8.7 The change management process

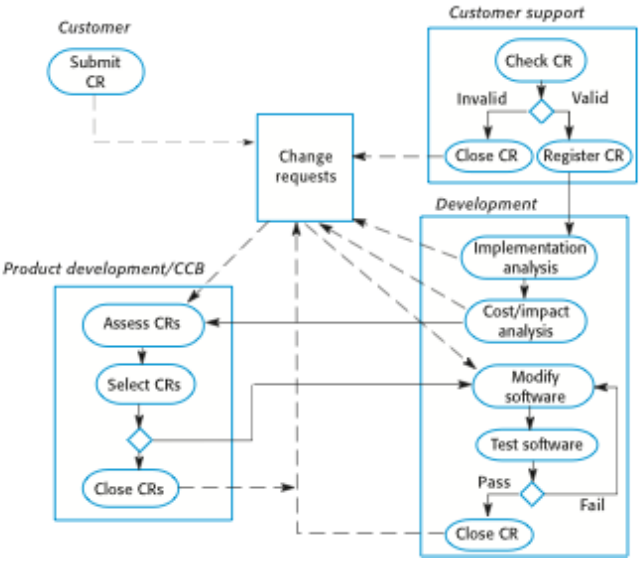

Figure 8.2:

8.8 A partially completed change request form (a)

Change Request Form

Project: SICSA/AppProcessing

Number: 23/02

Change requester: I. Sommerville

Date: 20/01/09

Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek

Analysis date: 25/01/09

Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

8.9 A partially completed change request form (b)

Change Request Form

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium

Change implementation:

Estimated effort: 2 hours

Date to SGA app. team: 28/01/09

CCB decision date: 30/01/09

Decision: Accept change. Change to be implemented in Release 1.2

Change implementor:

Date submitted to QA:

Date submitted to CM:

Comments:

Date of change: QA decision:

8.10 Factors in change analysis

- The consequences of not making the change
- The benefits of the change
- The number of users affected by the change
- The costs of making the change
- The product release cycle

8.11 Change management and agile methods

- In some agile methods, customers are directly involved in change management.
- The propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- Changes to improve the software improvement are decided by the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

8.12 Derivation history

SICSA project (XEP 6087)
APP-SYSTEMAUTHRBACUSER_ROLE
Object: currentRole
Author: R. Looek
Creation date: 13/11/2009

Modification history
Version Modifier Date Change Reason

1.0 J. Jones	1.1 R. Looek
11/11/2009	13/11/2009
Add header	New field
Submitted to CM	Change req. R07/02

8.13 Version management

- Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- Therefore version management can be thought of as the process of managing codelines and baselines. Codelines and baselines
- A codelineis a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- Codelinesnormally apply to components of systems so that there are different versions of each component.
- A baseline is a definition of a specific system.
- The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

8.14 Baselines

- Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- Baselines are important because you often have to recreate a specific version of a complete system.
- For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

8.15 Codelinesand baselines

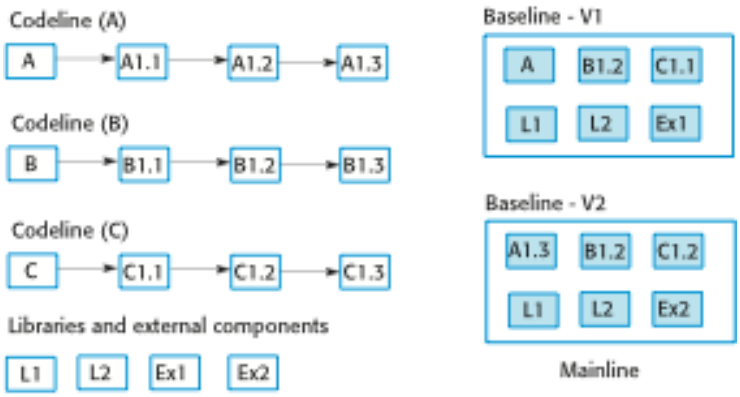

Figure 8.3:

8.16 Version management systems

- Version and release identification
- Managed versions are assigned identifiers when they are submitted to the system.
- Storage management
- To reduce the storage space required by multiple versions of components that differ only slightly, version management systems usually provide storage management facilities.
- Change history recording
- All of the changes made to the code of a system or component are recorded and listed.

8.17 Version management systems

- Independent development
- The version management system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.
- Project support

- A version management system may support the development of several projects, which share components.

8.18 Storage management using deltas

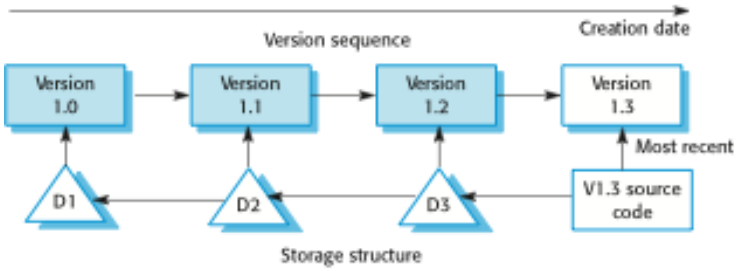

Figure 8.4:

8.19 Check-in and check-out from a version repository

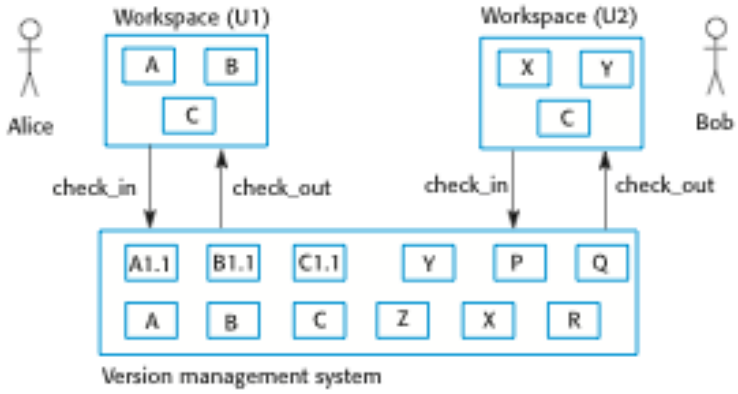

Figure 8.5:

8.20 Codeline branches

- Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.
- This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.
- If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.

8.21 Branching and merging

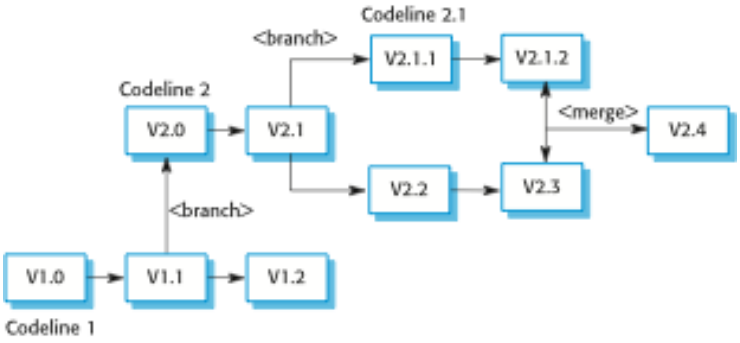

Figure 8.6:

8.22 Key points

- 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.
- The main configuration management processes are change management, version management, system building and release management.

- Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.
- Version management involves keeping track of the different versions of software components as changes are made to them.

8.23 System building

- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- The configuration description used to identify a baseline is also used by the system building tool.

8.24 B platforms

- The development system, which includes development tools such as compilers, source code editors, etc.
- Developers check out code from the version management system into a private workspace before making changes to the system.
- The build server, which is used to build definitive, executable versions of the system.
- Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.
- The target environment, which is the platform on which the system executes.

8.25 Development, build, and target platforms

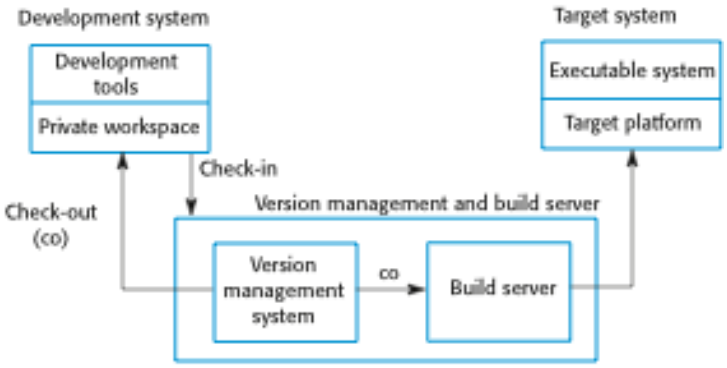

Figure 8.7:

8.26 System building

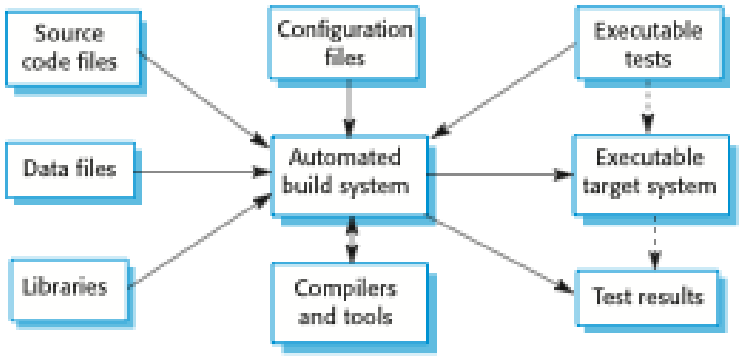

Figure 8.8:

8.27 Build system functionality

- Build script generation
- Version management system integration

- Minimal re-compilation
- Executable system creation
- Test automation
- Reporting
- Documentation generation

8.28 Minimizing recompilation

- Tools to support system building are usually designed to minimize the amount of compilation that is required.
- They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- A unique signature identifies each source and object code version and is changed when the source code is edited.
- By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

8.29 File identification

- Modification timestamps
- The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.
- Source code checksums
- The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.

8.30 Timestamps vs checksums

- Timestamps

- Because source and object files are linked by name rather than an explicit source file signature, it is not usually possible to build different versions of a source code component into the same directory at the same time, as these would generate object files with the same name.
- Checksums
- When you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible and different versions of a component may be compiled at the same time.

8.31 Agile building

- Check out the mainline system from the version management system into the developer's private workspace.
- Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
- Make the changes to the system components.
- Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

8.32 Agile building

- Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
- Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
- If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

8.33 Continuous integration

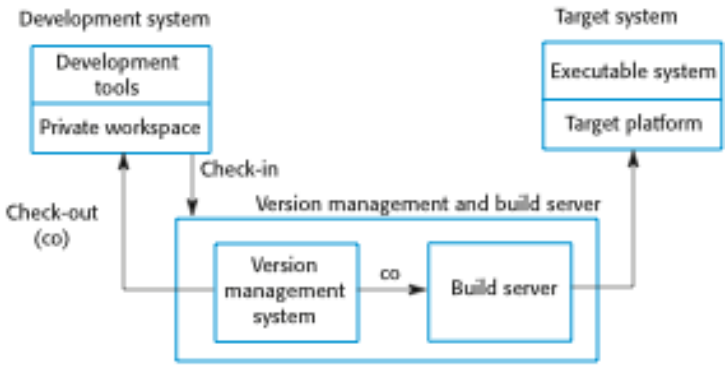

Figure 8.9:

8.34 Daily building

- The development organization sets a delivery time (say 2 p.m.) for system components.
- If developers have new versions of the components that they are writing, they must deliver them by that time.
- A new version of the system is built from these components by compiling and linking them to form a complete system.
- This system is then delivered to the testing team, which carries out a set of predefined system tests
- Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

8.35 Release management

- A system release is a version of a software system that is distributed to customers.
- For mass market software, it is usually possible to identify two types of release: major releases which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported.

- For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

8.36 Release tracking

- In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
- When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
- Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

8.37 Release reproduction

- To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
- You must keep copies of the source code files, corresponding executables and all data and configuration files.
- You should also record the versions of the operating system, libraries, compilers and other tools used to build the software.

8.38 Release planning

- As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- Release timing
- If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
- If system releases are too infrequent, market share may be lost as customers move to alternative systems.

8.39 Release components

- As well as the the executable code of the system, a release may also include:
- configuration files defining how the release should be configured for particular installations;
- data files, such as files of error messages, that are needed for successful system operation;
- an installation program that is used to help install the system on target hardware;
- electronic and paper documentation describing the system;
- packaging and associated publicitythat have been designed for that release.

8.40 Factors influencing system release planning

8.41 Key points

- System building is the process of assembling system components into an executable program to run on a target computer system.
- Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- System releases include executable code, data files, configuration files and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.

Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Lehman's fifth law (see Chapter 9)	This 'law' suggests that if you add a lot of new functionality to a system; you will also introduce bugs that will limit the amount of functionality that may be included in the next release. Therefore, a system release with significant new functionality may have to be followed by a release that focuses on repairing problems and improving performance.
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Customer change proposals	For custom systems, customers may have made and paid for a specific set of system change proposals, and they expect a system release as soon as these have been implemented.

Chapter 9

CH-26 Process improvement

9.1 Topics covered

- The process improvement process
- Process measurement
- Process analysis
- Process change
- The CMMI process improvement framework

9.2 Process improvement

- Many software companies have turned to software process improvement as a way of enhancing the quality of their software, reducing costs or accelerating their development processes.
- Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time.

9.3 Approaches to improvement

- The process maturity approach, which focuses on improving process and project management and introducing good software engineering practice.
- The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes.

- The agile approach, which focuses on iterative development and the reduction of overheads in the software process.
- The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements.

9.4 Process and product quality

- Process quality and product quality are closely related and process improvement benefits arise because the quality of the product depends on its development process.
- A good process is usually required to produce a good product.
- For manufactured goods, process is the principal quality determinant.
- For design-based activities, other factors are also involved, especially the capabilities of the designers.

9.5 Factors affecting software product quality

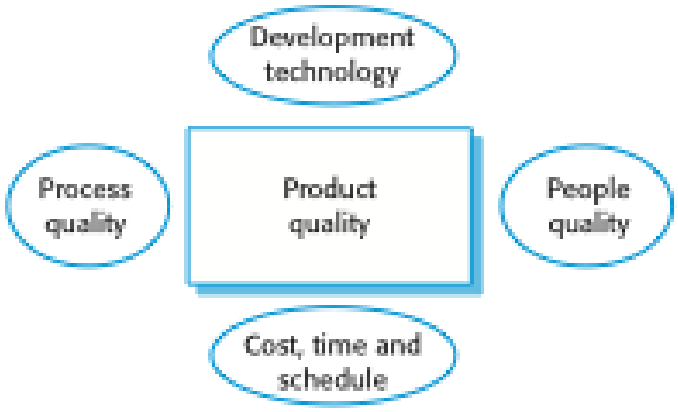

Figure 9.1:

9.6 Quality factors

- For large projects with 'average' capabilities, the development process determines product quality.
- For small projects, the capabilities of the developers is the main determinant.

- The development technology is particularly significant for small projects.
- In all cases, if an unrealistic schedule is imposed then product quality will suffer.

9.7 Process improvement process

- There is no such thing as an 'ideal' or 'standard' software process that is applicable in all organizations or for all software products of a particular type.
- You will rarely be successful in introducing process improvements if you simply attempt to change the process to one that is used elsewhere.
- You must always consider the local environment and culture and how this may be affected by process change proposals.
- Each company has to develop its own process depending on its size, the background and skills of its staff, the type of software being developed, customer and market requirements, and the company culture.

9.8 Improvement attributes

- You also have to consider what aspects of the process that you want to improve.
- Your goal might be to improve software quality and so you may wish to introduce new process activities that change the way software is developed and tested.
- You may be interested in improving some attribute of the process itself (such as development time) and you have to decide which process attributes are the most important to your company.

9.9 Process improvement stages

- Process measurement
- Attributes of the current process are measured. These are a baseline for assessing improvements.
- Process analysis
- The current process is assessed and bottlenecks and weaknesses are identified.
- Process change

- Changes to the process that have been identified during the analysis are introduced.

9.10 Process attributes

Process characteristic	Key issues
Understandability	To what extent is the process explicitly defined and how easy is it to understand the process definition?
Standardization	To what extent is the process based on a standard generic process? This may be important for some customers who require conformance with a set of defined process standards. To what extent is the same process used in all parts of a company?
Visibility	Do the process activities culminate in clear results, so that the progress of the process is externally visible?
Measurability	Does the process include data collection or other activities that allow process or product characteristics to be measured?
Supportability	To what extent can software tools be used to support the process activities?
Acceptability	Is the defined process acceptable to and usable by the engineers responsible for producing the software product?
Reliability	Is the process designed in such a way that process errors are avoided or trapped before they result in product errors?
Robustness	Can the process continue in spite of unexpected problems?
Maintainability	Can the process evolve to reflect changing organizational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?

9.11 The process improvement cycle

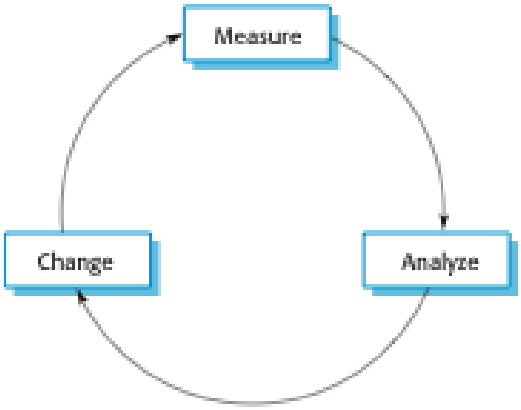

Figure 9.2:

9.12 Process measurement

- Wherever possible, quantitative process data should be collected
- However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible.
- Process measurements should be used to assess process improvements
- But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives.

9.13 Process metrics

- Time taken for process activities to be completed
- E.g. Calendar time or effort to complete an activity or process.
- Resources required for processes or activities
- E.g. Total effort in person-days.
- Number of occurrences of a particular event
- E.g. Number of defects discovered.

9.14 Goal-Question-Metric Paradigm

- Goals
- What is the organisation trying to achieve? The objective of process improvement is to satisfy these goals.
- Questions
- Questions about areas of uncertainty related to the goals. You need process knowledge to derive these.
- Metrics
- Measurements to be collected to answer the questions.

9.15 GQM questions

- The GQM paradigm is used in process improvement to help answer three critical questions:
- Why are we introducing process improvement?
- What information do we need to help identify and assess improvements?
- What process and product measurements are required to provide this information?

9.16 The GQM paradigm

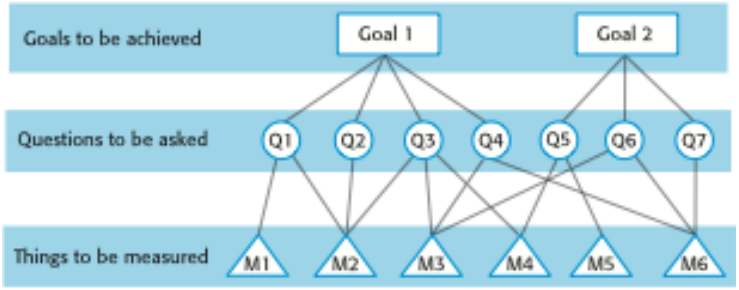

Figure 9.3:

9.17 Process analysis

- The study of existing processes to understand the relationships between parts of the process and to compare them with other processes.
- Process analysis and process measurement are intertwined.
- You need to carry out some analysis to know what to measure, and, when making measurements, you inevitably develop a deeper understanding of the process being measured.

9.18 Process analysis objectives

- To understand the activities involved in the process and the relationships between these activities.
- To understand the relationships between the process activities and the measurements that have been made.
- To relate the specific process or processes that you are analyzing to comparable processes elsewhere in the organization, or to idealized processes of the same type.

9.19 Process analysis techniques

- Published process models and process standards
- It is always best to start process analysis with an existing model. People then may extend and change this.
- Questionnaires and interviews
- Must be carefully designed. Participants may tell you what they think you want to hear.
- Ethnographic analysis
- Involves assimilating process knowledge by observation. Best for in-depth analysis of process fragments rather than for whole-process understanding.

9.20 Aspects of process analysis

Process aspect	Questions
Adoption and standardization	Is the process documented and standardized across the organization? If not, does this mean that any measurements made are specific only to a single process instance? If processes are not standardized, then changes to one process may not be transferable to comparable processes elsewhere in the company.
Software engineering practice	Are there known, good software engineering practices that are not included in the process? Why are they not included? Does the lack of these practices affect product characteristics, such as the number of defects in a delivered software system?
Organizational constraints	What are the organizational constraints that affect the process design and the ways that the process is performed? For example, if the process involves dealing with classified material, there may be activities in the process to check that classified information is not included in any material due to be released to external organizations. Organizational constraints may mean that possible process changes cannot be made.
Communications	How are communications managed in the process? How do communication issues relate to the process measurements that have been made? Communication problems are a major issue in many processes and communication bottlenecks are often the reasons for project delays.
Introspection	Is the process reflective (i.e., do the actors involved in the process explicitly think about and discuss the process and how it might be improved)? Are there mechanisms through which process actors can propose process improvements?
Learning	How do people joining a development team learn about the software processes used? Does the company have process manuals and process training programs?
Tool support	What aspects of the process are and aren't supported by software tools? For unsupported areas, are there tools that could be deployed cost-effectively to provide support? For supported areas, are the tools effective and efficient? Are better tools available?

Table 9.1:

9.21 Process models

- Process models are a good way of focusing attention on the activities in a process and the information transfer between these activities.
- Process models do not have to be formal or complete – their purpose is to provoke discussion rather than document the process in detail.
- Model-oriented questions can be used to help understand the process e.g.
- What activities take place in practice but are not shown in the model?
- Are there process activities, shown in the model, that you (the process actor) think are inefficient?

9.22 Process exceptions

- Software processes are complex and process models cannot effectively represent how to handle exceptions:
- Several key people becoming ill just before a critical review;
- A breach of security that means all external communications are out of action for several days;
- Organisational reorganisation;
- A need to respond to an unanticipated request for new proposals.
- Under these circumstances, the model is suspended and managers use their initiative to deal with the exception.

9.23 Key points

- The goals of process improvement are higher product quality, reduced process costs and faster delivery of software.
- The principal approaches to process improvement are agile approaches, geared to reducing process overheads, and maturity-based approaches based on better process management and the use of good software engineering practice.
- The process improvement cycle involves process measurement, process analysis and modeling, and process change.
- Measurement should be used to answer specific questions about the software process used. These questions should be based on organizational improvement goals.

9.24 Process change

- Involves making modifications to existing processes.
- This may involve:
- Introducing new practices, methods or processes;
- Changing the ordering of process activities;
- Introducing or removing deliverables;
- Introducing new roles or responsibilities.
- Change should be driven by measurable goals.

9.25 The process change process

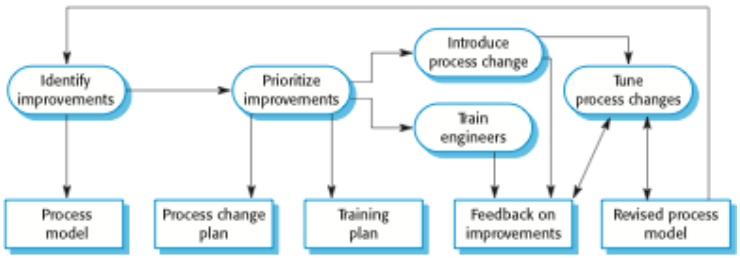

Figure 9.4:

9.26 Process change stages

- Improvement identification
- This stage is concerned with using the results of the process analysis to identify ways to tackle quality problems, schedule bottlenecks or cost inefficiencies that have been identified during process analysis.
- Improvement prioritization
- When many possible changes have been identified, it is usually impossible to introduce them all at once, and you must decide which are the most important.

- Process change introduction
- Process change introduction means putting new procedures, methods and tools into place and integrating them with other process activities.

9.27 Process change stages

- Process change training
- Without training, it is not possible to gain the full benefits of process changes. The engineers involved need to understand the changes that have been proposed and how to perform the new and changed processes.
- Change tuning
- Proposed process changes will never be completely effective as soon as they are introduced. You need a tuning phase where minor problems can be discovered, and modifications to the process can be proposed and introduced.

9.28 Process change problems

- Resistance to change
- Team members or project managers may resist the introduction of process changes and propose reasons why changes will not work, or delay the introduction of changes. They may, in some cases, deliberately obstruct process changes and interpret data to show the ineffectiveness of proposed process change.
- Change persistence
- While it may be possible to introduce process changes initially, it is common for process innovations to be discarded after a short time and for the processes to revert to their previous state.

9.29 Resistance to change

- Project managers often resist process change because any innovation has unknown risks associated with it.
- Project managers are judged according to whether or not their project produces software on time and to budget. They may prefer an inefficient but predictable process to an improved process that has organizational benefits, but which has short-term risks associated with it.
- Engineers may resist the introduction of new processes for similar reasons, or because they see these processes as threatening their professionalism.

- That is, they may feel that the new pre-defined process gives them less discretion and does not recognize the value of their skills and experience.

9.30 Change persistence

- The problem of changes being introduced then subsequently discarded is a common one.
- Changes may be proposed by an 'evangelist' who believes strongly that the changes will lead to improvement. He or she may work hard to ensure the changes are effective and the new process is accepted.
- If the 'evangelist' leaves, then the people involved may therefore simply revert to the previous ways of doing things.
- Change institutionalization is important
- This means that process change is not dependent on individuals but that the changes become part of standard practice in the company, with company-wide support and training.

9.31 The CMMI process improvement framework

- The CMMI framework is the current stage of work on process assessment and improvement that started at the Software Engineering Institute in the 1980s.
- The SEI's mission is to promote software technology transfer particularly to US defence contractors.
- It has had a profound influence on process improvement
- Capability Maturity Model introduced in the early 1990s.
- Revised maturity framework (CMMI) introduced in 2001.

9.32 The SEI capability maturity model

- Initial
- Essentially uncontrolled
- Repeatable
- Product management procedures defined and used
- Defined

- Process management procedures and strategies defined and used
- Managed
- Quality management strategies defined and used
- Optimising
- Process improvement strategies defined and used

9.33 Process capability assessment

- Intended as a means to assess the extent to which an organisation's processes follow best practice.
- By providing a means for assessment, it is possible to identify areas of weakness for process improvement.
- There have been various process assessment and improvement models but the SEI work has been most influential.

9.34 The CMMI model

- An integrated capability model that includes software and systems engineering capability assessment.
- The model has two instantiations
- Staged where the model is expressed in terms of capability levels;
- Continuous where a capability rating is computed.

9.35 CMMI model components

- Process areas
- 24 process areas that are relevant to process capability and improvement are identified. These are organised into 4 groups.
- Goals
- Goals are descriptions of desirable organisational states. Each process area has associated goals.
- Practices
- Practices are ways of achieving a goal - however, they are advisory and other approaches to achieve the goal may be used.

9.36 Process areas in the CMMI

Category	Process area
Process management	• Organizational process definition (OPD) • Organizational process focus (OPF) • Organizational training (OT) • Organizational process performance (OPP) • Organizational innovation and deployment (OID)
Project management	• Project planning (PP) • Project monitoring and control (PMC) • Supplier agreement management (SAM) • Integrated project management (IPM) • Risk management (RSKM) • Quantitative project management (QPM)
Engineering	• Requirements management (REQM) • Requirements development (RD) • Technical solution (TS) • Product integration (PI) • Verification (VER) • Validation (VAL)

Support	• Configuration management (CM) • Process and product quality management (PPQA) • Measurement and analysis (MA) • Decision analysis and resolution (DAR) • Causal analysis and resolution (CAR)

Table 9.2:

9.37 Goals and associated practices in the CMMI

Category	Process area
Process management	• Organizational process definition (OPD) • Organizational process focus (OPF) • Organizational training (OT) • Organizational process performance (OPP) • Organizational innovation and deployment (OID)

Project management	• Project planning (PP) • Project monitoring and control (PMC) • Supplier agreement management (SAM) • Integrated project management (IPM) • Risk management (RSKM) • Quantitative project management (QPM)
Engineering	• Requirements management (REQM) • Requirements development (RD) • Technical solution (TS) • Product integration (PI) • Verification (VER) • Validation (VAL)
Support	• Configuration management (CM) • Process and product quality management (PPQA) • Measurement and analysis (MA) • Decision analysis and resolution (DAR) • Causal analysis and resolution (CAR)

Table 9.3:

9.38 Goals and associated practices in the CMMI

Goal	Associated practices
The requirements are analyzed and validated, and a definition of the required functionality is developed.	Analyze derived requirements systematically to ensure that they are necessary and sufficient.
	Validate requirements to ensure that the resulting product will perform as intended in the user's environment, using multiple techniques as appropriate.
Root causes of defects and other problems are systematically determined.	Select the critical defects and other problems for analysis.
	Perform causal analysis of selected defects and other problems and propose actions to address them.
The process is institutionalized as a defined process.	Establish and maintain an organizational policy for planning and performing the requirements development process.

9.39 CMMI assessment

- Examines the processes used in an organisation and assesses their maturity in each process area.
- Based on a 6-point scale:
- Not performed;
- Performed;
- Managed;
- Defined;
- Quantitatively managed;
- Optimizing.

9.40 Examples of goals in the CMMI

Goal	Process area
Corrective actions are managed to closure when the project's performance or results deviate significantly from the plan.	Project monitoring and control (specific goal)
Actual performance and progress of the project are monitored against the project plan.	Project monitoring and control (specific goal)
The requirements are analyzed and validated, and a definition of the required functionality is developed.	Requirements development (specific goal)
Root causes of defects and other problems are systematically determined.	Causal analysis and resolution (specific goal)
The process is institutionalized as a defined process.	Generic goal

9.41 The staged CMMI model

- Comparable with the software CMM.
- Each maturity level has process areas and goals. For example, the process area associated with the managed level include:
- Requirements management;
- Project planning;
- Project monitoring and control;
- Supplier agreement management;
- Measurement and analysis;
- Process and product quality assurance.

9.42 The CMMI staged maturity model

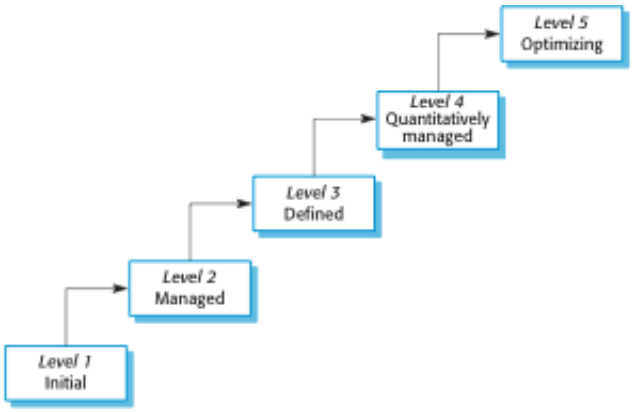

Figure 9.5:

9.43 Institutional practices

- Institutions operating at the managed level should have institutionalised practices that are geared to standardisation.
- Establish and maintain policy for performing the project management process;
- Provide adequate resources for performing the project management process;
- Monitor and control the project planning process;
- Review the activities, status and results of the project planning process.

9.44 The continuous CMMI model

- This is a finer-grain model that considers individual or groups of practices and assesses their use.
- The maturity assessment is not a single value but is a set of values showing the organisations maturity in each area.
- The CMMI rates each process area from levels 1 to 5.
- The advantage of a continuous approach is that organisations can pick and choose process areas to improve according to their local needs.

9.45 A process capability profile

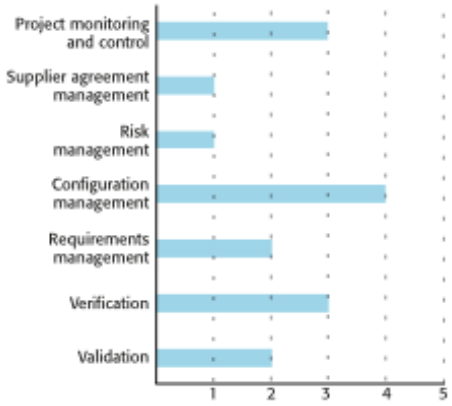

Figure 9.6:

9.46 Key points

- The CMMI process maturity model is an integrated process improvement model that supports both staged and continuous process improvement.
- Process improvement in the CMMI model is based on reaching a set of goals related to good software engineering practice and describing, standardizing and controlling the practices used to achieve these goals.
- The CMMI model includes recommended practices that may be used, but these are not obligatory.