Software Engineering

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

Contents - The Basic Part

Chapter 1. Introduction

Topics Covered

- Professional software development
- Case studies

Professional Software Development

Software Project Failures

- **Increasing system complexity**
  - As new software engineering techniques help us to build larger, more complex systems, the demands change.
  - Systems must be built and delivered more quickly.
  - Larger and even more complex systems are required.
  - Systems must have new capabilities that were previously thought to be impossible.

- Software product is often more expensive and less reliable than it should be.
  - Due to failure to use **software engineering methods**
    - It is easy to write computer programs without using software engineering methods and techniques.
    - Many companies do not use software engineering methods in their everyday work.

Software Engineering

- **Software engineering** is concerned with theories, methods and tools for professional and cost-effective software development.

- **Software costs** often dominate computer system(hardware) costs.
  - More to maintain than to develop
  - For systems with a long life, maintenance costs may be several times development costs.

Frequently Asked Questions about Software Engineering

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

Frequently Asked Questions about Software Engineering

Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.
What differences has the web made to software engineering?	The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

Software Products

- **Generic products**
  - Stand-alone systems that are marketed and sold to any customer who wishes to buy them.
  - Examples :
    - PC software such as graphics programs, project management tools, CAD software, and software for specific markets such as appointments systems for dentists
  - Software specification is owned by the software developer.
    - Decisions on software change are made by the developer.

- **Customized products**
  - Software that is commissioned by a specific customer to meet their own needs.
  - Examples :
    - Embedded control systems, air traffic control software, traffic monitoring systems
  - Software specification is owned by the customer for the software.
    - Customers make decisions on software changes that are required.

Software Engineering

- **Software engineering** is an *engineering discipline* that is concerned with all *aspects of software production* from the early stages of system specification through to maintaining the system after it has gone into use.

- "***Engineering discipline***"
  - Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.

- "***All aspects of software production***"
  - Not just technical process of development, but also project management and the development of tools, methods, etc. to support software production.
  - Most costs are for changing the software after it has gone into use.

Typical Activities in Software Engineering

- **Software specification**
  - Customers and engineers define the software to produce and the constraints on its operation.
- **Software development**
  - The software is designed and programmed.
- **Software validation**
  - The software is checked to ensure that it is what the customer requires.
- **Software evolution**
  - The software is modified to reflect changing customer and market requirements.

General Issues Affecting Software

- Heterogeneity
  - Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.
- Business and social change
  - Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software.
- Security and trust
  - As software is intertwined with all aspects of our lives, it is essential that we can trust that software.
- Scale
  - Software should be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

Software Engineering Diversity

- There are many different types of software system.
  - **No universal set of software techniques** that is applicable to all software types.

- The software engineering methods and tools used depend on
  - The **type** of application being developed, the **requirements** of customers, and the **background** of development teams.

- **Application Types**
  - Stand-alone applications
  - Interactive transaction-based applications
  - Embedded control systems
  - Batch processing systems
  - Entertainment systems
  - Systems for modelling and simulation
  - Data collection systems
  - Systems of systems

Application Types

Type	Features
Stand-alone applications	Application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to network.
Interactive transaction-based applications	Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.
Embedded control systems	Software control systems that control and manage hardware devices.
Batch processing systems	Business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.
Entertainment systems	Systems that are primarily for personal use and which are intended to entertain the user.
Systems for modelling and simulation	Systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.
Data collection systems	Systems that collect data from their environment using a set of sensors and send that data to other systems for processing.
Systems of systems	Systems that are composed of a number of other software systems.

Fundamentals of Software Engineering

- Fundamental principles applicable to all types of software system, irrespective of the development techniques used:
  - ***"Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software."***
  - ***"Dependability and performance are important for all types of system."***
  - ***"Understanding and managing the software specification and requirements are important."***
  - ***"Where appropriate, you should reuse software that has already been developed rather than write new software."***

Web-based Software Engineering

- The **Web** is now a **platform** for running various application.
  - **Web services** (discussed in Chapter 19) allow application functionality to be accessed over the web.
  - **Cloud computing** is an approach to the provision of computer services where applications run remotely on the 'cloud'.
    - Users do not buy software but pay according to use.

- **Web-based systems**
  - Complex distributed systems
  - Fundamental ideas of software engineering can be applied in the same way.
    - Software reuse
    - Incremental and agile development
    - Service-oriented systems
    - Rich interfaces

Developing Web-based Software Systems

- **Software reuse**
  - Software reuse is the dominant approach for constructing web-based systems.
  - When building these systems, you think about how you can assemble them from pre-existing software components and systems.
- **Incremental and agile development**
  - Web-based systems should be developed and delivered incrementally.
  - It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.
- **Service-oriented systems**
  - Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services.
- **Rich interfaces**
  - Interface development technologies such as AJAX and HTML5 have emerged that support the creation of rich interfaces within a web browser.

Case Studies

4 Case Studies

- **Personal insulin pump control system**
  - An embedded system in an insulin pump used by diabetics to maintain blood glucose control
- **Mentcare: Mental health case patient management system**
  - A system used to maintain records of people receiving care for mental health problems
- **Wilderness weather station**
  - A data collection system that collects data about weather conditions in remote areas
- **iLearn: a digital learning environment**
  - A system to support learning in schools

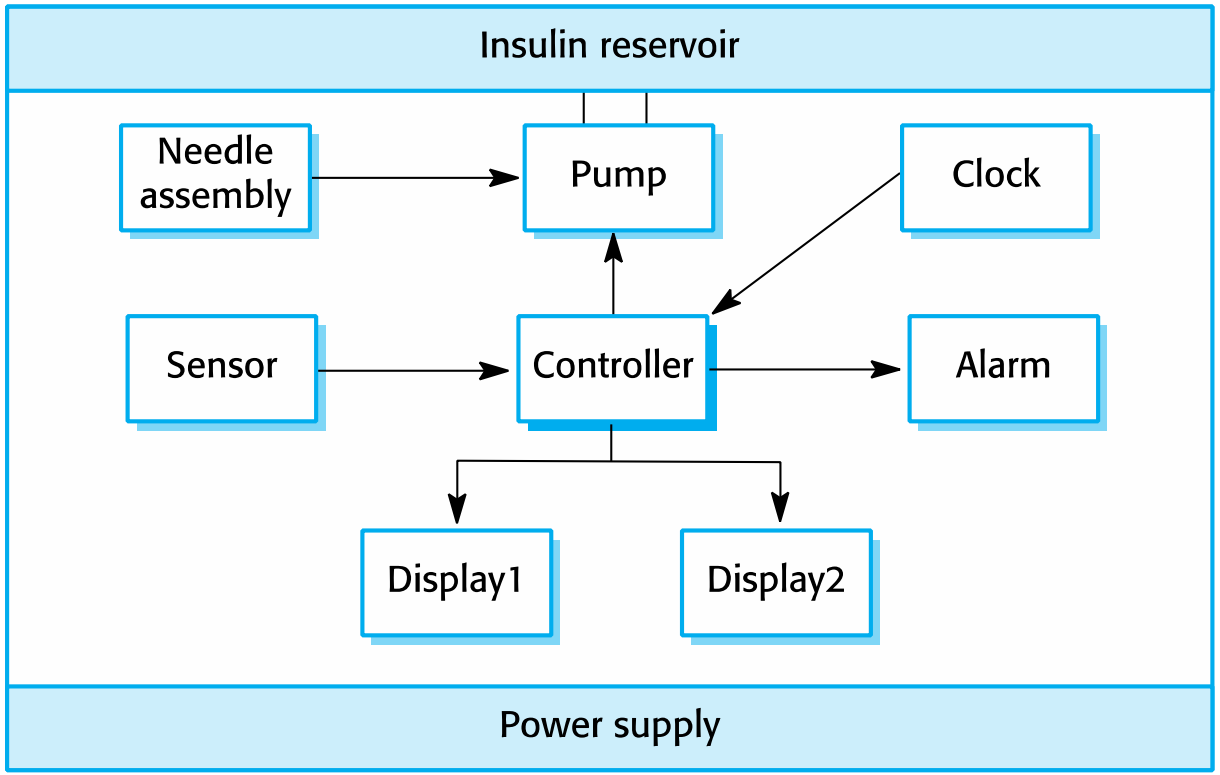
Insulin Pump Control System

- **A safety-critical system**
  - Low blood sugars can lead to brain malfunctioning, coma and death.
  - High-blood sugars have long-term consequences like eye and kidney damage.

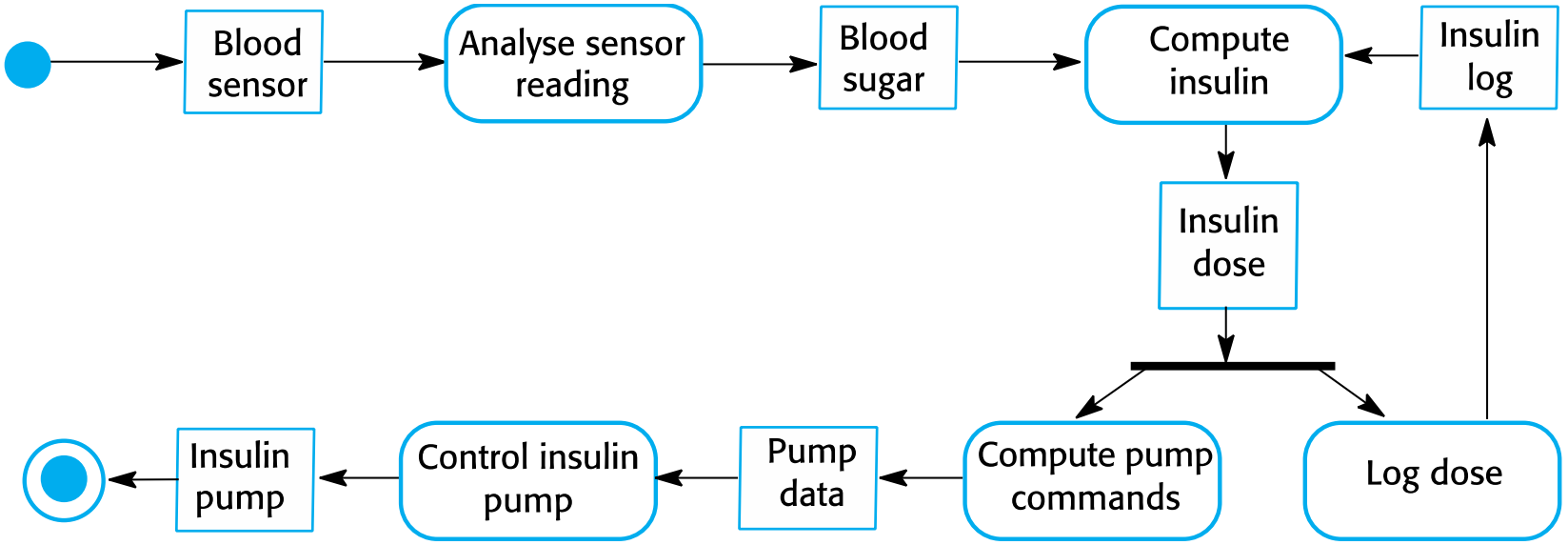
- Features
  - Collects data from a blood sugar sensor and calculates the amount of insulin required to be injected.
  - Calculation is based on the rate of change of blood sugar levels.
  - Sends signals to a micro-pump to deliver the correct dose of insulin.

- Essential high-level requirements
  - The system shall be available to deliver insulin when required.
  - The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

Insulin Pump : Hardware Architecture



Insulin Pump : Activity Model



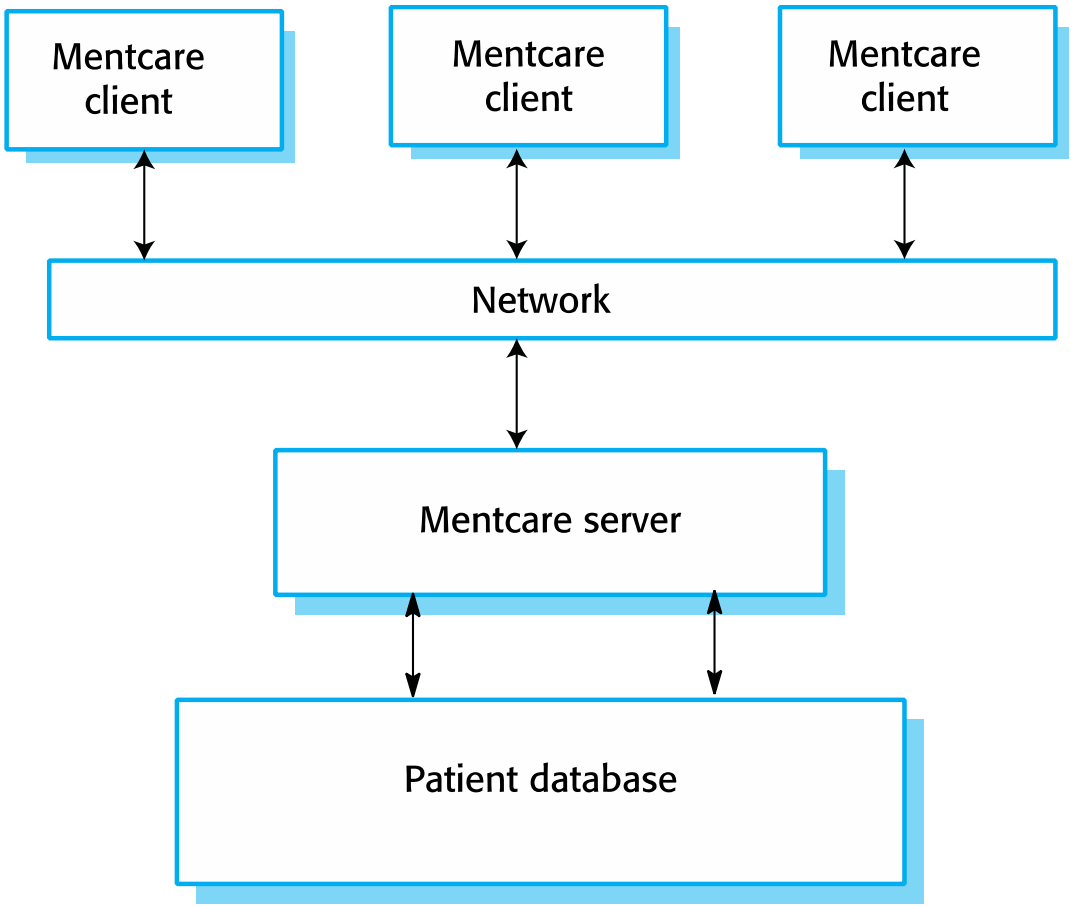
Mentcare: A Patient Information System for Mental Health Care

- **A medical information system**
  - Maintains information about patients suffering from mental health problems and the treatments that they have received.
  - Intended for use in clinics

- Motivations
  - Most mental health patients do not require dedicated hospital treatment, but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems.
  - To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centres.

- Features
  - It makes use of a centralized database of patient information, but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
  - When the local systems have secure network access, they use patient information in the database, but they can download and use local copies of patient records when they are disconnected.

Mentcare System : Overall Organization



Mentcare System : Key Features

- Goals
  - To generate management information that allows health service managers to assess performance against local and government targets.
  - To provide medical staff with timely information to support the treatment of patients.

- Key features to meet the goals
  - Individual care management
    - Clinicians can create records for patients, edit the information in the system, and view patient history.
    - The system supports data summaries so that doctors can quickly learn about the key problems and treatments that have been prescribed.
  - Patient monitoring
    - The system monitors the records of patients that are involved in treatment and issues warnings if any potential problems are detected.
  - Administrative reporting
    - The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, and the drugs prescribed and their costs.

Mentcare System : Other Concerns

- **Privacy**
  - It is essential that patient information is confidential and is never disclosed to anyone apart from authorized medical staff and the patient themselves.

- **Safety**
  - Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
  - The system must be available when needed, otherwise safety may be compromised, and it may be impossible to prescribe the correct medication to patients.

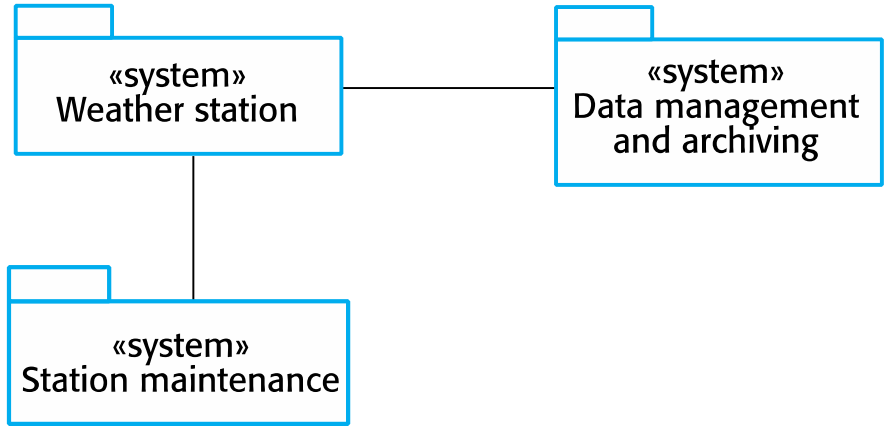
Wilderness Weather Station

- The government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.

- Weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.
  - The weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

Weather Station : System Organization

- The weather station systems
  - This is responsible for collecting weather data, carrying out some initial data processing, and transmitting it to the data management system.
- The data management and archiving system
  - This system collects the data from all of the wilderness weather stations, carries out data processing and analysis, and archives the data.
- The station maintenance system
  - This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems.



Weather Station : Additional Software Functionality

- Additional Software Functionality
  - Monitor the instruments, power, and communication hardware, and report faults to the management system.
    - Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit, but also that generators are shutdowned in potentially damaging weather conditions, such as high wind.
  - Support dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

iLearn: A Digital Learning Environment

- A digital learning environment
  - A framework in which a set of general-purpose and specially designed tools for learning may be embedded
  - A set of applications are geared to the needs of the learners using the system.

- Features
  - The tools included in each version of the environment are chosen by teachers and learners to suit their specific needs.
    - These can be general applications such as spreadsheets, learning management applications such as a Virtual Learning Environment (VLE) to manage homework submission and assessment, games and simulations.

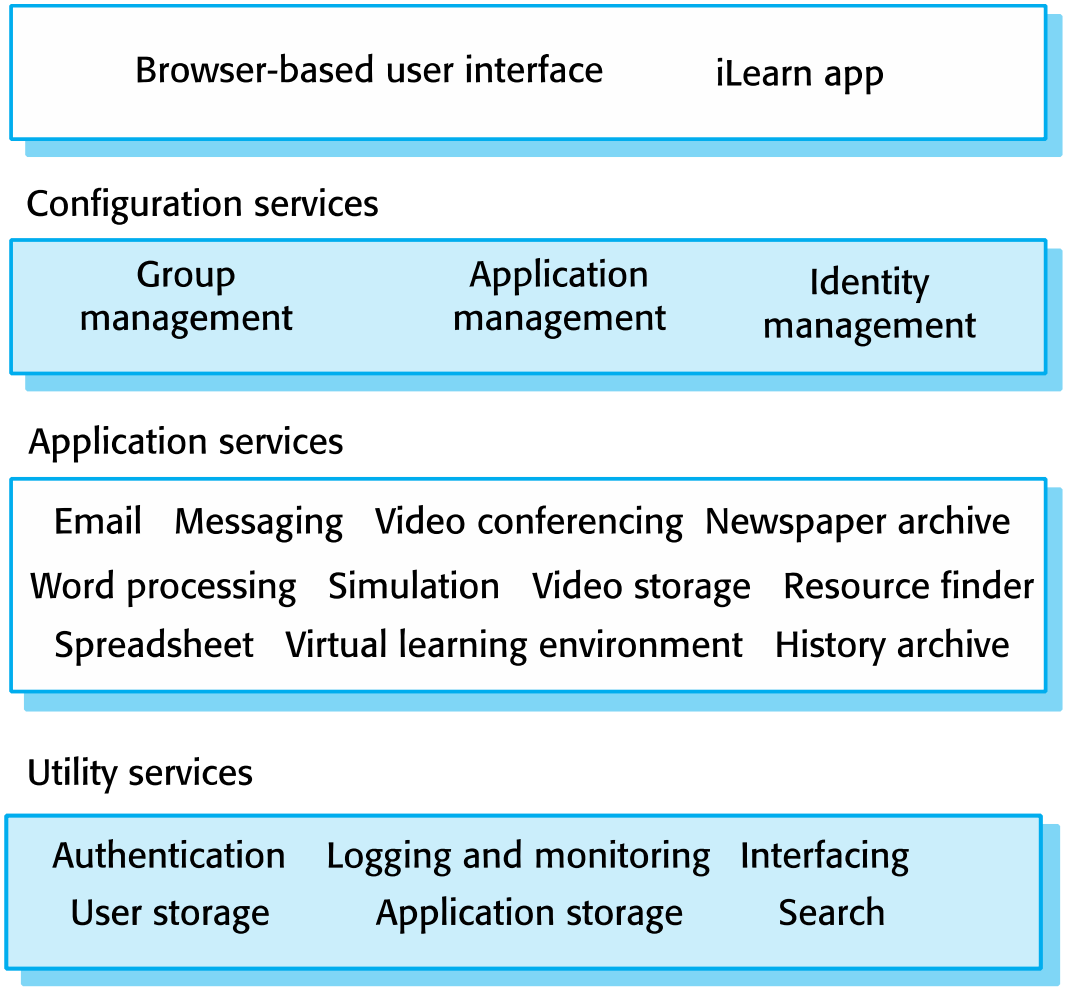
- iLearn is a service-oriented system.
  - All system components considered to be a replaceable service.
  - The system can be updated incrementally as new services become available.
  - The system can rapidly configure the system itself to create versions of the environment for different groups such as very young children who cannot read, senior students, etc.

iLearn : Service-Oriented Systems

- iLearn services
  - *Utility services*
    - Provide basic application-independent functionality and which may be used by other services in the system.
  - *Application services*
    - Provide specific applications such as email, conferencing, photo sharing etc. and access to specific educational content such as scientific films or historical resources.
  - *Configuration services*
    - Used to adapt the environment with a specific set of application services and do define how services are shared between students, teachers and their parents.

- iLearn service for integration
  - *Integrated services*
    - Services which offer an API (application programming interface) and which can be accessed by other services through that API.
    - Direct service-to-service communication is therefore possible.
  - *Independent services*
    - Services which are simply accessed through a browser interface and which operate independently of other services.
    - Information can only be shared with other services through explicit user actions such as copy and paste; re-authentication may be required for each independent service.

iLearn : Architecture



Key Points

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Essential software product attributes are maintainability, dependability and security, efficiency and acceptability.
- The high-level activities of specification, development, validation and evolution are part of all software processes.
- The fundamental notions of software engineering are universally applicable to all types of system development.
- There are many different types of system and each requires appropriate software engineering tools and techniques for their development.
- The fundamental ideas of software engineering are applicable to all types of software system.

Chapter 2. Software Processes

Topics Covered

- Software process models
- Process activities
- Coping with change
- Process improvement

Software Process

- A structured set of activities required to develop a software system

- Many different software processes but all involve:
  - **Specification** : defining what the system should do
  - **Design and implementation** : defining the organization of the system and implementing the system
  - **Validation** : checking that it does what the customer wants
  - **Evolution** : changing the system in response to changing customer needs.

- **Software process model** is an abstract representation of a process, presenting a description of a process from some perspectives.
  - Waterfall → Plan-driven
  - Iterative → UP/Agile

Plan-Driven vs. Agile Processes

- **Plan-driven process**
  - All process activities are planned in advance.
  - Progress is measured against this plan.

- **Agile process**
  - Planning is incremental.
  - It is easier to change the process to reflect changing customer requirements.

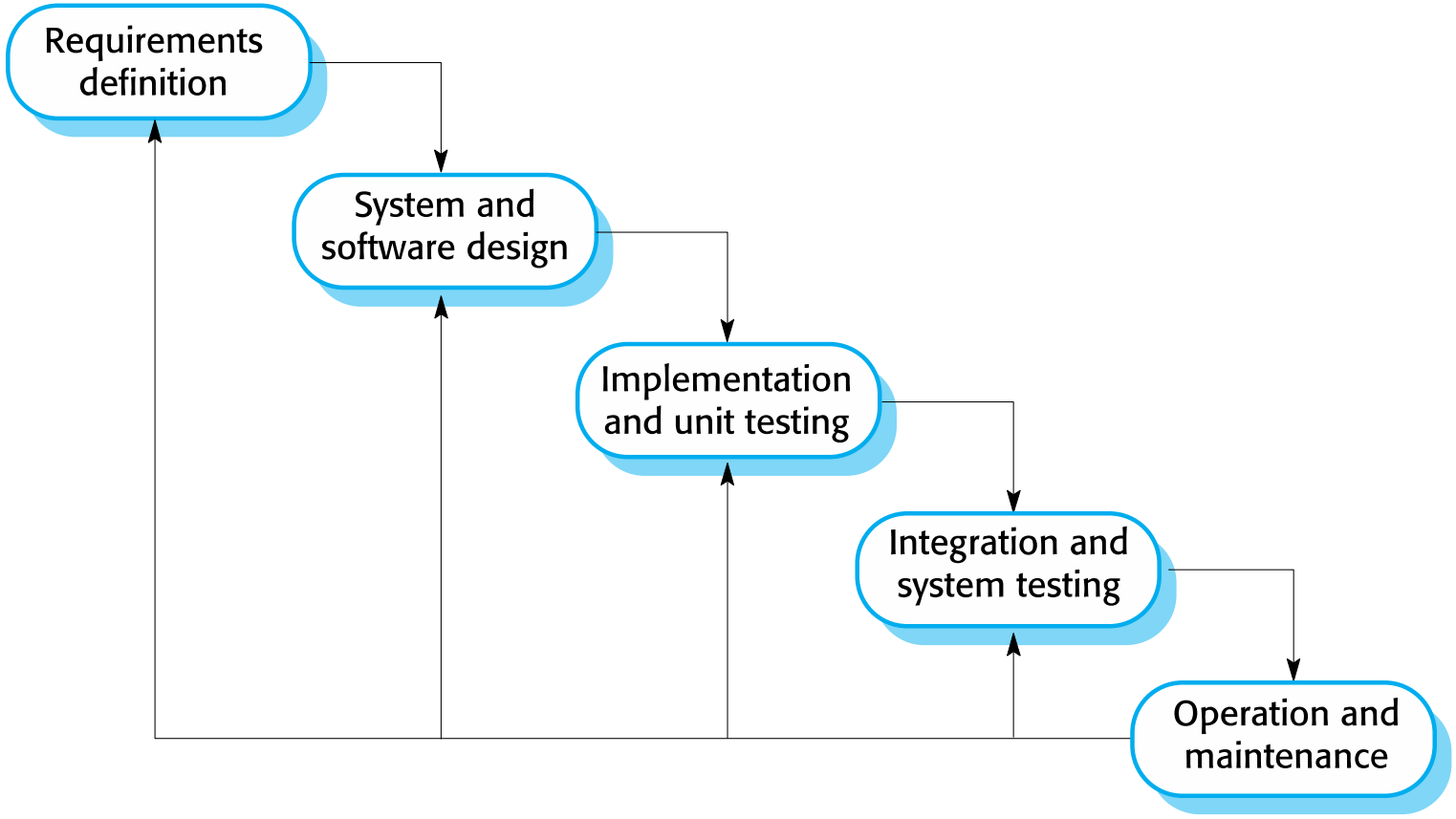
- In practice, most practical processes include elements of both plan-driven and agile approaches.
  - There are no right or wrong software processes.

Software Process Models

Software Process Models

- **Waterfall model**
  - Plan-driven model
  - Separate and distinct phases of specification and development
- **Incremental development**
  - Specification, development and validation are interleaved.
  - May be plan-driven or agile.
- **Integration and configuration (aka CBD(Component-Based Development))**
  - The system is assembled from existing configurable components.
  - May be plan-driven or agile.
- In practice, most large systems are developed using a process that incorporates elements from all of these models.

The Waterfall Model

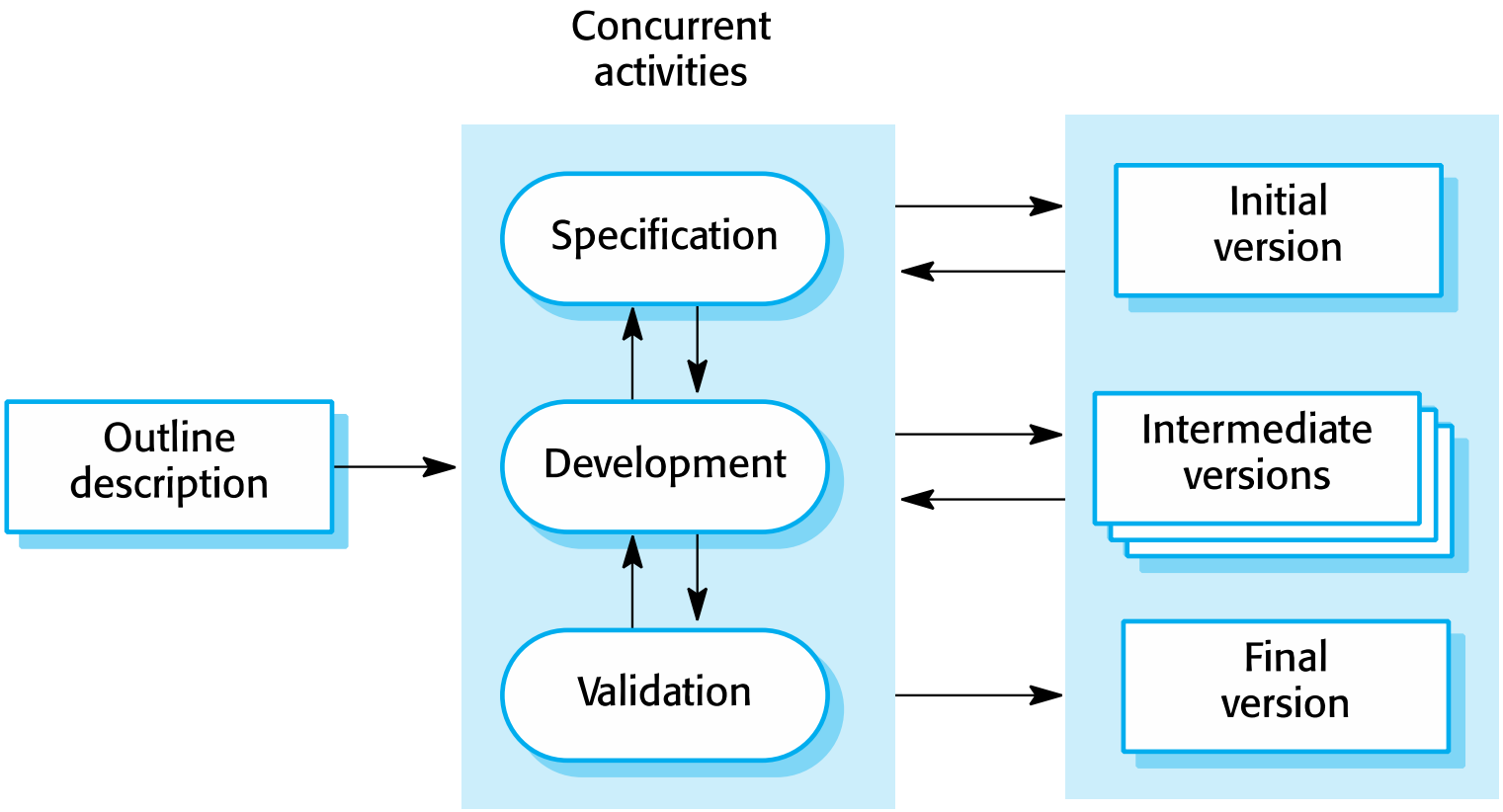


The Waterfall Model

- It has distinct/separated phases.
  - In principle, a phase has to be complete before moving onto the next phase.
  - Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
  - Only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
    - Few business systems have stable requirements.

- 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



Incremental Development

- A number of increments are developed in parallel.
  - Each increment is developed independent with each other, and integrated later.
  - The cost of accommodating changing customer requirements can be reduced.
    - Easier to get customer feedback on the development work.
  - More rapid delivery and deployment of useful software to the customer is possible.

- The process is not visible.
  - Many increments are developed concurrently.
  - Documentations are not easy.
  - System structure tends to degrade as new increments are added.
    - Regular change tends to corrupt its structure.
    - Incorporating further software changes becomes increasingly difficult and costly.

Integration and Configuration

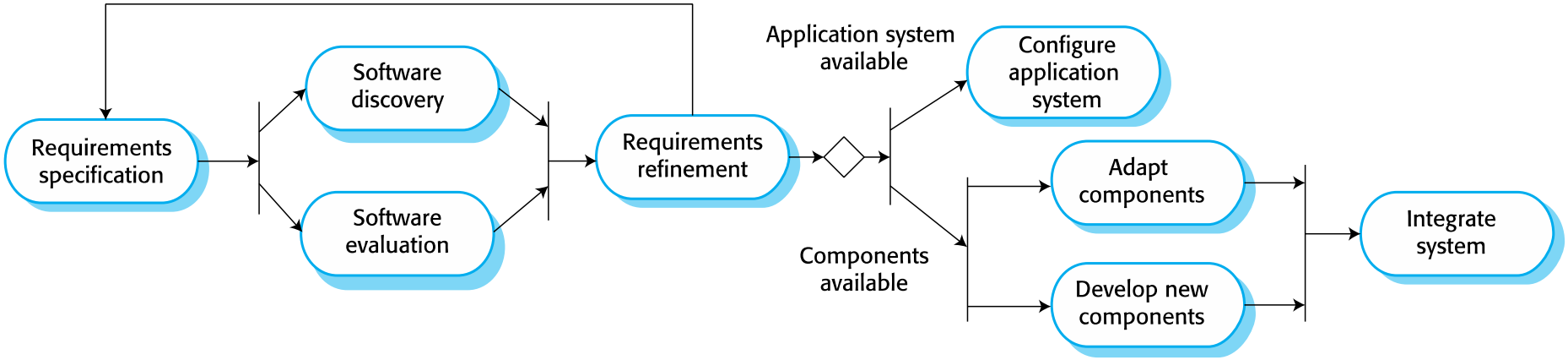
- It is based on software reuse.
  - Systems are integrated from existing components or application systems (sometimes called **COTS** (Commercial-off-the-shelf) systems).
  - Reused elements should be configured to adapt their behaviour and functionality.

- Reuse is now the standard approach for building many types of business systems.

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

Integration and Configuration

- Reuse-oriented software engineering process



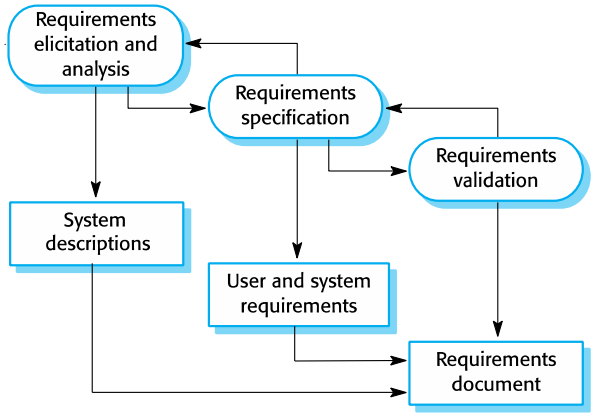
- **Advantages :**
  - Reduced costs and risks as less software is developed from scratch.
  - Faster delivery and deployment of systems are possible.

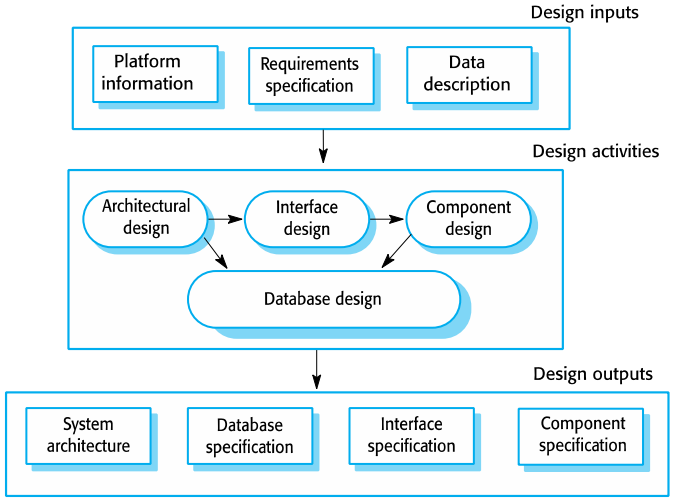
- **Disadvantages :**
  - Requirements compromises are inevitable, so system may not meet real needs of users.
  - Loss of control over evolution of reused system elements

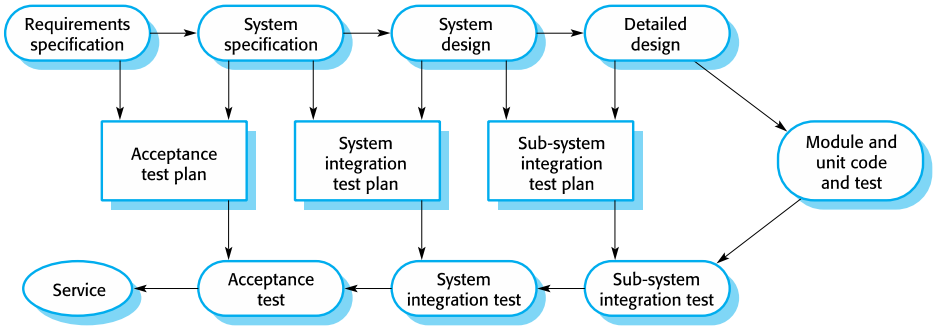
Process Activities

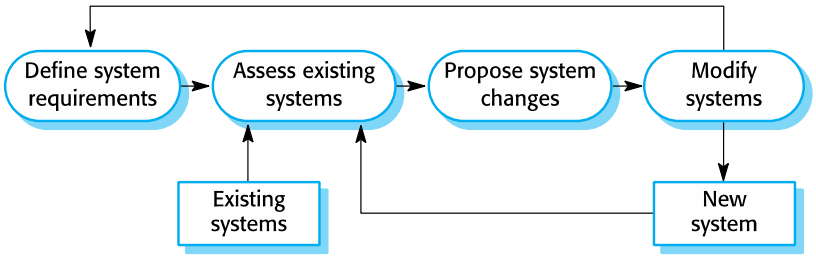
Process Activities

- The four basic process activities of specification, development, validation and evolution are organized differently in different development processes.







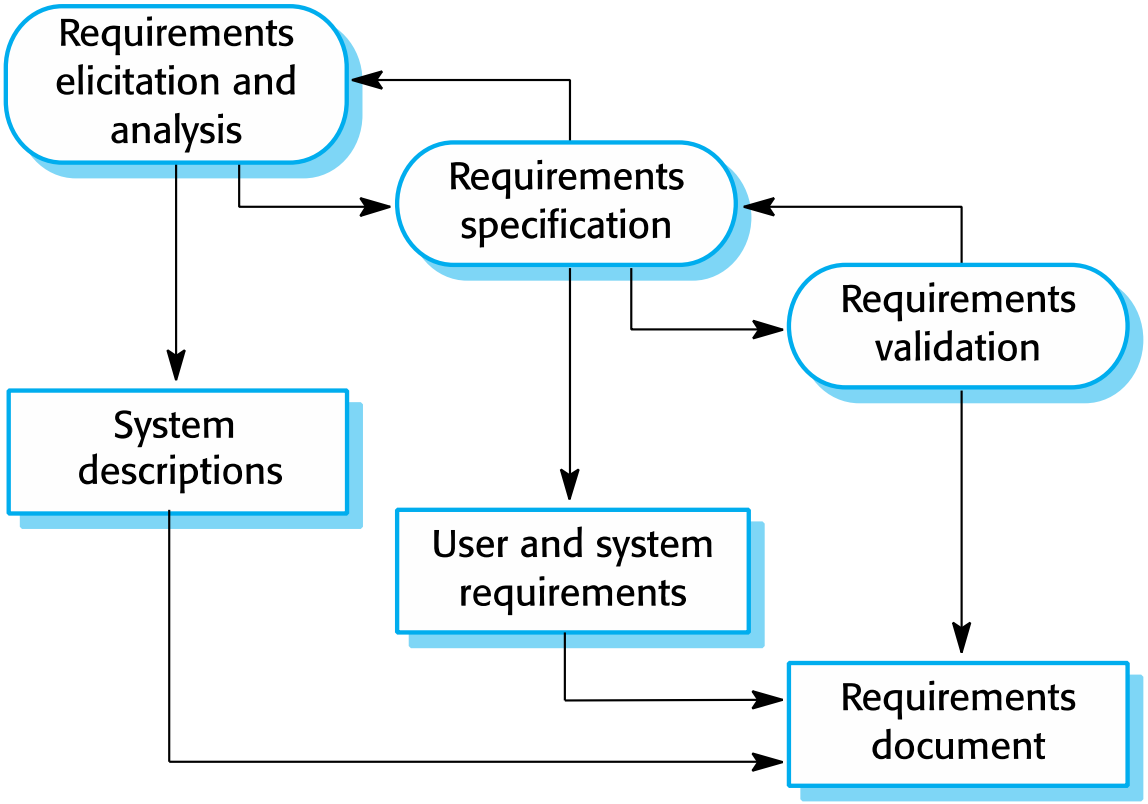


1. Requirements Engineering Process

- **RE (Requirements Engineering)**
  - The process of establishing what services are required and the constraints on the system's operation and development

- Requirements engineering process
  - Requirements elicitation and analysis
    - "What do the system stakeholders require or expect from the system?"
  - Requirements specification
    - "Defining the requirements in detail"
  - Requirements validation
    - "Checking the validity of the requirements"

Requirements Engineering Process

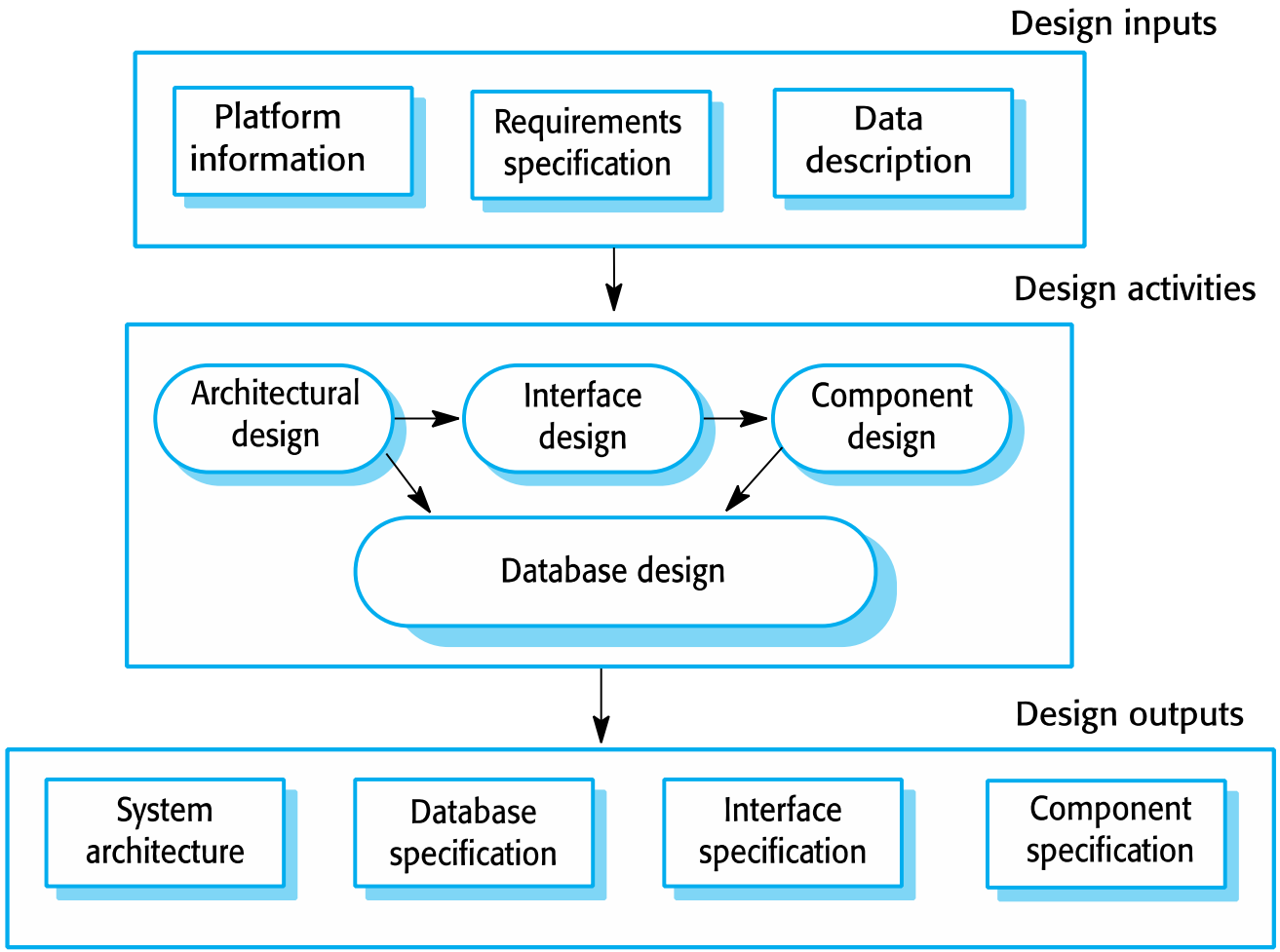


2. Software Design and Implementation

- The process of converting the system specification into an executable system.
  - Software design
    - Design a software structure that realizes the specification
  - Implementation
    - Translate this structure into an executable program

- The activities of design and implementation are closely related and may be inter-leaved.

A General Model of Design Process



Design Activities

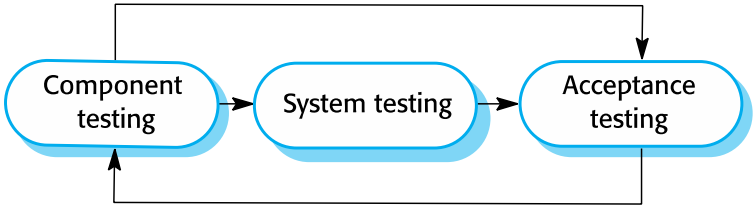
- Architectural design
  - Identify the overall structure of the system, the principal components (subsystems or modules), their relationships, and how they are distributed.
- Database design
  - Design the system data structures and how these are to be represented in a database.
- Interface design
  - Define the interfaces between system components.
- Component selection and design
  - Search for reusable components. If unavailable, you design how it will operate.

Implementation Activities

- The software is implemented either by developing a program or programs or by configuring application system.
  - Programming
    - An individual activity with no standard process.
  - Debugging
    - An activity of finding program faults and correcting these faults.
  - Design and implementation are interleaved activities for most types of software system.

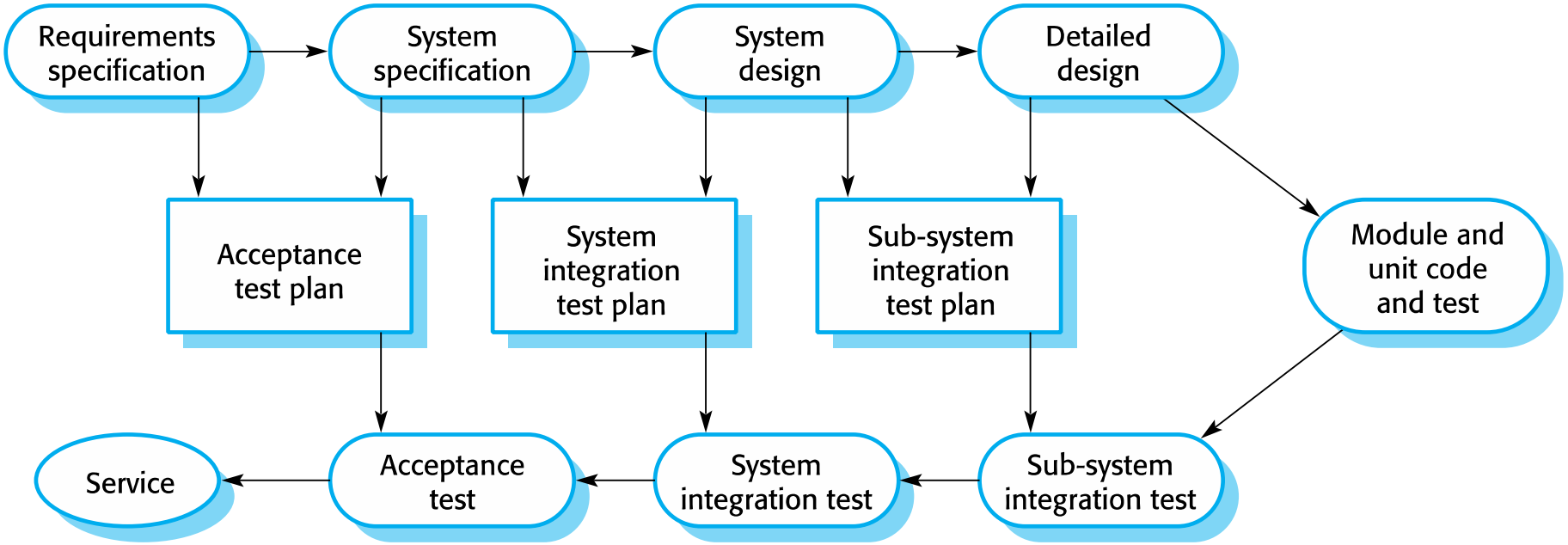
3. Software Validation

- Verification and validation (V&V) intends to show that a system conforms to its specification and meets the requirements of the system customer.
  - Involves checking, review processes and system testing.
  - Testing is the most commonly used V & V activity.



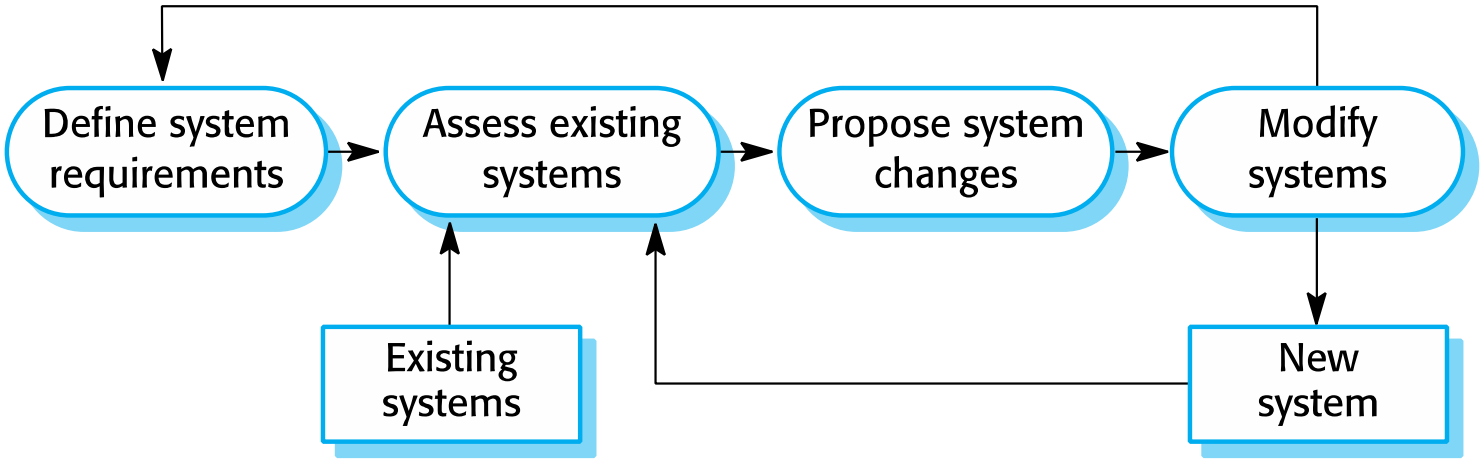
- Stages of Testing
  - **Component testing**
    - Unit testing
    - Individual components are tested independently.
    - Components may be functions or objects or coherent groupings of these entities.
  - **System testing**
    - Testing of the system as a whole.
    - Testing of emergent properties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

V-Model of Software Testing



4. Software Evolution

- Software that supports the business must also evolve and change, as requirements change through changing business circumstances.
  - Software is inherently flexible and can change.
  - In maintenance phase



Coping with Change

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** such as re-analyzing requirements or implementing new functionality

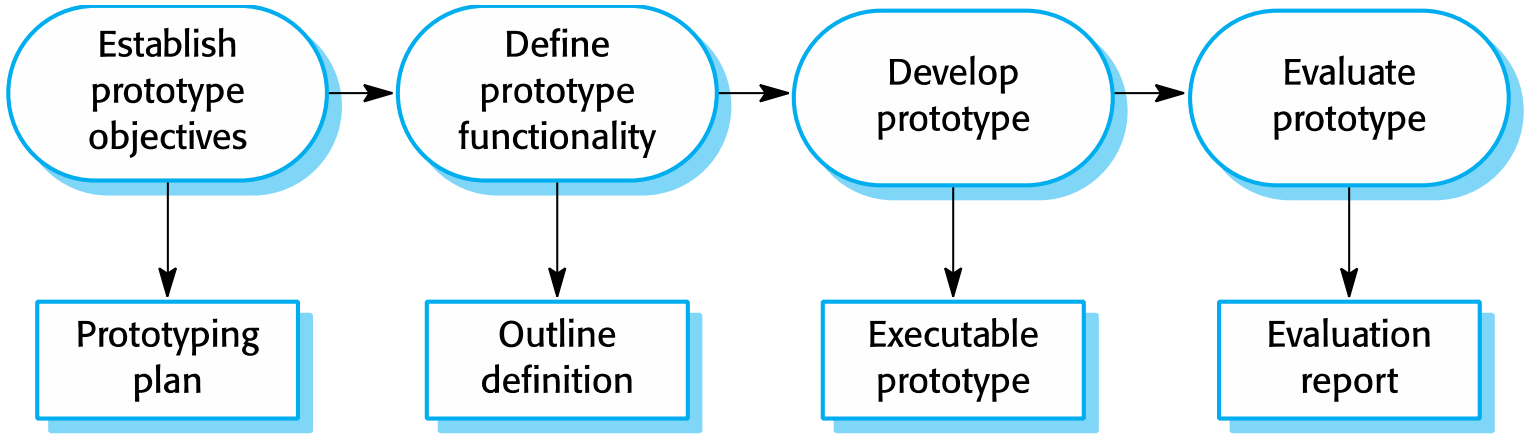
- Techniques to reduce the rework cost :
  - Change anticipation
  - Change tolerance
  - System prototyping
  - Incremental delivery

Techniques to Reduce the Costs of Rework

- Change anticipation
  - 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
  - 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 be altered to incorporate the change.
- System prototyping
  - A version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions.
    - This approach supports change anticipation.
- Incremental delivery
  - System increments are delivered to the customer for comment and experimentation.
    - This supports both change avoidance and change tolerance.

System Prototyping

- A version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions.
  - A prototype is **an initial version of a system** used to demonstrate concepts and try out design options.
  - In requirements engineering, to help requirements elicitation and validation
  - In design, to explore options and develop a UI design
  - In the testing, to run back-to-back tests.



Prototype Development

- May be based on rapid prototyping languages or tools and involve leaving out functionality.
  - Prototype should focus on areas of the product that are not well-understood.
  - Error checking and recovery may not be included in the prototype.
  - Focus on functional rather than non-functional requirements.

- Two types of Prototyping
  - **Throw-away prototyping**
  - **Incremental delivery**

Two Types of Prototyping

- Throw-away prototyping
  - 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 organizational quality standards.

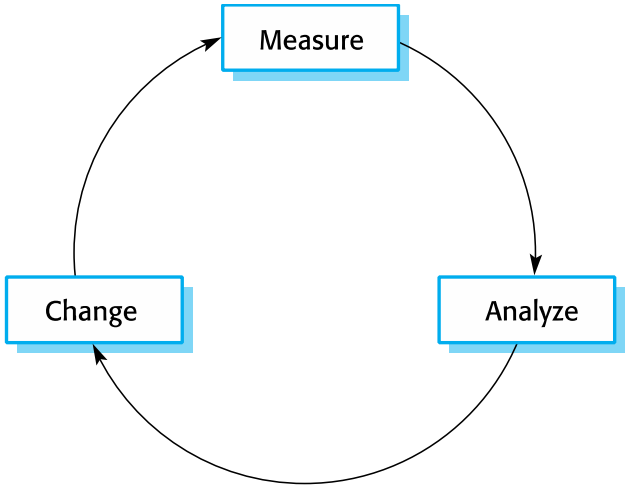
- Incremental delivery
  - The development and delivery is broken down into increments with each increment delivering part of the required functionality.
    - User requirements are prioritized, and the highest priority requirements are included in early increments.
    - Once the development of an increment is started, the requirements are frozen, though requirements for later increments can continue to evolve.
  - Called evolutionary prototyping

Process Improvement

Process Improvement

- Process improvement
  - Understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time.

- Many software companies have turned to software process improvement as a way of enhancing the quality of their software and reducing costs.
  - **The level of process maturity**, such as CMMi, reflects the extent to which good technical and management practice has been adopted in organizational software development processes.
  - Activities of process improvement
    - Measurement
    - Analysis
    - Change



Process Improvement Activities

- **Process measurement**
  - Measure one or more attributes of the software process or product.
  - These measurements forms a baseline that helps you decide if process improvements have been effective.
- **Process analysis**
  - The current process is assessed, and process weaknesses and bottlenecks are identified.
  - Process models (process maps) that describe the process may be developed.
- **Process change**
  - Process changes are proposed to address some of the identified process weaknesses.
  - These are introduced and the cycle resumes to collect data about the effectiveness of the changes.

Process Measurement

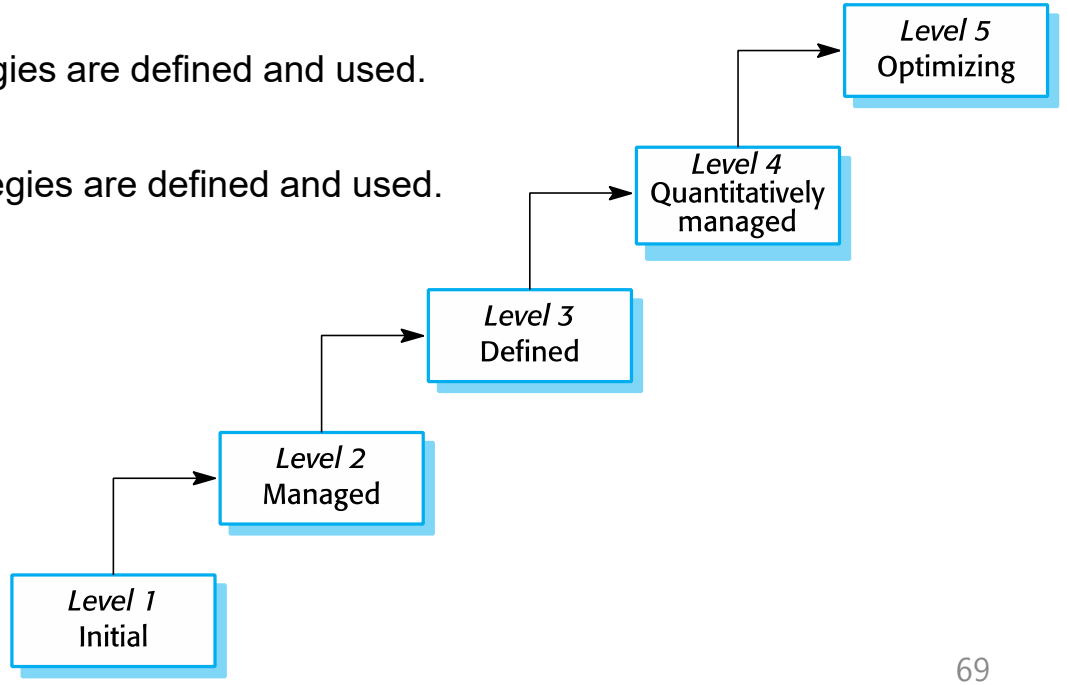
- Wherever possible, **quantitative process data** should be collected.
  - However, organizations often do not have clearly defined process standards.
    - It is very difficult as we don't know what to measure.
  - A process should be defined **before** any measurement is possible.

- The organizational objectives should drive the process improvements.

- **Examples of process metrics**
  - Time taken for process activities to be completed
    - E.g., Calendar time, 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

The SEI CMMi

- **CMMi** (Capability Maturity Model Integrated)
  1. Initial
     - Essentially uncontrolled
  2. Repeatable
     - Product management procedures are defined and used.
  3. Defined
     - Process management procedures and strategies are defined and used.
  4. Managed
     - Quality management strategies are defined and used.
  5. Optimizing
     - Process improvement strategies are defined and used.



Key Points

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- 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.

Key Points

- Processes should include activities such as prototyping and incremental delivery to cope with change.
- `Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- 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 SEI process maturity framework identifies maturity levels that essentially correspond to the use of good software engineering practice.

Chapter 3. Agile Software Development

Topics Covered

- Agile methods
- Agile development techniques
- Agile project management
- Scaling agile methods

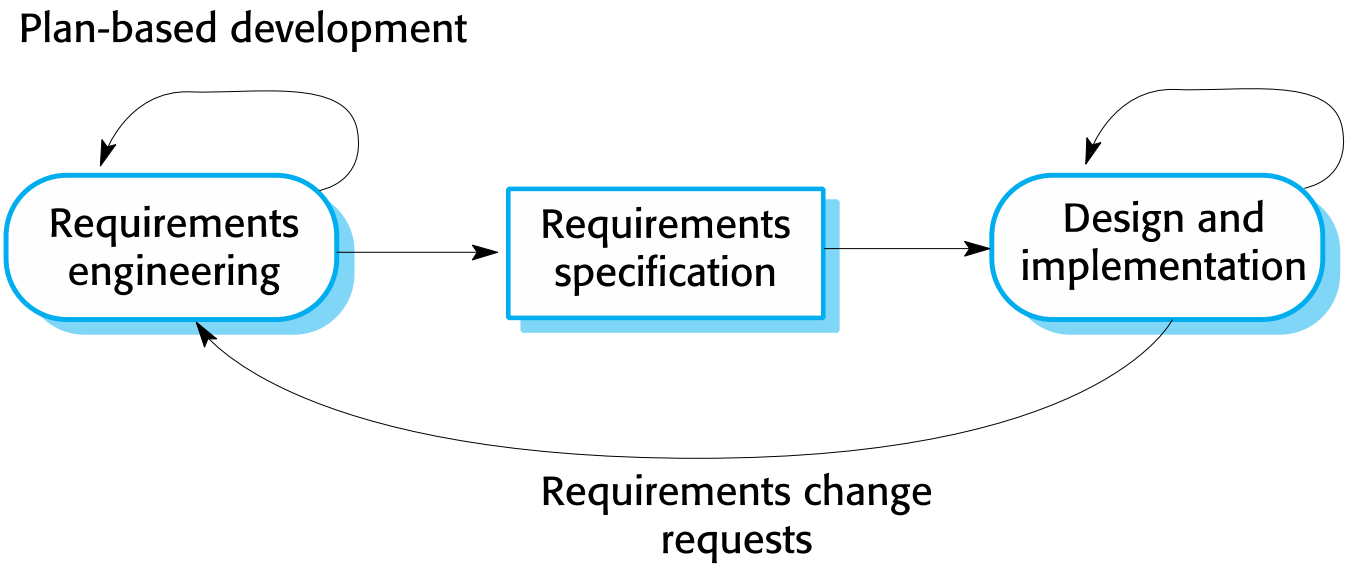
Rapid Software Development

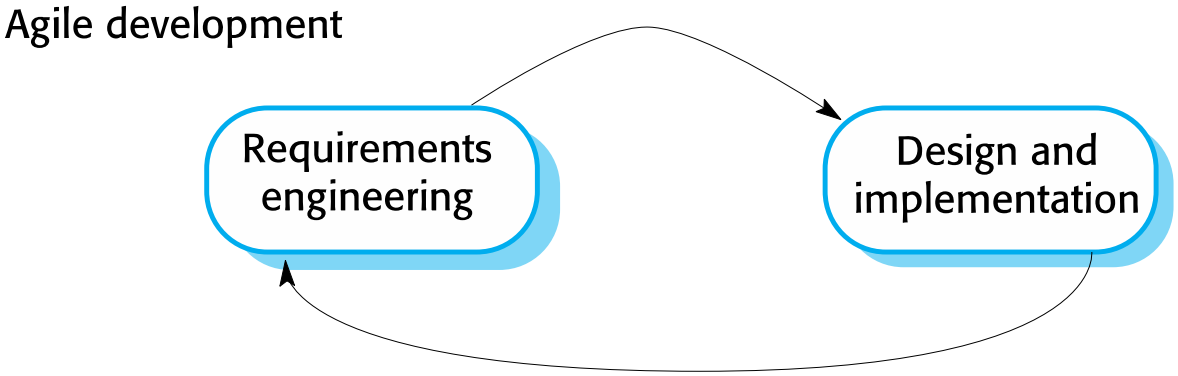
- Rapid development and delivery is now often the most important requirement for software systems.
  - Software must evolve quickly to reflect changing business needs.
  - Plan-driven development does not meet these business needs.

- **Agile development methods** emerged in the late 1990s.
  - To radically reduce the delivery time for working software systems.

- Features of Agile development
  - Program specification, design and implementation are inter-leaved.
  - The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation.
  - Frequent delivery of new versions for evaluation
  - Extensive tool support (e.g., automated testing tools)
  - Minimal documentation to focus on working code

Plan-Driven vs. Agile Development





Plan-Driven vs. Agile Development

- **Plan-driven development**
  - Based on separate development stages with the outputs to be produced at each of these stages planned in advance.
  - Not necessarily waterfall model, but plan-driven incremental development is possible.
    - Iteration occurs within activities.

- **Agile development**
  - Specification, design, implementation and testing are interleaved.
  - The outputs from the development process are decided through a process of negotiation during the software development process.

Agile Methods

Agile Methods

- Motivation
  - Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s
  - To reduce overheads in the software process and to be able to respond quickly to changing requirements without excessive rework

- Agile methods
  - Focus on the code rather than the design
  - Based on an iterative approach to software development
  - Intend to deliver working software quickly and evolve this quickly to meet changing requirements

- **Agile Manifesto**
  - **Individuals and interactions over processes and tools**
  - **Working software over comprehensive documentation**
  - **Customer collaboration over contract negotiation**
  - **Responding to change over following a plan**

Principles of Agile Methods

Principle	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.

Applicability of Agile Method

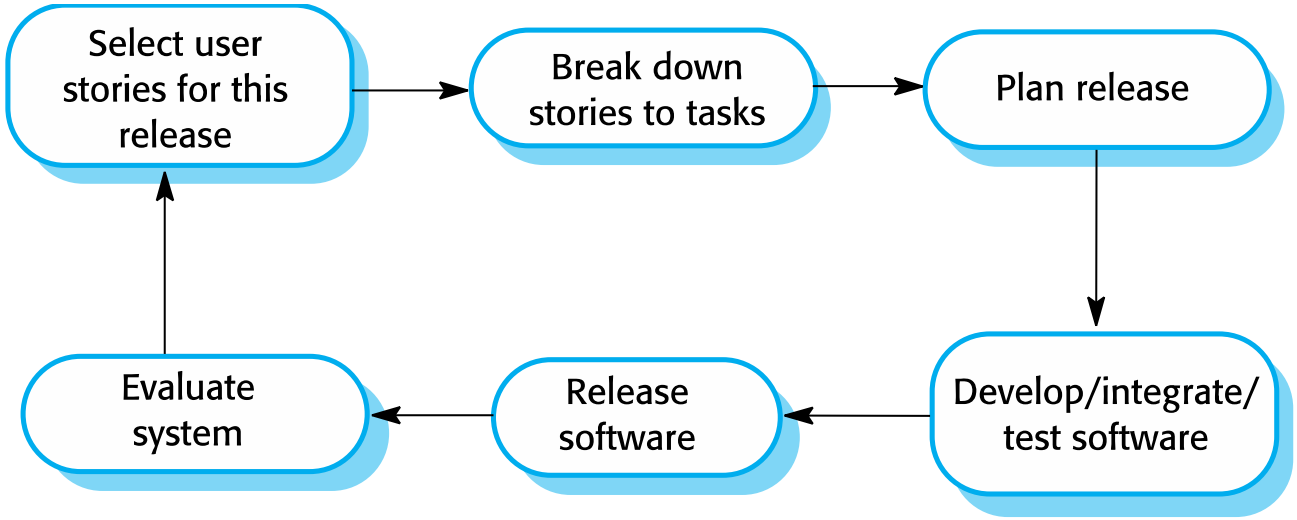
- Development of **small** or medium-sized product for sale
  - Almost all software products and apps are now developed using an agile approach.

- Custom system development within an organization where,
  - There is a clear commitment from the customer to become involved in the development process, and
  - There are few external rules and regulations that affect the software

Agile Development Techniques

Extreme Programming

- **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.



XP Practices

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.
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.

XP Principles

- **The XP principles**
  - Incremental development is supported through small, frequent system releases.
  - Customer involvement means full-time customer engagement with the team.
  - Collective ownership through pair programming.
  - Change supported through regular system releases.
  - Maintaining simplicity through constant refactoring of code.

XP in Practice

- The XP method is not widely used, since
  - Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.

- However, XP practices are widely used in other development methods.
  - **User stories for specification**
  - **Refactoring**
  - **Test-first development**
  - **Pair programming**

User Stories for Requirements

- User requirements are expressed as user stories or scenarios.
  - Written on cards and the development team break them down into implementation tasks. Tasks are the basis of schedule and cost estimates.
  - Customer or user is part of the XP team and is responsible for making decisions on requirements.
    - The customer chooses the stories for inclusion in the next release.

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either 'current medication', 'new medication' or 'formulary'.

If you select 'current medication', you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, 'new medication', the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose 'formulary', you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click 'OK' or 'Change'. If you click 'OK', your prescription will be recorded on the audit database. If you click 'Change', you reenter the 'Prescribing medication' process.

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.
Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.
Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low.
If within the range, enable the 'Confirm' button.

Refactoring

- The XP refactoring:
  - Constant code improvement to make changes easier when they must be implemented.
    - In contrast to conventional SE, saying “Do design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.”
  - Programming teams 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.
    - Examples:
      - Re-organization of a class hierarchy to remove duplicate code
      - Tidying up and renaming attributes and methods to make easier to understand
      - Replacement of inline code with calls to methods that have been included in a program library
  - 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.

Test-First Development

- **TFD (Test-First Development)**
  - Testing is central to XP.
  - "The program should be tested after every change has been made."

- Difficulties in TFD
  - Programmers prefer programming to testing and sometimes they take short cuts when writing tests.
  - Some tests can be very difficult to write incrementally.
  - It is difficult to judge the completeness of a set of (a lot of) tests.

- Features of the XP testing
  - Test-driven 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.

Test-Driven Development

- **TDD (Test-Driven 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.
- **Automated test execution environment** is mandatory.
  - 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 customer is a part of the team in XP
  - Help develop acceptance tests for the stories that are to be implemented in the next release of the system.
  - 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.

Test Case Description for 'Dose Checking'

Test 4: Dose checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Test Automation

- **Test automation**
  - "Tests are written as executable components before the task is implemented."
  - Automated test framework is required.
  - Each testing component should
    - Be stand-alone (independent),
    - Simulate the submission of input to be tested, and
    - Check that the result meets the output specification.

- **Automated test framework**
  - A system that makes it easy to write executable tests and submit a set of tests for execution
  - Example
    - A series of *x*Unit
  - As testing is automated, a set of tests is always ready to do quickly.
    - 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.

Pair Programming

- **Pair programming**
  - "Involves programmers working in pairs, developing code together."
  - Programmers sit together at the same computer 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.
  - Helps develop common ownership of code and spreads knowledge across the team.
  - Serves as an informal review process as each line of code is looked at by more than 1 person.
  - Encourages refactoring as the whole team can benefit from improving the system code.
- Pair programming is not necessarily inefficient.
  - Some evidence suggests that a pair working together is more efficient than 2 programmers working separately.

Agile Project Management

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.
  - Should be adapted to incremental development and the practices used in agile methods.
  - **Scrum**

Scrum

- An agile method that focuses on managing iterative development rather than specific agile practices.

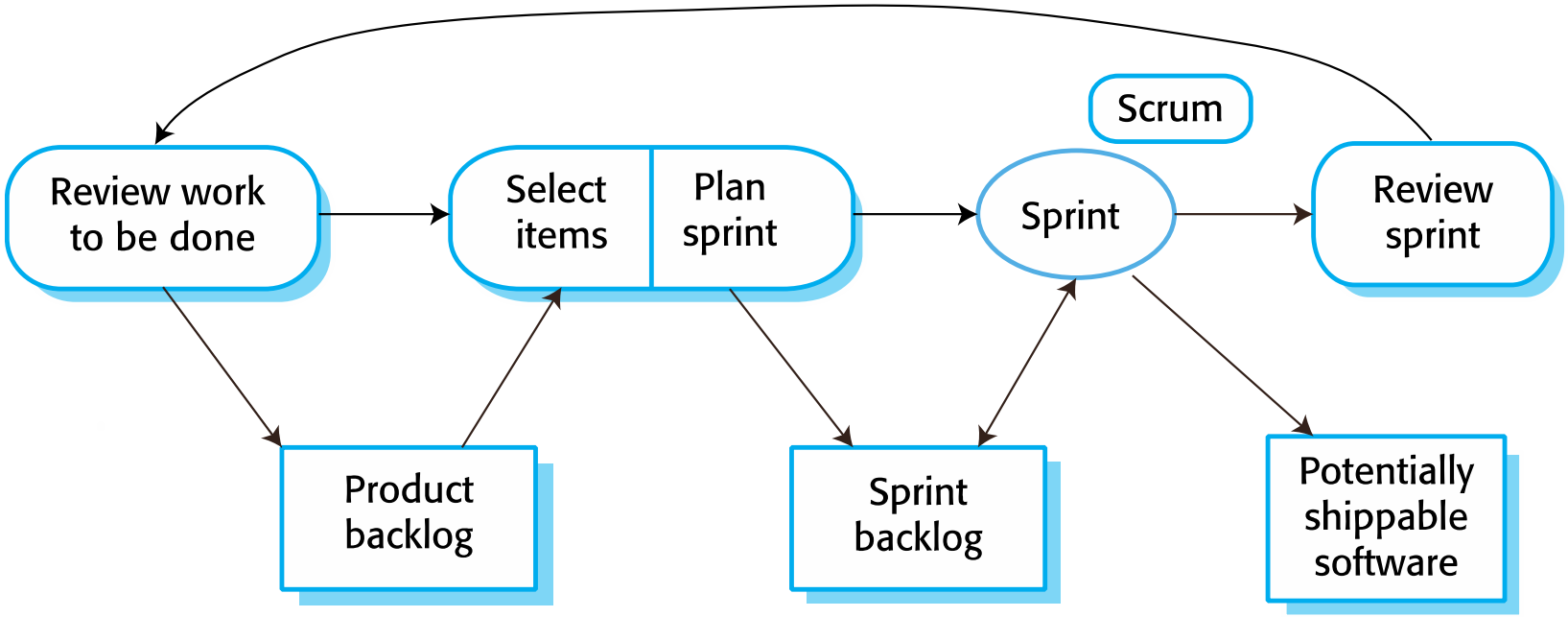
- **3 phases** in Scrum
  - Initial phase
    - An outline planning phase, where you establish the general objectives for the project and design the software architecture.
  - A series of **sprint** cycles
    - Each cycle develops an increment of the system. (2~4 weeks for each sprint)
  - Project closure phase
    - Wraps up the project, completes required documentation, and assesses the lessons learned from the project.

- The whole team members attend **short daily meetings called Scrum**.
  - All team members share information, describe their progress since the last meeting, problems that have arisen, and what is planned for the following day.
  - Everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

Scrum Terminology

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
Scrum Master	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

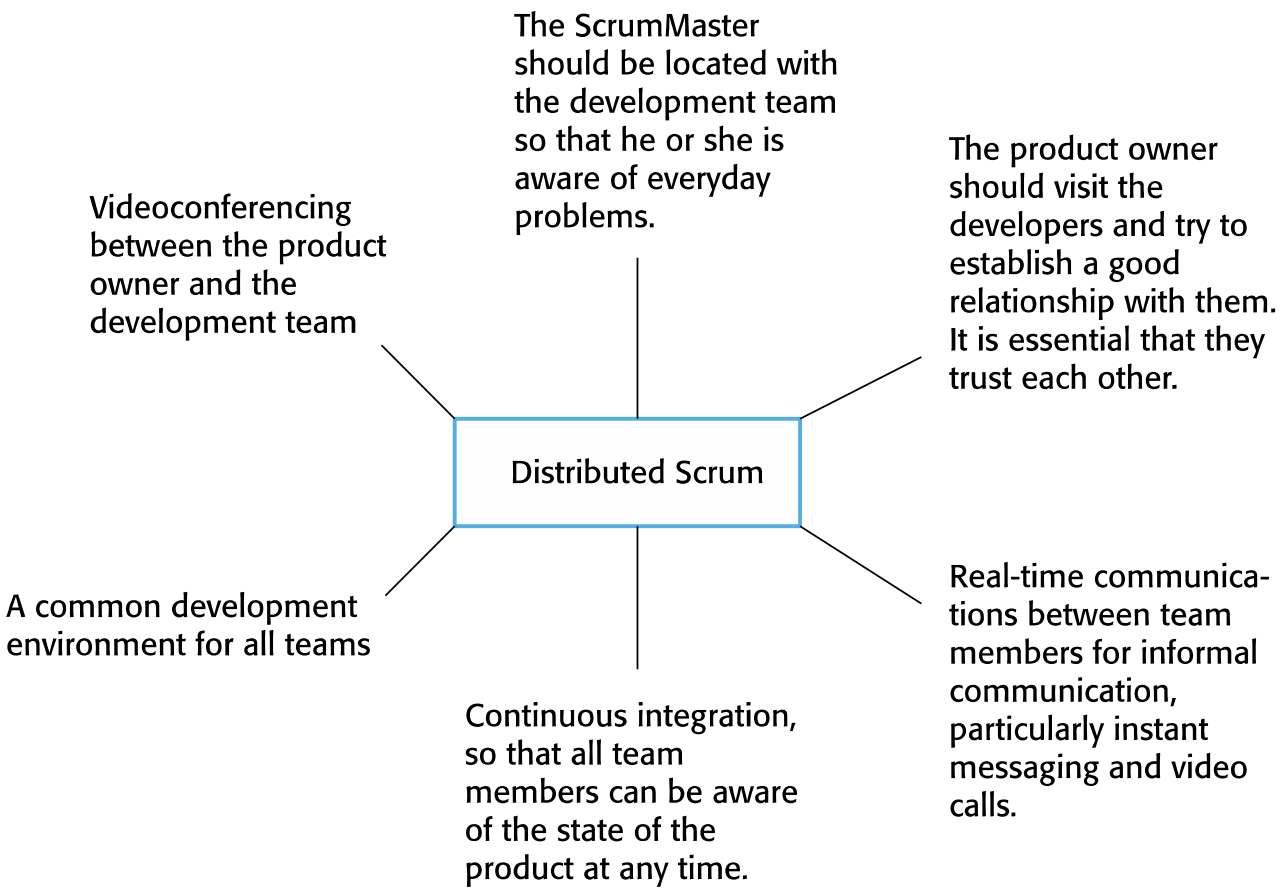
The Scrum Sprint Cycles



The Scrum Sprint Cycles

- Sprints are fixed length, normally **2~4 weeks**.
- 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 project teams who work with the customer to select the features and functionality from the product backlog 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.

Distributed Scrum



Scaling Agile Methods

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.

- **Scaling up agile methods** involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.
  - **Scaling up** : Using agile methods for developing large software systems that cannot be developed by a small team.
  - **Scaling out** : How agile methods can be introduced across a large organization with many years of software development experience.

- When scaling agile methods, it is important to maintain agile fundamentals:
  - Flexible planning
  - Frequent system releases
  - Continuous integration
  - Test-driven development
  - Good team communications

Practical Problems with Agile Methods

- Contractual issue
  - The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
- Maintenance issue
  - Agile methods are most appropriate for new software development rather than software maintenance.
  - But, the majority of software costs in large companies come from maintaining their existing software systems.
- System Issue
  - Agile methods are designed for small co-located teams.
  - But, much software development now involves worldwide distributed teams.
- Organization Issue
  - Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.

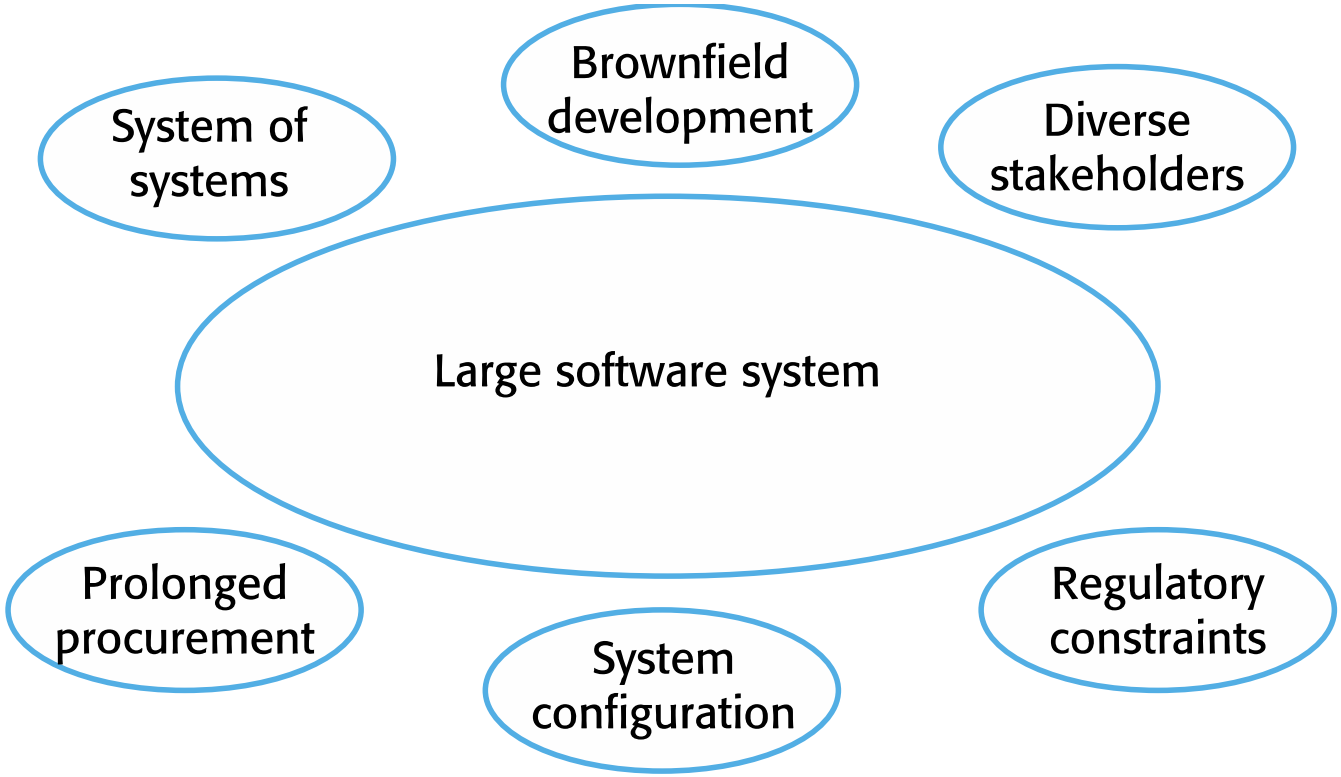
Large Systems Development

- Large systems are usually
  - Collections of separated communicating systems, where separate teams developed each system and are working in different places and time zones.
  - '**Brownfield systems**', including and interacting with a number of existing systems.
  - Concerned with system configuration rather than original code development.
  - Often constrained by external rules and regulations limiting the way that they can be developed.

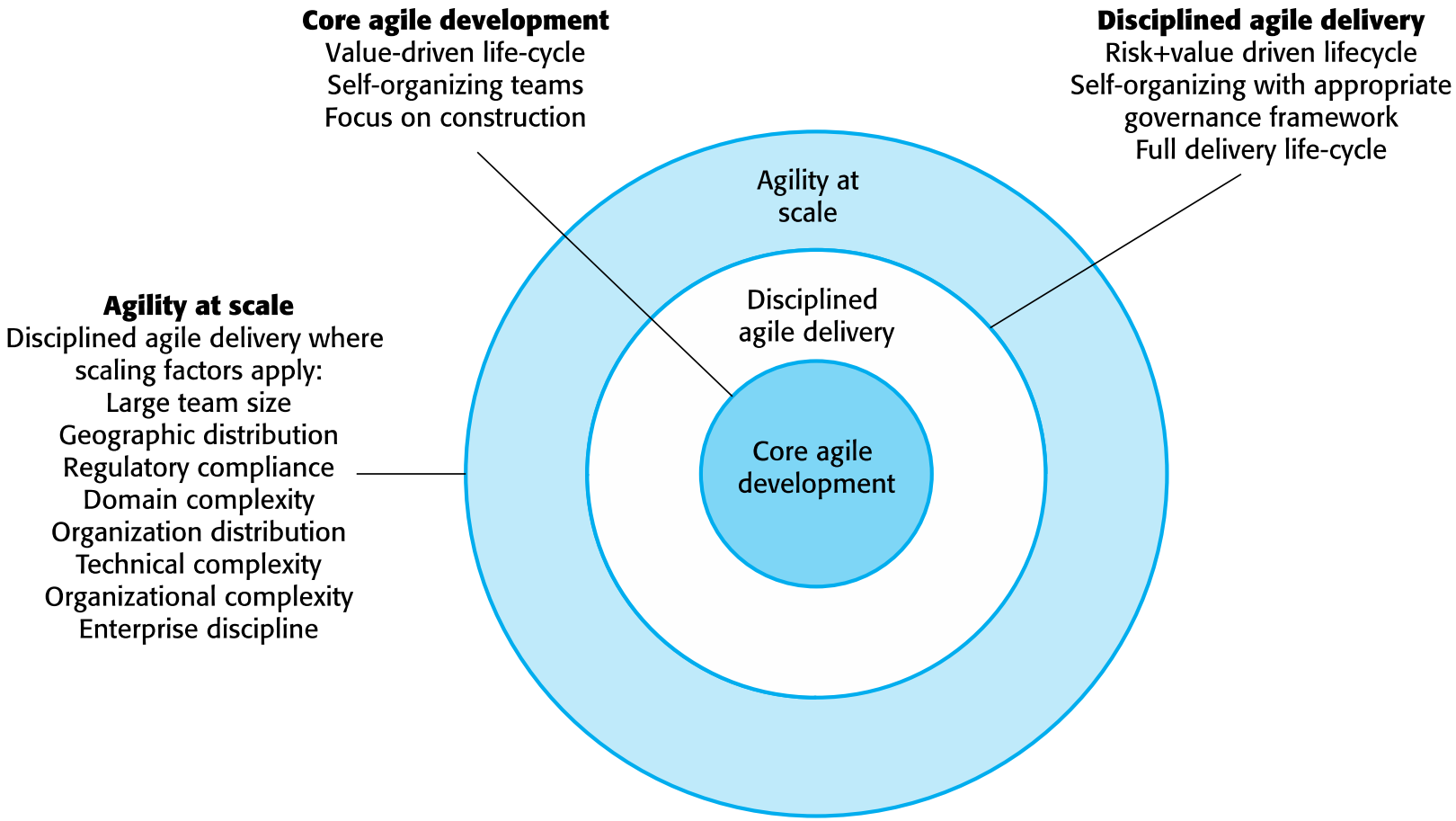
- Large systems usually have a diverse set of stakeholders.
  - It is practically impossible to involve all of these different stakeholders in the development process.
  - Many of the system requirements are concerned with their interaction and so don't really lend themselves to flexibility and incremental development.

- Large systems have a long procurement and development time.
  - It is difficult to maintain coherent teams who know about the system over that period.

Factors in Large Systems



IBM's Agility at Scale Model



Scaling-Up to Large Systems

- At developing large systems,
  - A completely incremental approach to requirements engineering is impossible.
  - There cannot be a single product owner or customer representative.
  - Not possible to focus only on the code of the system.

- Suggestion for scaling-up to large systems
  - Cross-team communication mechanisms should be designed and used.
  - It is essential to maintain frequent system builds and regular releases.
    - Although continuous integration is practically impossible.

  - Example : multi-team Scrum

Multi-Team Scrum

- Role replication
  - Each team has a Product Owner for their work component and ScrumMaster.
- Product architects
  - Each team chooses a product architect and these architects collaborate to design and evolve the overall system architecture.
- Release alignment
  - The dates of product releases from each team are aligned so that a demonstrable and complete system is produced.
- Scrum of Scrums
  - There is a daily Scrum of Scrums where representatives from each team meet to discuss progress and plan work to be done.

Scaling-Out across Organizations

- Large organizations often have quality procedures and standards that all projects are expected to follow
  - 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.
  - Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.

Key Points

- Agile methods are incremental development methods that focus on rapid software development, frequent releases of the software, reducing process overheads by minimizing documentation and producing high-quality code.

- Agile development practices include
  - User stories for system specification
  - Frequent releases of the software
  - Continuous software improvement
  - Test-first development
  - Customer participation in the development team.

Key Points

- Scrum 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.
- Many practical development methods are a mixture of plan-based and agile development.
- Scaling agile methods for large systems is difficult.
  - Large systems need up-front design and some documentation and organizational practice may conflict with the informality of agile approaches.

Chapter 4. Requirements Engineering

Topics covered

- Functional and non-functional requirements
- Requirements engineering processes
- Requirements elicitation
- Requirements specification
- Requirements validation
- Requirements change

Requirements Engineering

- The **process** of establishing the services that a customer requires from a system and the constraints under which it operates and is developed.
  - System requirements : descriptions of the system services and constraints that are generated during the requirements engineering process.

- Requirements
  - Range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
  - May be the basis for a bid for a contract - must be open to interpretation
  - May be the basis for the contract itself - must be defined in detail

Types of Requirement

- **User requirements**
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints
  - Written for customers.

- **System requirements**
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints
  - Defines what should be implemented so may be part of a contract between client and contractor.
  - Specified for developers.

User and System Requirements

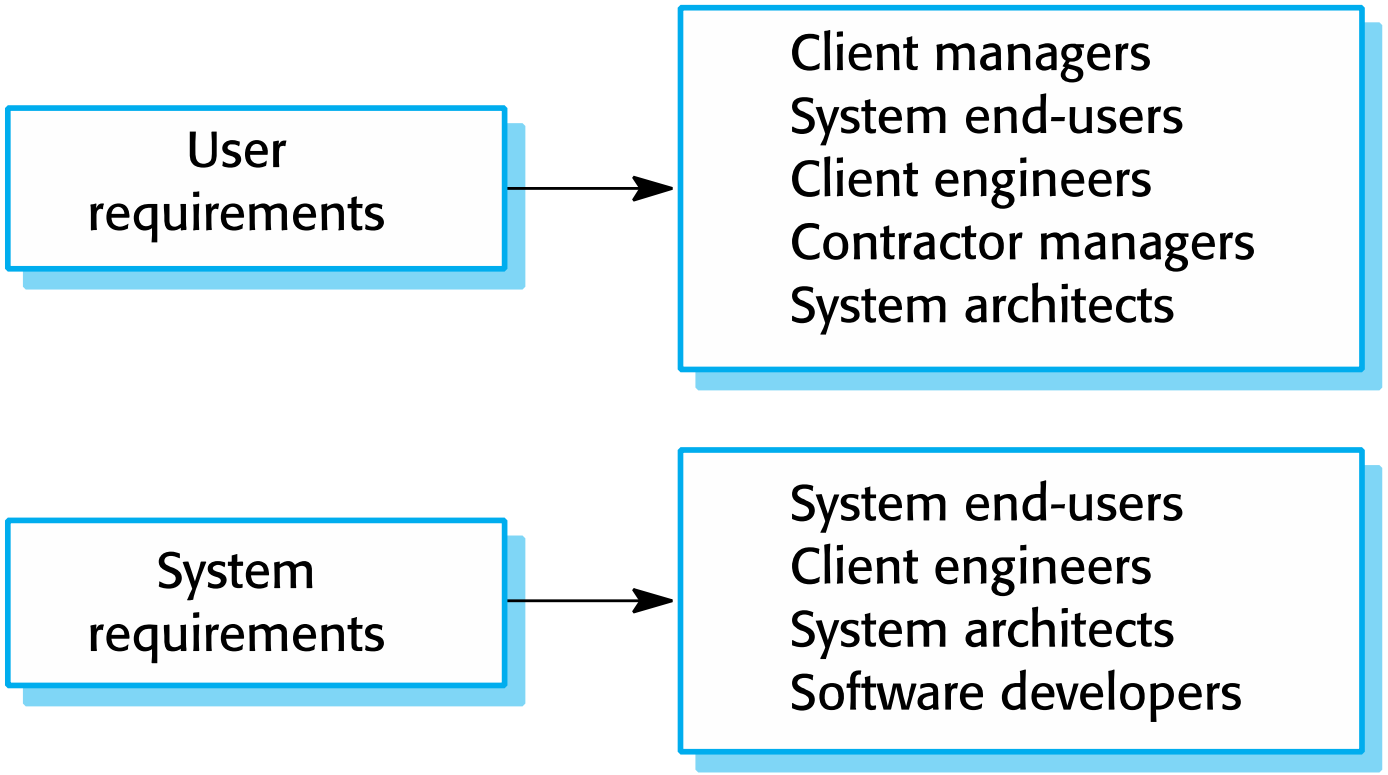
User requirements definition

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

System requirements specification

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

Readers of Requirements Specifications



System Stakeholders

- **Any person or organization** who is affected by the system in some way and so who has a legitimate interest

- Stakeholder types
  - End users
  - System managers
  - System owners
  - External stakeholders

Stakeholders in the Mentcare System

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

Agile Methods and Requirements

- Many agile methods argue that producing detailed system requirements is a waste of time as requirements change so quickly.
  - The requirements document is therefore always out of date.

- Agile methods usually use incremental requirements engineering and may express requirements as ‘user stories’ (discussed in Chapter 3).

- This is practical for business systems, but problematic for systems that require pre-delivery analysis (e.g., critical systems) or systems developed by several teams.

Functional and Non-Functional Requirements

Functional and Non-Functional Requirements

- **Functional requirements**
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.

- **Non-functional requirements**
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.

- **Domain requirements**
  - Constraints on the system from the domain of operation

Functional Requirements

- Describing functionality or system services depends on the type of software, expected users and the type of system where the software is used.
  - **Functional user requirements** may be high-level statements of what the system should do.
  - **Functional system requirements** should describe the system services in detail.

- Example : functional requirements for Mentcare System
  - “A user shall be able to search the appointments lists for all clinics.”
  - “The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.”
  - “Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.”

Requirements Imprecision

- Problems arise when functional requirements are not precisely stated.
  - Ambiguous requirements may be interpreted in different ways by developers and users.

- Consider the term '*search*' in requirement 1
  - User intention : search for a patient name across all appointments in all clinics;
  - Developer interpretation : search for a patient name in an individual clinic. User chooses clinic then search.

Requirements Completeness and Consistency

- In principle, requirements should be both **complete** and **consistent**.
- Complete
  - They should include descriptions of all facilities required.
- Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

Non-Functional Requirements

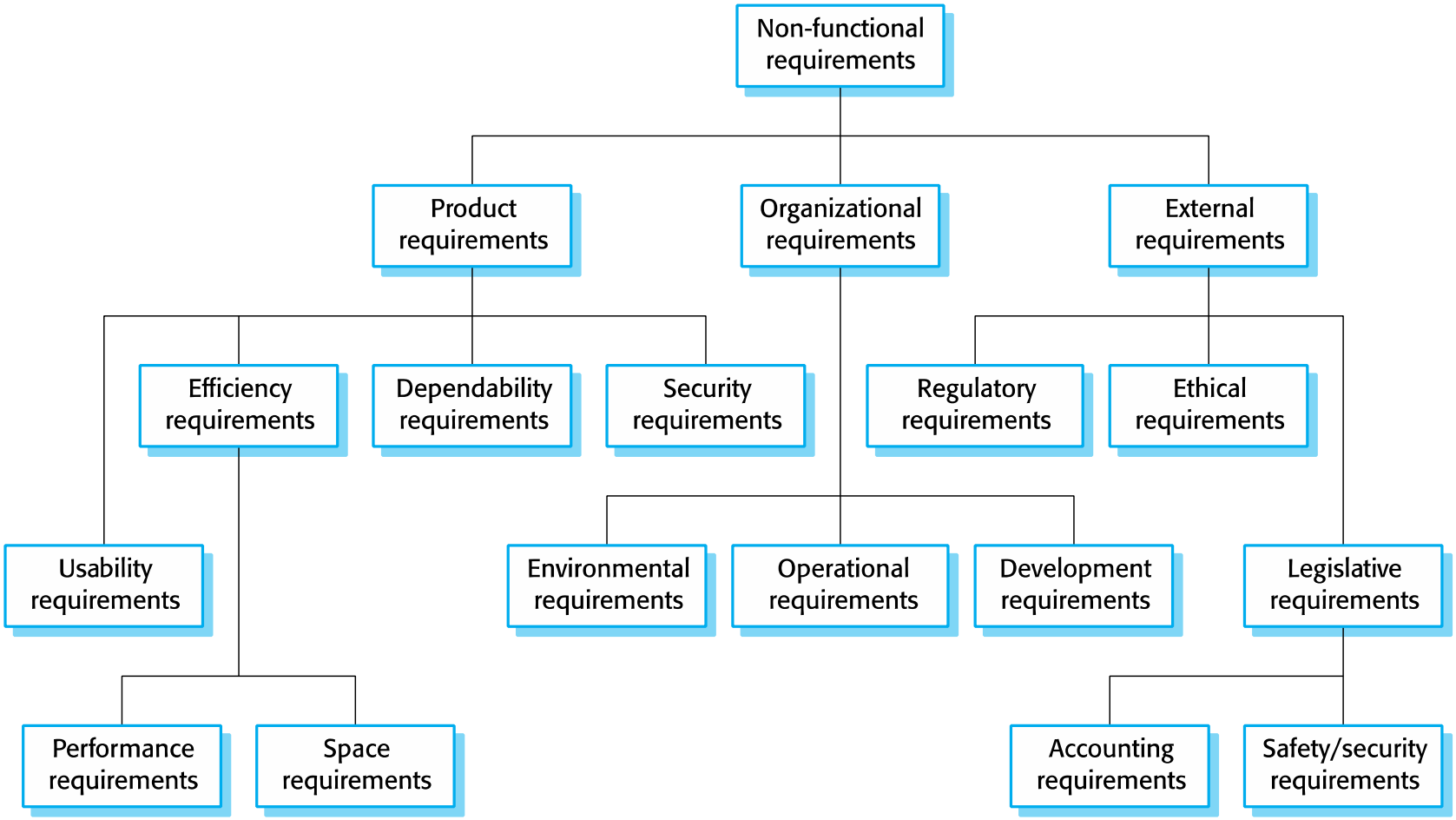
- Define system **properties** and **constraints**.
  - Reliability, response time and storage requirements, I/O device capability, system representations, etc.
  - Mandating a particular IDE, programming language or development method.
  - Standards to comply with

- Non-functional requirements may be more critical than functional requirements.
  - If these are not met, the system may be useless.

- Non-functional requirements may affect the overall architecture of a system rather than the individual components.
  - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.

- A single non-functional requirement often generate a number of related functional requirements that define system services that are required.
  - It may also generate requirements that restrict existing requirements.

Types of Non-Functional Requirements



Non-functional Classifications

- **Product requirements**
  - Requirements which specify that the delivered product must behave in a particular way
  - E.g., execution speed, reliability, etc.
- **Organizational requirements**
  - Requirements which are a consequence of organisational policies and procedures
  - E.g., process standards used, implementation requirements, etc.
- **External requirements**
  - Requirements which arise from factors which are external to the system and its development process
  - E.g., interoperability requirements, legislative requirements, etc.

Mentcare System : Non-Functional Requirements

Product requirement
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement
Users of the Mentcare system shall authenticate themselves using their health authority identity card.

External requirement
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Goals and Requirements

- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
  - Goals are helpful to developers as they convey the intentions of the system users.

- **Goal**
  - A general intention of the user such as ease of use.

- **Verifiable non-functional requirement**
  - A statement using some measure that can be objectively tested.

- Example : Usability
  - Goal : “The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.”
  - Verifiable non-functional requirement : “Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.”

Metrics for Specifying Non-Functional Requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

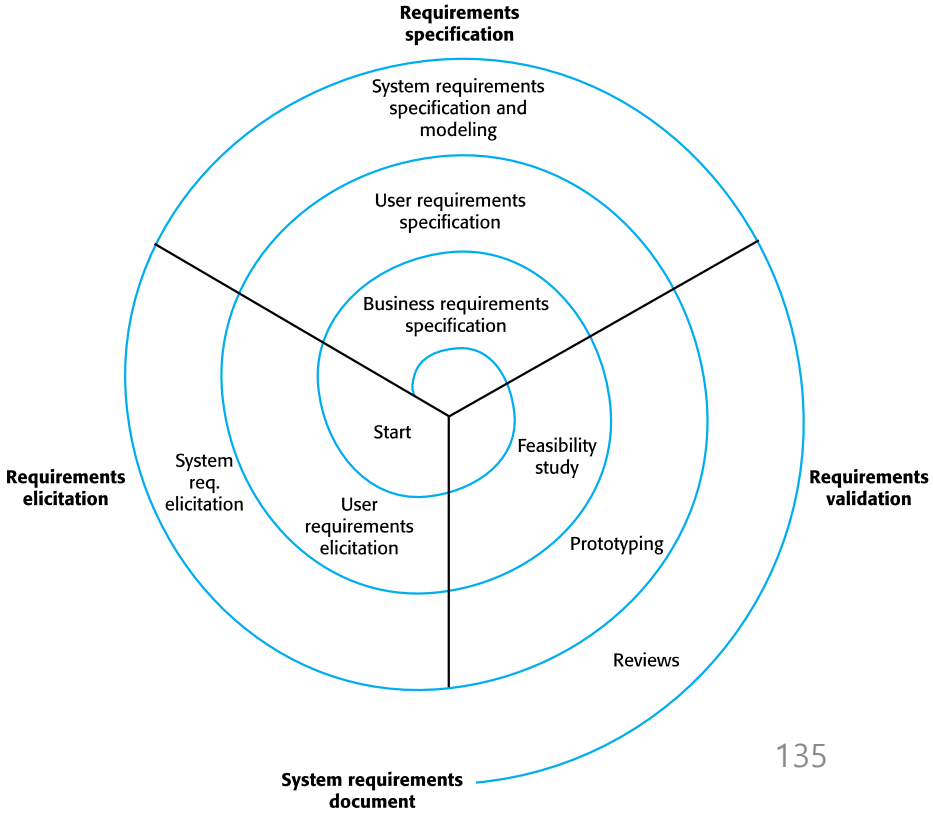
Requirements Engineering Processes

Requirements Engineering Processes

- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.

- However, there are **4 generic activities** common to all processes
  - **Requirements elicitation**
  - **Requirements analysis**
  - **Requirements validation**
  - **Requirements management**

- In practice, RE is an iterative activity in which these processes are interleaved.



Requirements Elicitation

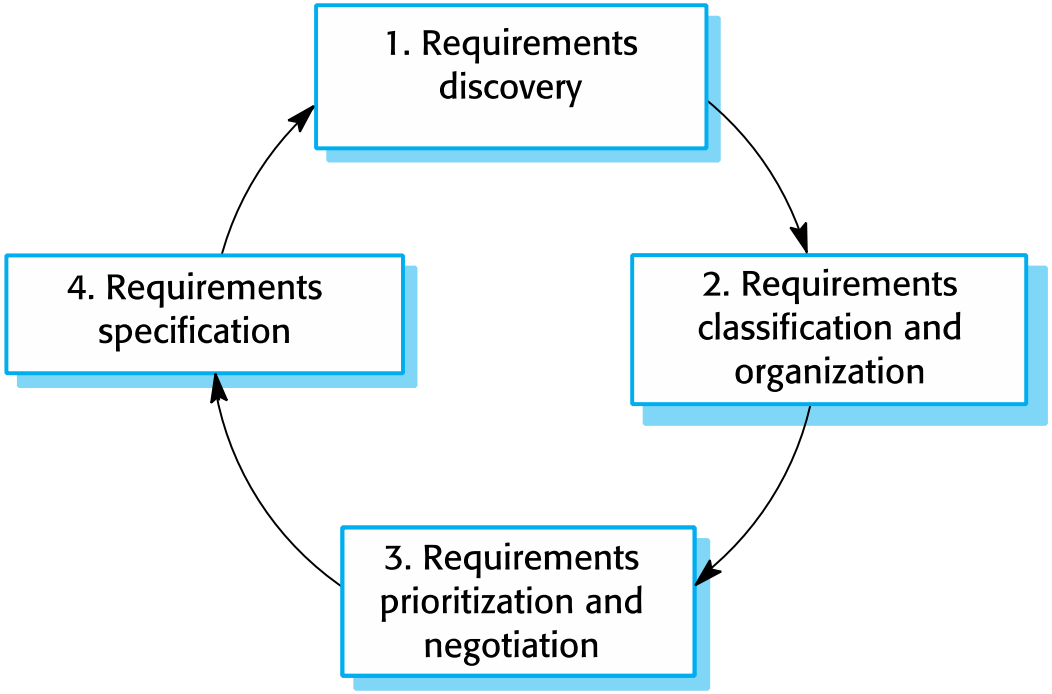
Requirements Elicitation and Analysis

- Called requirements elicitation or requirements discovery.
  - Involves technical staff working with customers to find out about the application domain, the services that the system should provide, and the system's operational constraints.
  - May involve various stakeholders such as end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc.

- Difficulties in requirements elicitation
  - Stakeholders don't know what they really want.
  - Stakeholders express requirements in their own terms.
  - Different stakeholders may have conflicting requirements.
  - Organizational and political factors may influence the system requirements.
  - The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

Requirements Elicitation Process

- RE elicitation process includes
  1. Requirements discovery
  2. Requirements classification and organization
  3. Requirements prioritization and negotiation
  4. Requirements specification



Process Activities

- **Requirements discovery**
  - Interacting with stakeholders to discover their requirements.
  - Domain requirements are also discovered at this stage.
- **Requirements classification and organization**
  - Groups related requirements and organises them into coherent clusters.
- **Prioritization and negotiation**
  - Prioritising requirements and resolving requirements conflicts.
- **Requirements specification**
  - Requirements are documented and input into the next round of the spiral.

Requirements Discovery

- The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.
  - Systems normally have a range of stakeholders.
  - Interaction is with system stakeholders from managers to external regulators.

- Techniques for requirements discovery
  1. **Requirements workshop**
  2. **Brainstorming**
  3. **Storyboards (Use-Case scenario)**
  4. **Interviews**
  5. **Questionnaires**
  6. **Role playing**
  7. **Prototypes**
  8. **Review customer requirement specification**

1. Requirements Workshop

- **Gather all stakeholders together** for an intensive and focused period
  - Promotes participation by everyone
  - Create consensus on the scope, risks and key features of the software system
  - Results immediately available

- Produce artifacts such as:
  - Problem statement , Key features , Initial business object model
  - Use-case diagram , Prioritized risk list

- Provide a framework for applying other elicitation techniques such as
  - Brainstorming, use-case workshops, storyboarding, etc.

- Accelerate the elicitation process

2. Brainstorming

- **Rules for Brainstorming**
  - Clearly state the objective of the session
  - Generate as many ideas as possible
  - Let your imagination soar
  - Do not allow criticism or debate
  - Mutate and combine ideas

- **Generate as many ideas as possible**
  - Even the impractical, absurd ideas should not be neglected
  - Merge the various ideas to create new ideas

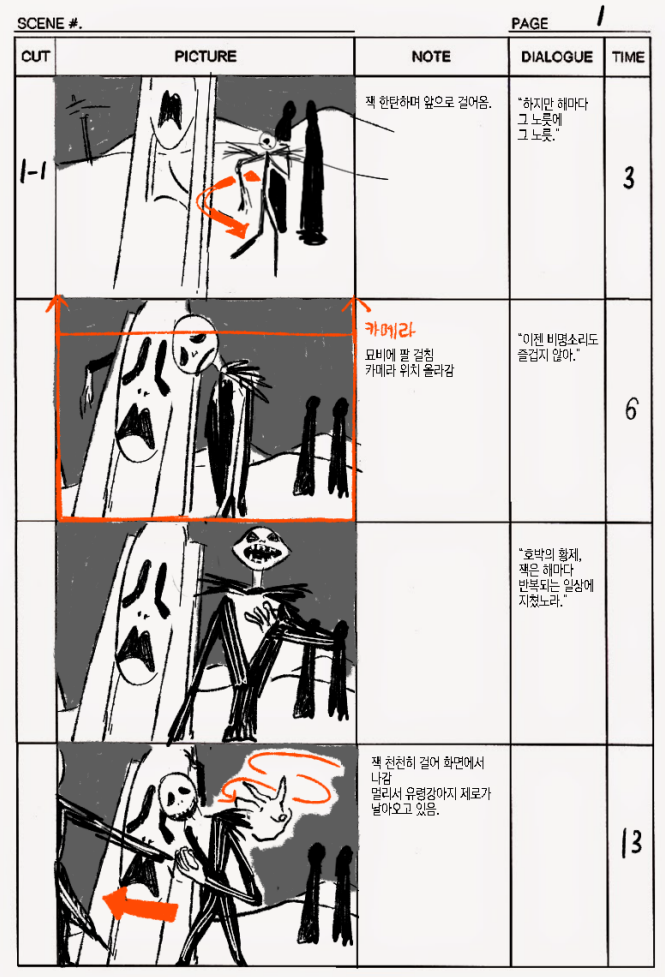
- **Express freely**
  - Do not explain or specify the ideas
  - Do not evaluate or argue about the ideas
  - Do not put names on the ideas
  - Encourage the unexpected and imaginative

- **Put up ideas openly**
  - Ideas should be put up on a whiteboard where all can see
  - Participants themselves may put up ideas on the board
  - Put tabs of Post-Its on the center table

3. Storyboards

- Visually tell and show:
  - Who/what the players are (actors)
  - What happens to them
  - When it happens

- Benefits
  - Help gather and refine customer requirements
  - Encourage creative and innovative solutions
  - Encourage team review
  - Prevent features that no one wants
  - Ensure that features are implemented in an accessible and intuitive way
  - Ease the interviewing process
  - Help to avoid blank-page syndrome



4. Interviews

- Provide **a simple and direct technique** to gain understanding of problems and solutions

- Types of interviews
  - **Open interview**
    - No pre-set agenda
    - Irrelevant data can be gathered
    - Needs time and training
  - **Closed interview**
    - Fairly open-questions agenda
    - Needs extended preparation
    - Prevents biases

- Interview tips
  - Avoid asking people to describe things they don't usually describe
    - Example: Describe how to tie your shoes
  - Avoid "Why…?" questions
  - Ask open-ended (context-free) questions
  - Listen, listen, listen!

Context-Free Questions

- High-level, abstract questions
  - Explore needs from stakeholder perspective
    - User's problems
    - Potential solutions
  - Unbiased with application or solutions knowledge
  - Typically posed early in a project

User questions	• *Who are the users?* • *What the key responsibilities of each user?* • *What is the user's background, capabilities, environment?*
Process questions	• *What is the problem?* • *How do you currently solve the problem?* • *How would you like to solve the problem?* • *Where else can the solution to this problem be found?*
Product questions	• *What environment will the product encounter?* • *What business problems could this product create?* • *What are your expectations for usability? reliability?*
Meta-questions	• *Do my questions seem relevant?* • *Are you the right person to answer these questions?* • *Are your answers requirements?* • *Is there anything else I should be asking you?*

5. Questionnaires

- Give access to a wide audience
  - Apply to broad markets where questions are well-defined
- Appear scientific because of statistical analysis
  - Powerful, but not a substitute for an interview
- Assumptions:
  - Relevant questions can be decided in advance
  - Questions phrased, so reader hears as intended

고 객 명		소 속		조사일자	20 년 월 일
■ 설문조사내용					
구 분	평 가 내 용		평 가		점 수
품질	1. 고객께서 알고 있는 회사명의 이미지는 어떠하십니까?		A : 좋 다 B : 보 통 C : 나쁘다		
	2. 고객께서는 회사명으로 전화를 하셨을 때 친절하게 응대를 하였습니까?		A : 좋 다 B : 보 통 C : 나쁘다		
	3. 고객께서는 회사명의 제품에 만족 하십니까?		A : 만 족 B : 보 통 C : 불만족		
	4. 당사의 제품에 이상 발생 시 고객께서 요구했을 때 신속한 대응이 있었습니까?		A : 그렇다 B : 보 통 C : 아니다		
	5. 품질관련해서 동종업계화의 수준에 있어 어느 정도의 수준이라고 생각하십니까?		A : 상 위 B : 중 위 C : 하 위		
	평 가 점 수	A : 20 B : 15 C : 10		소 계	
생산기술	1. 회사명을 과거에 방문하신 적이 있으면 어떤 느낌을 받았습니까?		A : 좋 다 B : 보 통 C : 나쁘다		
	2. 회사명의 생산기술에 있어 동종 업계와의 기술수준은 어느 정도라고 생각하십니까?		A : 상 위 B : 중 위 C : 하 위		
	3. 제품에 대한 상호간 정보교환은 잘 이루어지고 있습니까?		A : 그렇다 B : 보 통 C : 아니다		
	4. 신규 제품 개발 시 경쟁업체와 비교했을 때 개발일정 및 진행이 잘 되고 있습니까?		A : 그렇다 B : 보 통 C : 아니다		
	평 가 점 수	A : 25 B : 20 C : 15		소 계	
영업	1. 당사의 납입 준수율은 어느 정도라고 생각하십니까?		A : 80%이상 B : 50%이상 C : 50%미만		
	2. 당사의 납품수량은 정확하다고 생각하십니까?		A : 정 확 B : 보 통 C : 미 흡		
	3. 회사명의 서비스(친절함) 수준은 어느 정도라고 생각하십니까?		A : 좋 다 B : 보 통 C : 나쁘다		
	4. 포장 및 제품에 청결함과 더불어 사용하는데 불편함이 있었습니까?		A : 그렇다 B : 보 통 C : 아니다		
	평 가 점 수	A : 25 B : 20 C : 15		소 계	
고객의견				평 점	

6. Role Playing

- Perform requirements elicitation from the viewpoint of the roles
  - Learns and performs user's job
  - Performs a scripted walkthrough

- Advantages
  - Gain real insights into the problem domain
  - Understand problems that users may face

7. Prototypes

- Demonstrate some or all of the externally observable behaviors of a system through building prototypes quickly

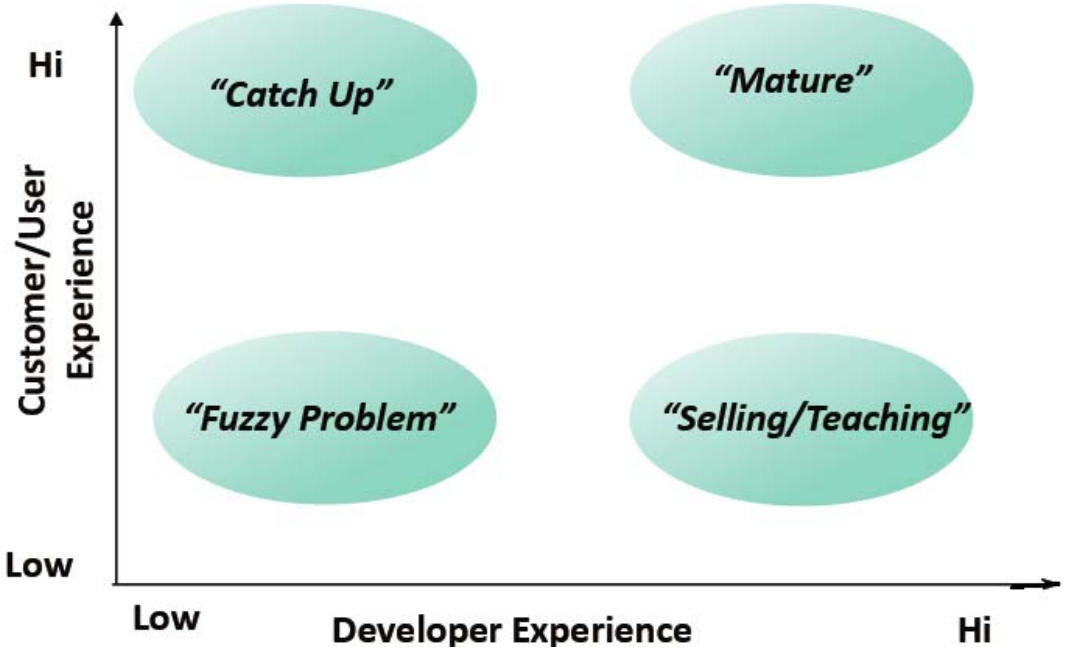
- Used to:
  - Demonstrate understanding of the problem domain
  - Gain feedback on proposed solution
  - Validate known requirements
  - Discover unknown requirements
  - Create simulations

  - Elicit and understand requirements
  - Prove and understand technology
  - Reduce risk
  - Enhance shared understanding
  - Improve
    - Cost and schedule estimates
    - Feature definition

8. Review Customer Requirement Specifications

- **Customer Requirements Review**
  - Recognize and label
    - Application behaviors
    - Behavioral attributes
    - Issues and assumptions
  - **Ask customers**

Which Techniques to Use?

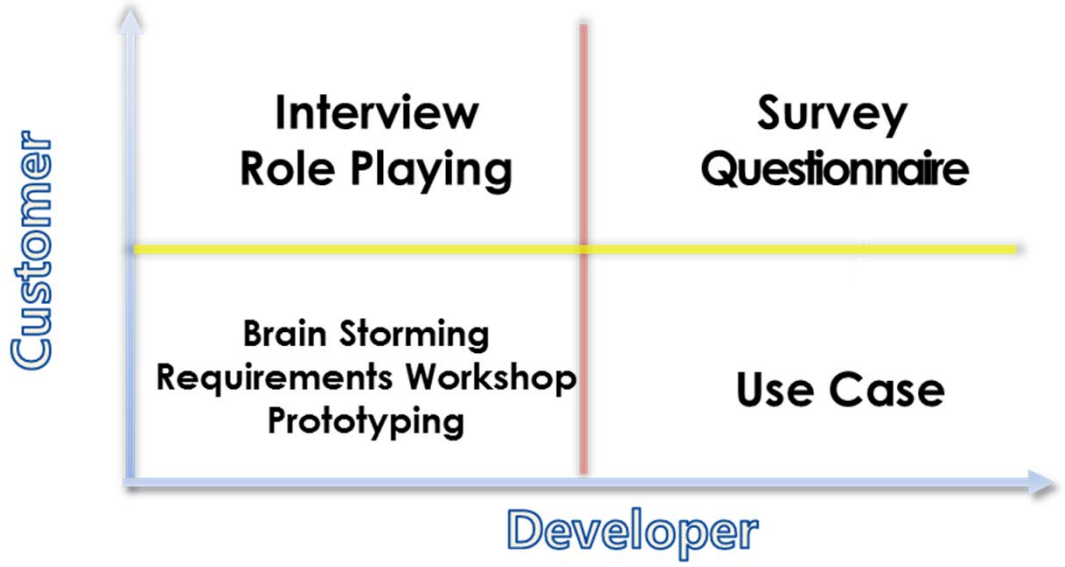


- **Catch Up**
  - Role Playing
  - Interview
  - Requirements Review
- **Fuzzy Problem**
  - Requirements Workshops
  - Brainstorming
  - Storyboards
- **Selling / Teaching**
  - Use Case
  - Business Modeling
- **Mature**
  - Questionnaires
  - Prototyping

Which Techniques to Use?

- **No single technique is sufficient** for realistic projects.

- Appropriate method should be chosen based on:
  - The size and complexity of requirements
  - Problem domain
  - The number of involved stakeholders



Requirements Specification

Requirements Specification

- The process of writing down the user and system requirements in a requirements document.
  - **User requirements** have to be understandable by end-users and customers who do not have a technical background.
  - **System requirements** are more detailed requirements and may include more technical information.

- The requirements may be part of a contract for the system development
  - It is therefore important that these are as complete as possible.

Ways of Writing a System Requirements Specification

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Requirements and Design

- In principle, requirements should state what the system should do and the design should describe how it does this.

- In practice, requirements and design are inseparable
  - A system architecture may be designed to structure the requirements.
  - The system may inter-operate with other systems that generate design requirements.
  - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
  - This may be the consequence of a regulatory requirement.

Natural Language Specification

- Requirements are written as natural language sentences supplemented by diagrams and tables.
  - Used for writing requirements because it is expressive, intuitive and universal.

- Difficulties in writing requirements in natural languages
  - **Lack of clarity**
    - Precision is difficult without making the document difficult to read.
  - **Requirements confusion**
    - Functional and non-functional requirements tend to be mixed-up.
  - **Requirements amalgamation**
    - Several different requirements may be expressed together.

- Guidelines
  - Invent a standard format and use it for all requirements.
  - Use language in a consistent way.
    - Use shall for mandatory requirements, should for desirable requirements.
  - Use text highlighting to identify key parts of the requirement.
  - Avoid the use of computer jargon.
  - Include an explanation (rationale) of why a requirement is necessary.

Insulin Pump : Natural Language Speicification

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. *(Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)*

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1.
(A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)

Structured Specifications

- An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
  - Works well for some types of requirements such as requirements for embedded control system.
  - Too rigid for writing business system requirements.

- Examples:
  - Form-based specification
  - Tabular specification
  - Use-Case

Insulin Pump : A Structured Specification

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requirements

Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

Post-condition r0 is replaced by r1 then r1 is replaced by r2.

Side effects None.

Tabular Specification

- Used to supplement natural language.
  - Particularly useful when you have to define a number of possible alternative courses of action.

- For example,
  - The insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

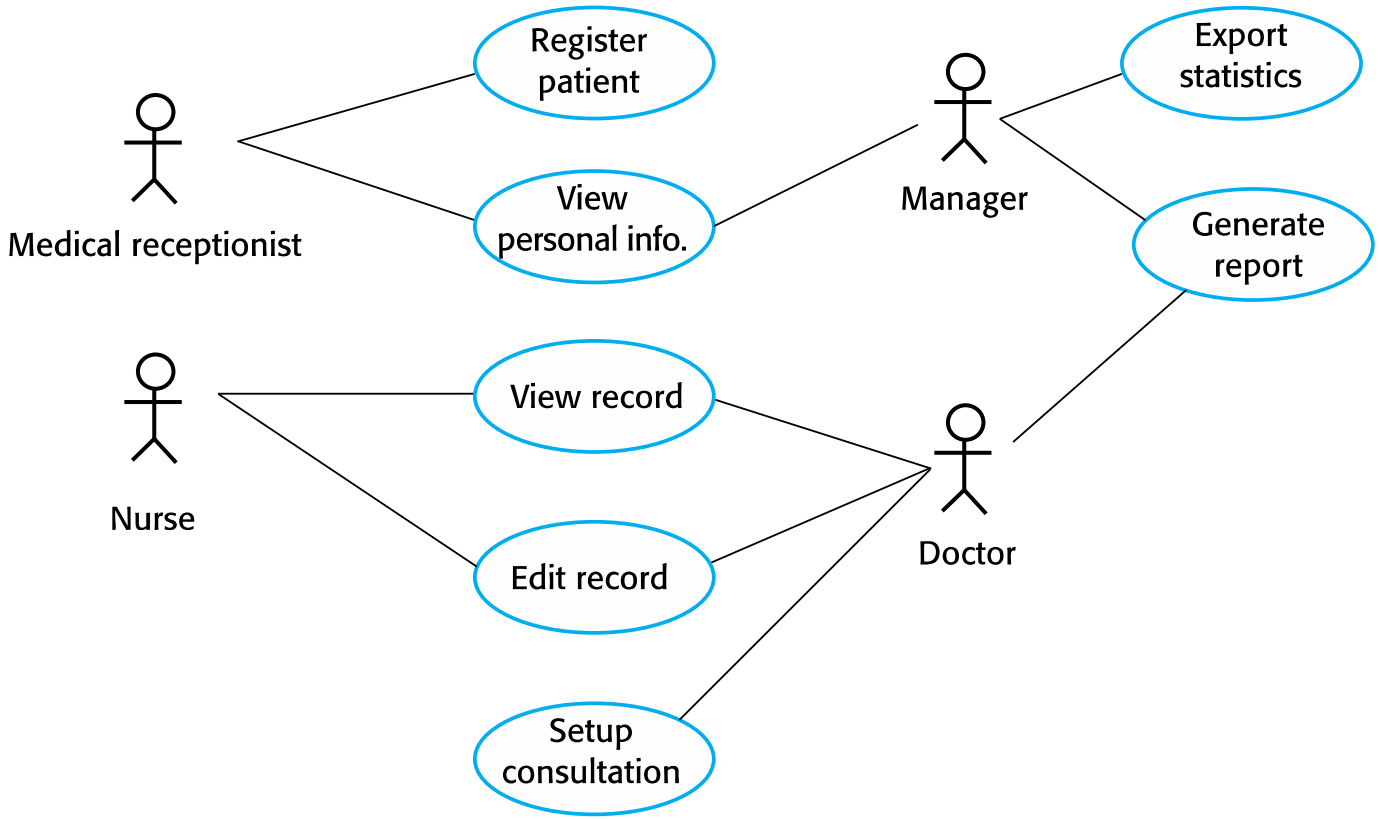
Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 – r1) ≥ (r1 – r0))	CompDose = round ((r2 – r1)/4) If rounded result = 0 then CompDose = MinimumDose

Use Cases

- Use-cases are a kind of scenario that are included in the UML.
  - Identify the actors in an interaction and which describe the interaction itself.

- A set of use cases should describe all possible interactions with the system.
  - High-level graphical model (UML Use-Case Diagram) is used to summarize all use-cases.
  - UML Sequence Diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

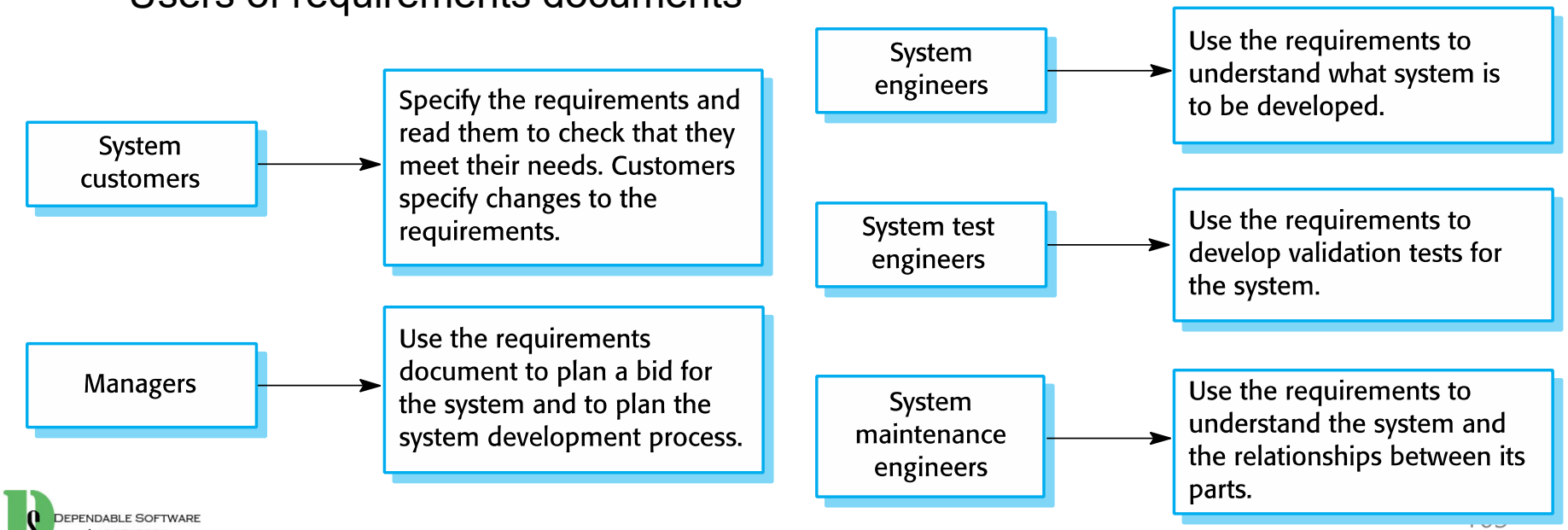
Mentcare System : Use-Case Diagram



The Software Requirements Document

- The software requirements document is **the official statement** of what is required of the system developers.
  - Should include both a definition of user requirements and a specification of the system requirements.
  - It is NOT a design document.
    - As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

- Users of requirements documents

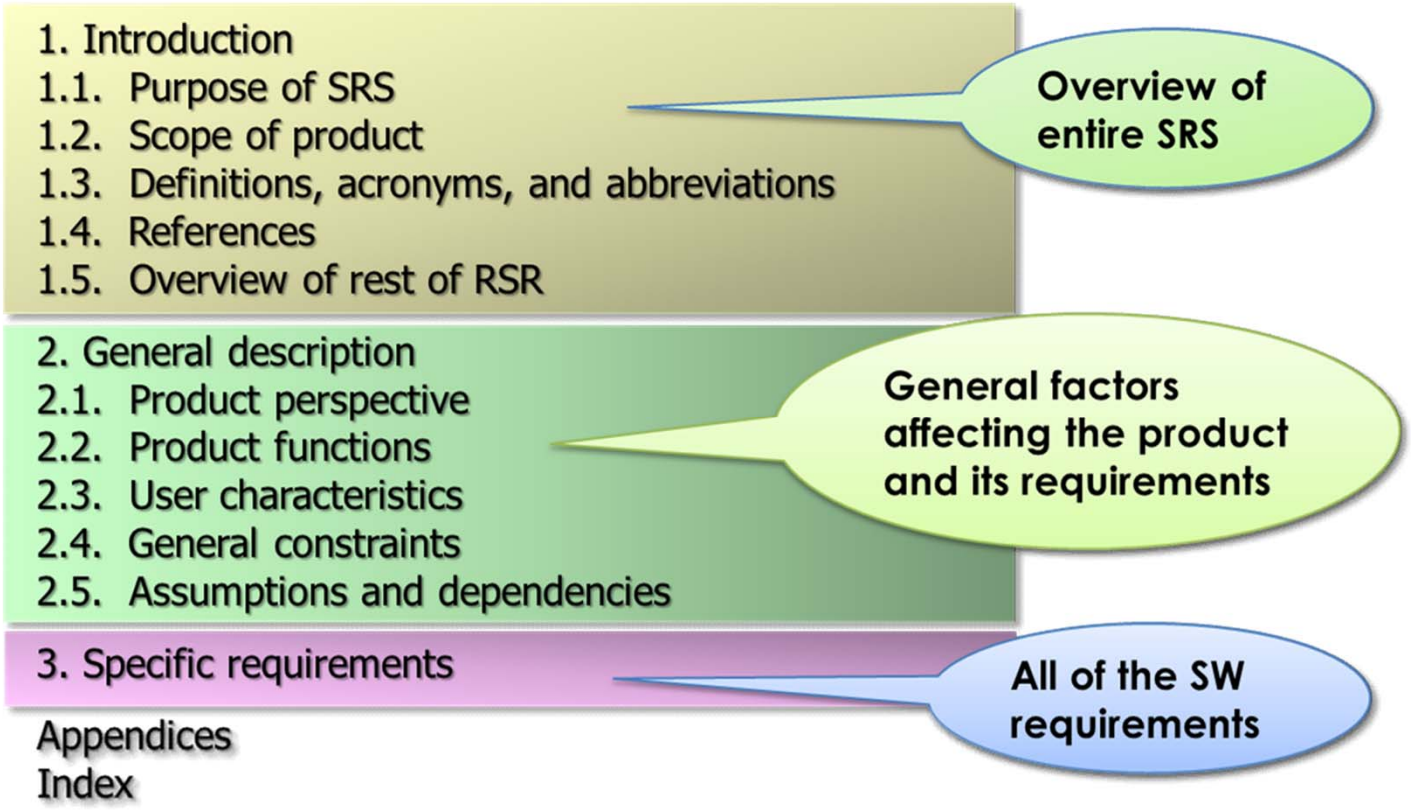


Requirements Document Variability

- Information in requirements document depends on the type of system and the approach to development used.
  - If systems are developed incrementally, it will typically have less detail in the requirements document.

- **Requirements documents standards** have been designed.
  - E.g., **IEEE standards**
  - Mostly applicable to the requirements for large systems engineering projects

SRS Standard: IEEE STD 830-1998



SRS Standard: IEEE STD 830-1998

3.1 External interface requirements
- 3.1.1 User interfaces
- 3.1.2 Hardware interfaces
- 3.1.3 Software interfaces
- 3.1.4 Communications interface

3.2 Functional requirements

3.2.1 Functional requirements 1
- 3.2.1.1 Introduction
- 3.2.1.2 Inputs
- 3.2.1.3 Processing
- 3.2.1.4 Outputs

3.2.2 Functional requirements 2

3.2.n Functional requirements n

3.3 Performance requirements

3.4 Database

3.5 Design Constrains
- 3.5.1 Standards compliance
- 3.5.2 Hardware limitations

3.6 Attributes
- 3.6.1 Reliability
- 3.6.2 Availability
- 3.6.3 Security
- 3.6.4 Maintainability
- 3.6.5 Portability

Elements of Requirements Documents

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Requirements Validation

Requirements Validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.

- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

- Requirements checking
  - Validity : Does the system provide the functions which best support the customer's needs?
  - Consistency : Are there any requirements conflicts?
  - Completeness : Are all functions required by the customer included?
  - Realism : Can the requirements be implemented given available budget and technology
  - Verifiability : Can the requirements be checked?

Requirements Validation Techniques

- **Requirements reviews**
  - Systematic manual analysis of the requirements.
- **Prototyping**
  - Using an executable model of the system to check requirements
- **Test-case generation**
  - Developing tests for requirements to check testability.

Requirements Reviews

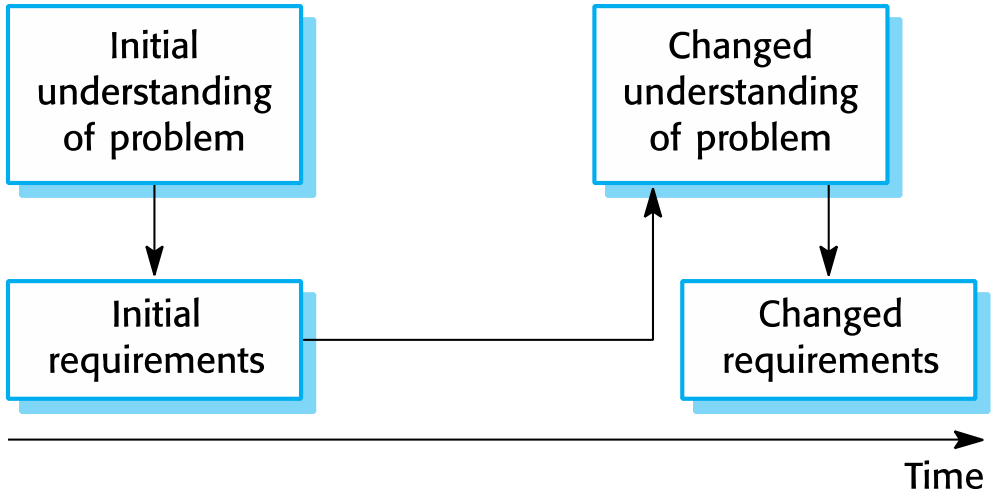
- Regular reviews should be held while the requirements definition is being formulated.
  - Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal.
  - Good communications between developers, customers and users can resolve problems at an early stage.
- Review checks
  - Verifiability
    - “Is the requirement realistically testable?”
  - Comprehensibility
    - “Is the requirement properly understood?”
  - Traceability
    - “Is the origin of the requirement clearly stated?”
  - Adaptability
    - “Can the requirement be changed without a large impact on other requirements?”

Requirements Change

Changing Requirements

- The business and technical environment of the system **always changes**.
  - New hardware may be introduced.
  - Business priorities may change.
  - New legislation and regulations to abide by may be introduced.

- The people who pay for a system and the users of that system are rarely the same people.
  - System customers may conflict with end-user requirements after delivery, and new features may have to be added for user support

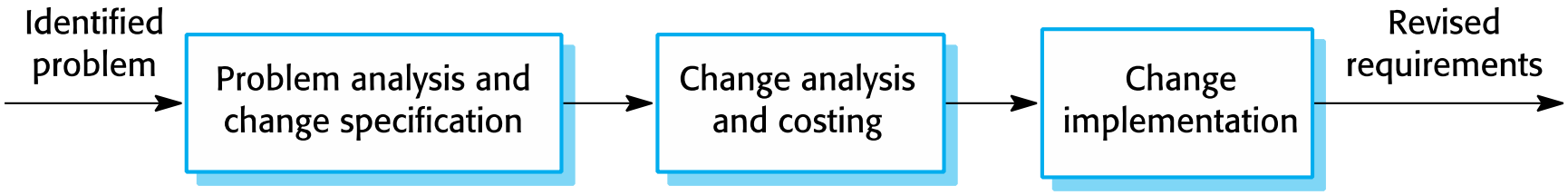


Requirements Management

- **Requirements management** is the process of managing changing requirements during the requirements engineering process and system development, and even after delivery
  - We need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes.
  - Need to establish a formal process for making change proposals and linking these to system requirements.

Requirements Change Management

- Deciding if a requirements change should be accepted or not.
  - Problem analysis and change specification
    - The problem or the change proposal is analyzed to check that it is valid.
  - Change analysis and costing
    - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
  - Change implementation
    - The requirements document and, where necessary, the system design and implementation, are modified.



Key Points

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used.
- They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.
- Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
- You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.

Key Points

- Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
- Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

Chapter 5. System Modeling

Topics Covered

- Context models
- Interaction models
- Structural models
- Behavioral models
- Model-driven engineering

System Modeling

- **System modeling** is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
  - Help analysts to understand the functionality of the system and communicate with customers
  - Mostly based on notations in the **Unified Modeling Language (UML)**

- System perspectives (views)
  - **External perspective** : models the context or environment of the system
  - **Interaction perspective** : models the interactions between a system and its environment, or between the components of a system
  - **Structural perspective** : models the organization of a system or the structure of the data processed by the system
  - **Behavioral perspective** : models the dynamic behavior of the system and how it responds to events

Use of Graphical Models

- Use of the **UML** graphical models
  - **Communication** (Sketch)
    - As a means of facilitating discussion about an existing or proposed system
    - Incomplete and incorrect models are OK as their role is to support discussion.
  - **Documentation** (Blueprint)
    - Models should be an accurate representation of the system but need not be complete.
  - **System generation** (Code generation)
    - Models must be both correct and complete.

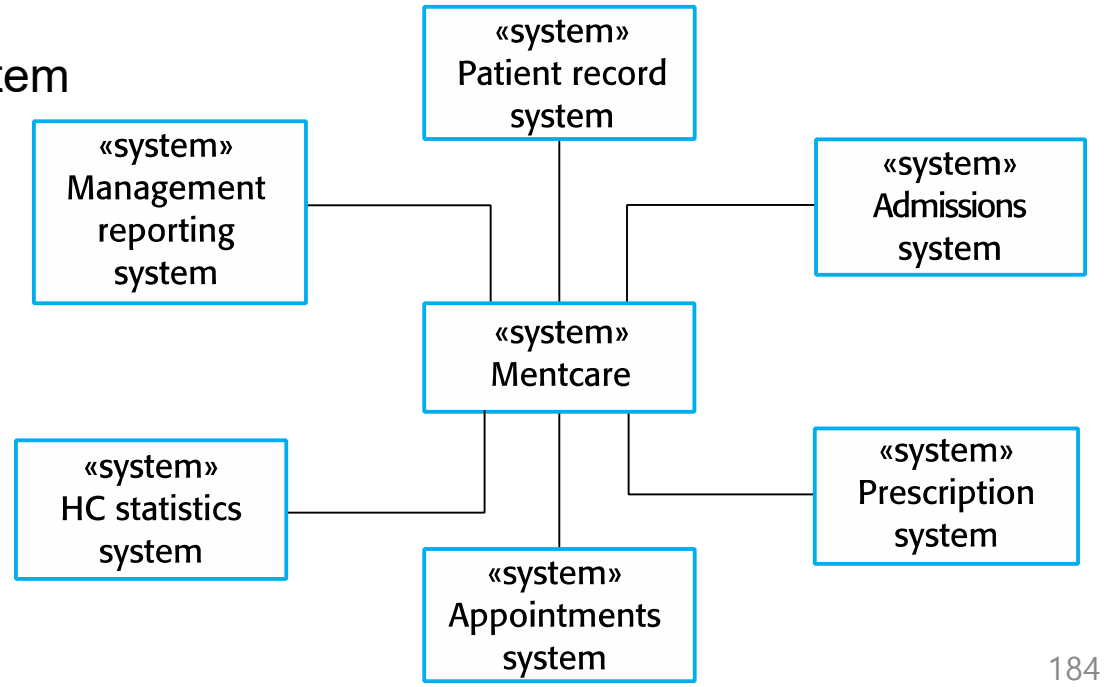
- UML diagrams used for system modeling
  - **Activity diagrams** show the activities involved in a process or in data processing.
  - **Use case diagrams** show the interactions between a system and its environment.
  - **Sequence diagrams** show interactions between external actors and the system, or between system components.
  - **Class diagrams** show the object classes and the associations between these classes.
  - **State diagrams** show how the system reacts to internal and external events.

Context Models

Context Models

- **Context models** illustrate the operational context of a system.
  - External perspective
  - Show what lies outside the system boundaries.
    - Social and organizational concerns may affect the decision on where to position system boundaries.
  - **Architectural models** show the system and its relationship with other systems.

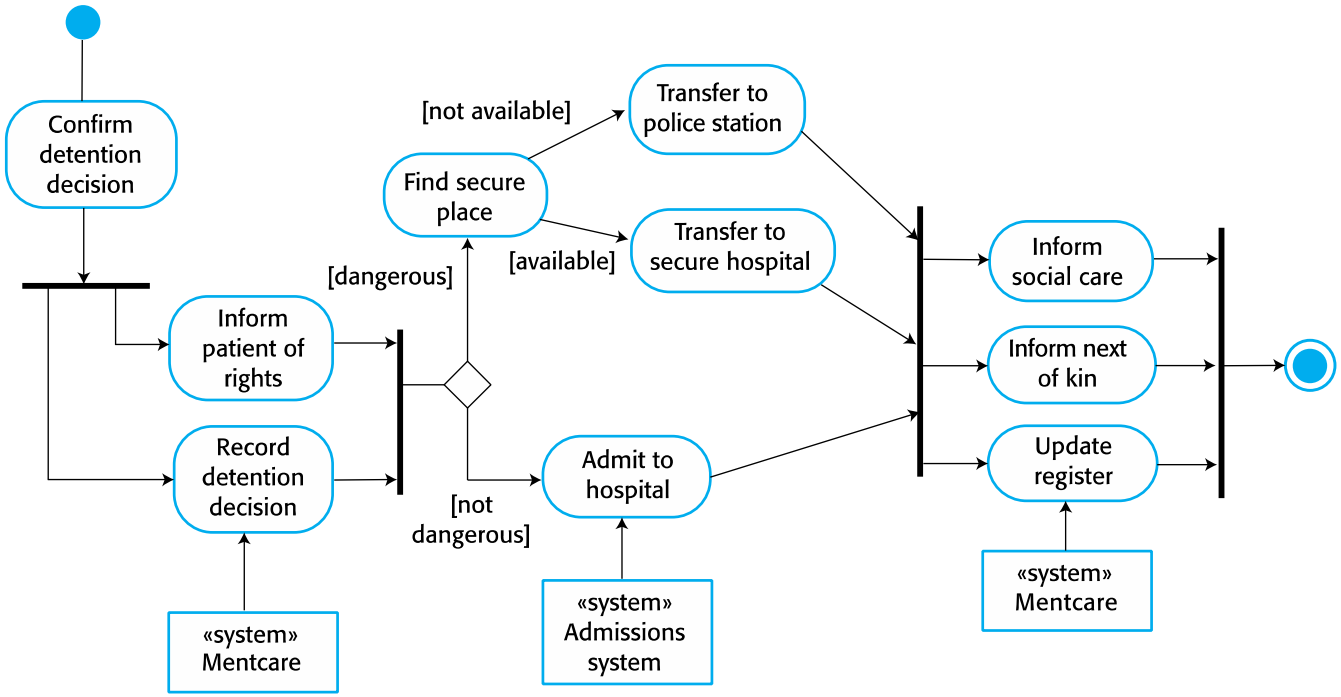
- Example: Mentcare System



Process Models

- **Process models** reveal how the system is used in business processes.
  - Context models simply show the other systems in the environment, not how the system will be used in that environment.
  - **UML activity diagrams** may be used to define business process models.
  - Not just external perspective, but mixed with others

- Example : Involuntary Detention (강제구금)



Interaction Models

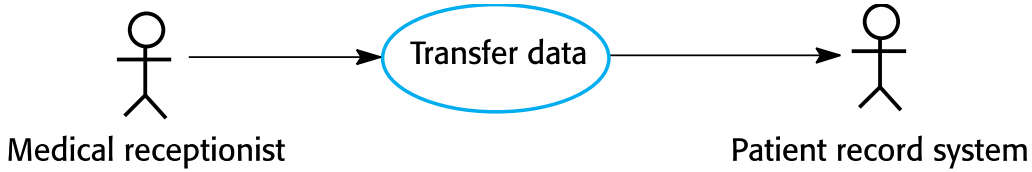
Interaction Models

- **Use case diagrams** and **sequence diagrams** are often used for interaction modelling.
  - **User interaction** helps to identify user requirements.
  - **System-to-system interaction** highlights the communication problems that may arise.
  - **Component interaction** helps to understand if a proposed system structure is likely to deliver the required system performance and dependability.

Use Case Modeling

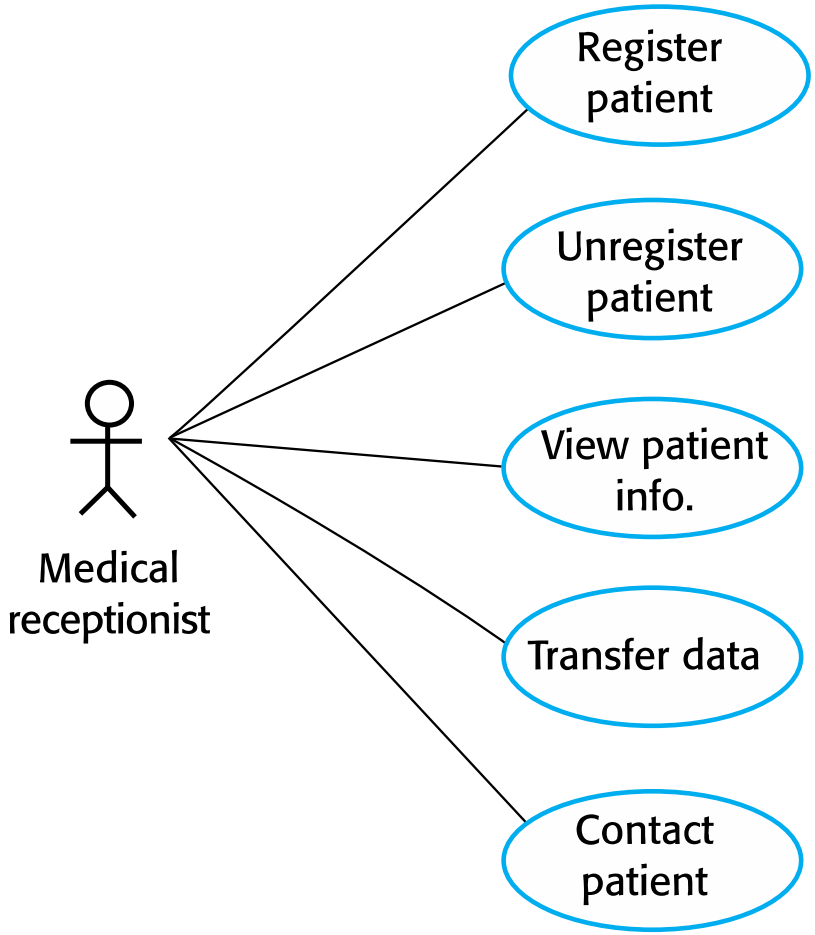
- **Use case** represents a discrete task that involves external interaction with a system.
  - Use case is a text scenario.
  - Use case diagrams provide an overview of all use cases.

- Example : “Transfer Data” use-case in Mentcare System



MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient’s diagnosis and treatment.
Data	Patient’s personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

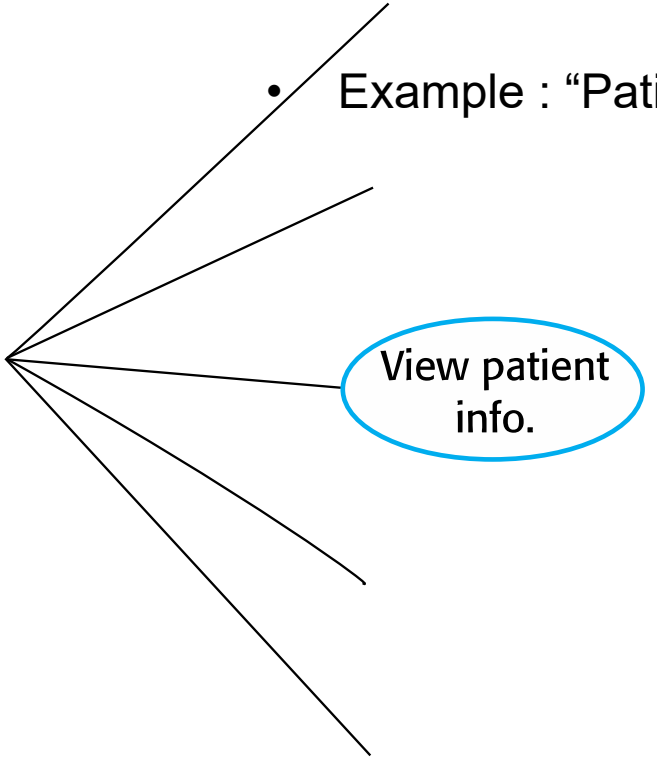
Mentcare System : Use-Cases of Medical Receptionist

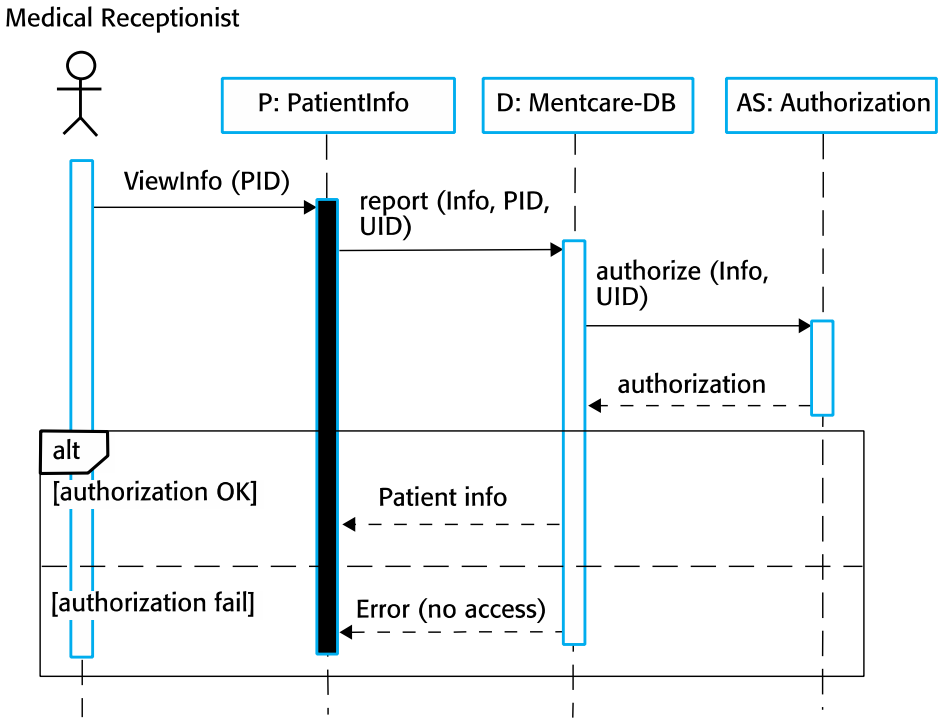


Sequence Diagrams

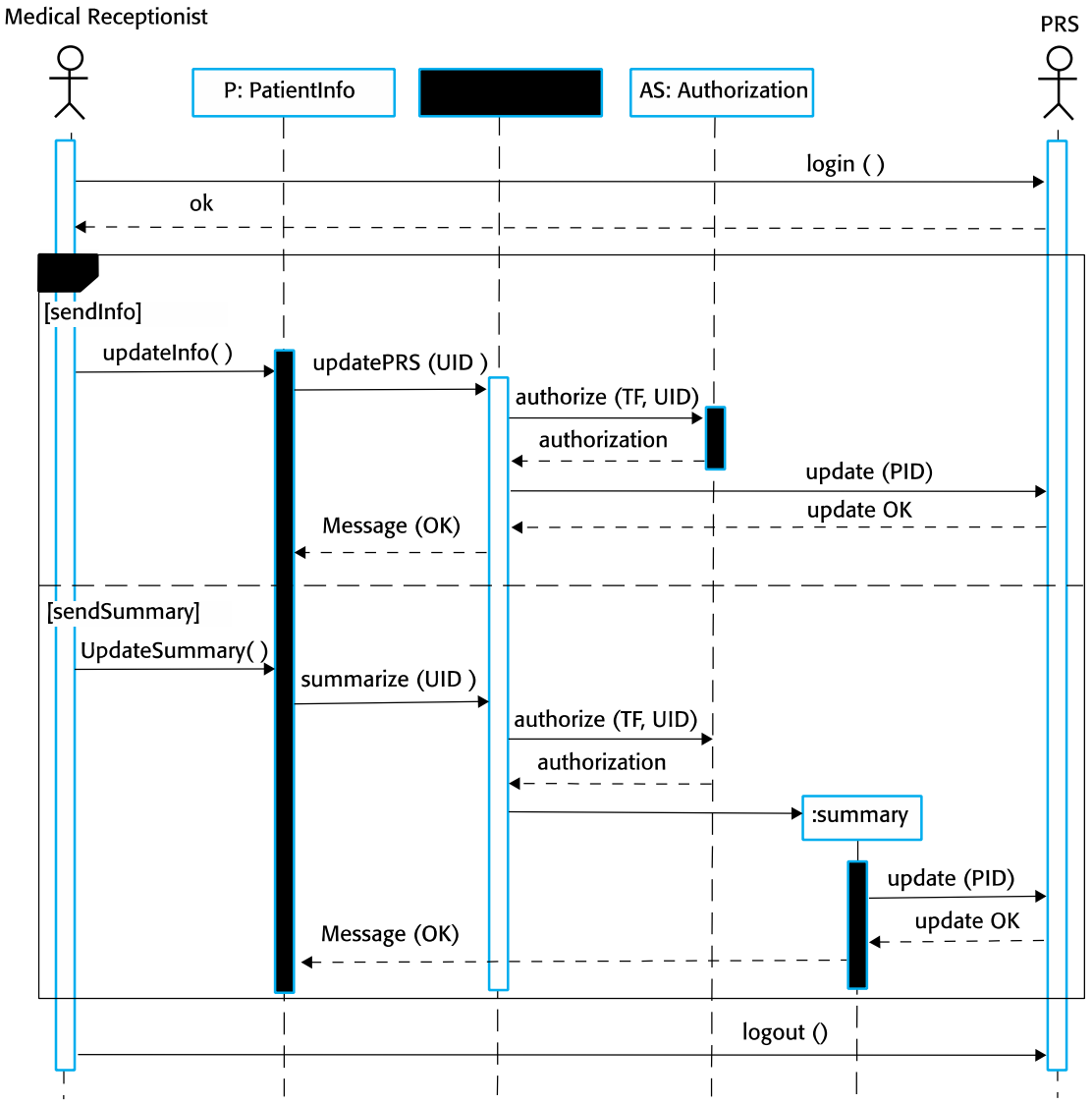
- **Sequence diagrams** show the sequence of interactions that take place during a particular use case or use case instance.
  - The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
  - Interactions between objects are indicated by annotated arrows.

- Example : “Patient Information” use-case in Mentcare System





Mentcare System : Sequence Diagram for Transfer Data



Structural Models

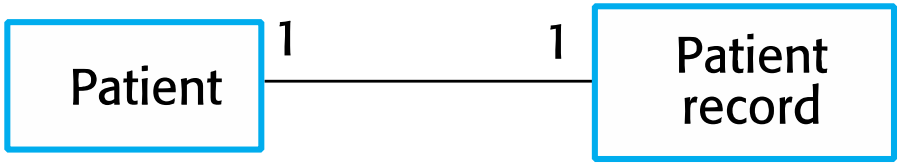
Structural Models

- **Structural models** display the organization of a system in terms of the components that make up that system and their relationships.
  - **Static models** show the structure of the system design.
    - Class diagram
  - **Dynamic models** show the organization of the system when it is executing.
    - Component diagram, Object diagram, Composite structure diagram

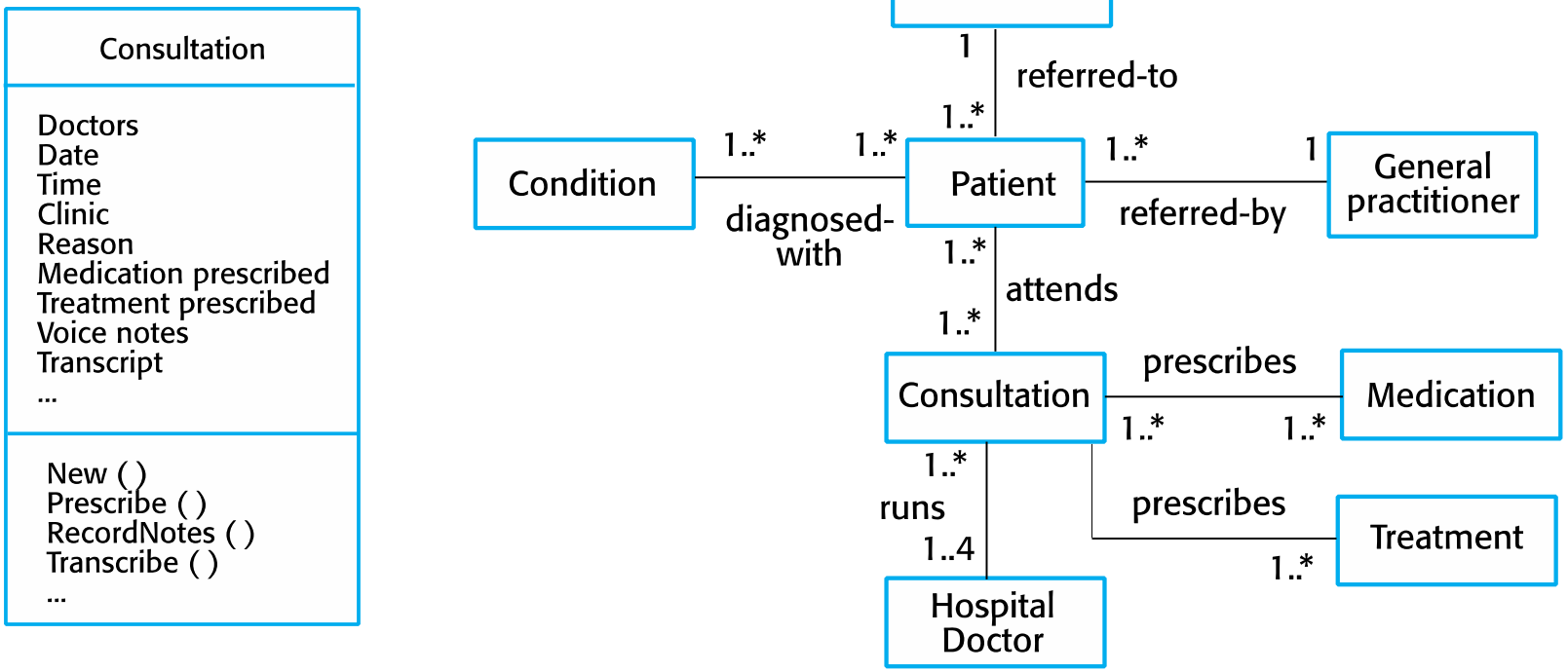
Class Diagrams

- **Class diagrams** are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
  - An object class is a general definition of one kind of system object.
  - An association is a link between classes that indicates that there is some relationship between these classes.

- **Domain models** in OOAD
  - A class diagram-like model to identify objects in early phases of SDLC
  - Objects may represent something in the real world, such as a patient, a prescription, doctor, etc. as well as actual objects will be implemented with software.



Mentcare System : Classes and Associations



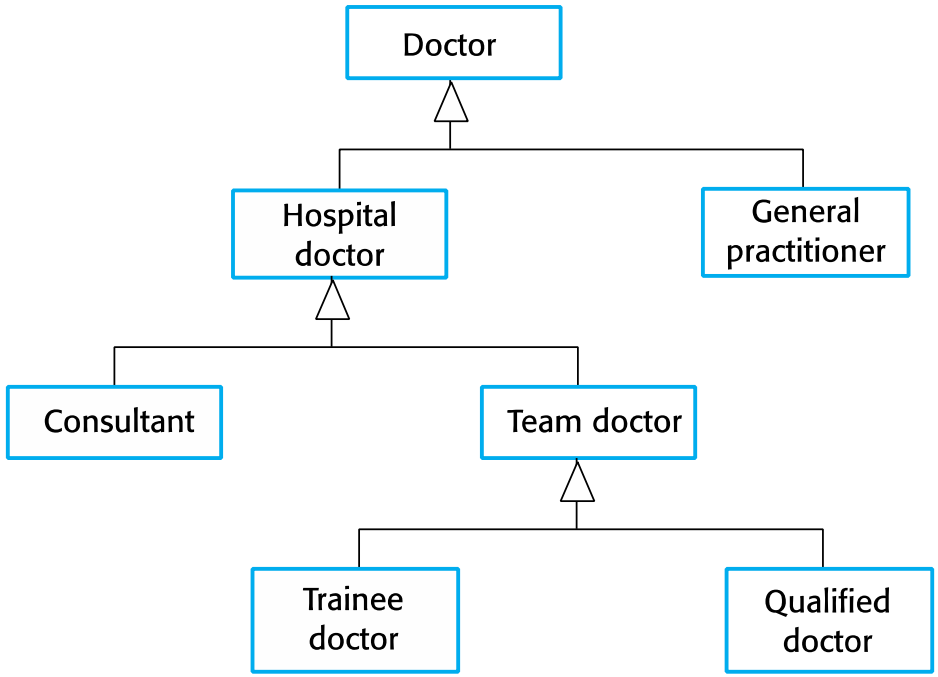
Generalization

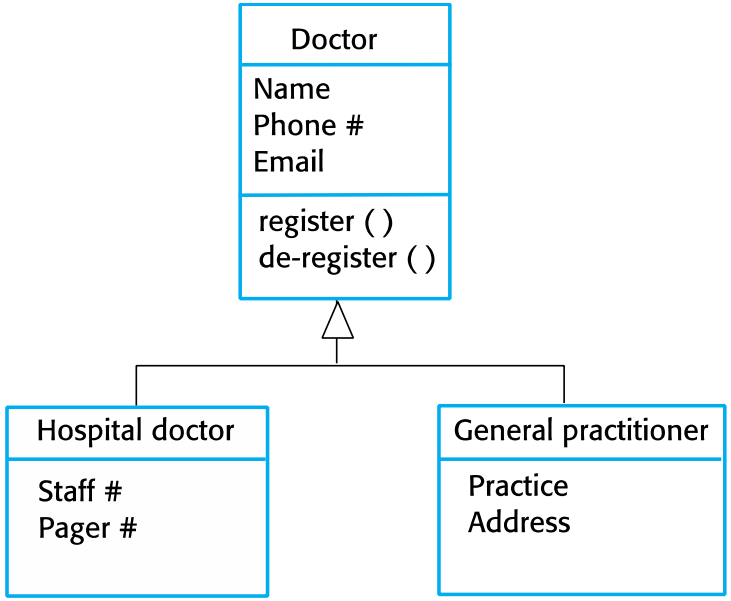
- **Generalization** is a technique that we use to manage complexity.
  - Allows us to infer that different members of these classes have some common characteristics
  - Example: squirrels and rats are rodents.

  - In object-oriented languages, such as Java, generalization is implemented using the **class inheritance** mechanisms built into the language.

- In a generalization,
  - The attributes and operations associated with higher-level classes are also associated with the lower-level classes.
  - The lower-level classes are subclasses which inherit the attributes and operations from their parent classes.
  - Lower-level classes can add more specific attributes and operations.

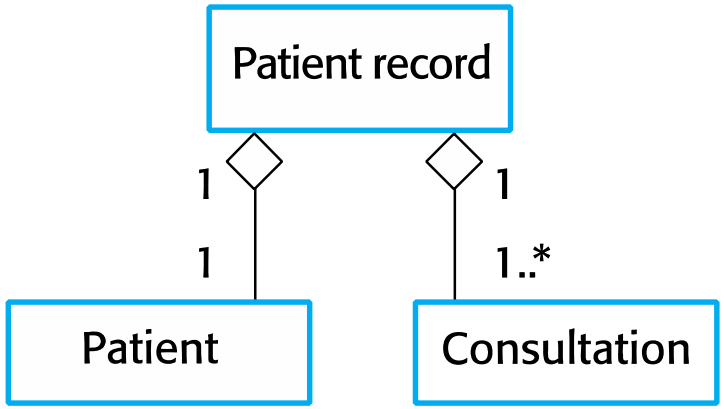
Mentcare System : A Generalization Hierarchy





Aggregation

- **Aggregation** shows how classes are composed of other classes.
  - Aggregation models are similar to the part-of relationship in semantic data models.



Behavioral Models

Behavioral Models

- **Behavioral models** model the dynamic behavior of a system when it executes.
  - They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.

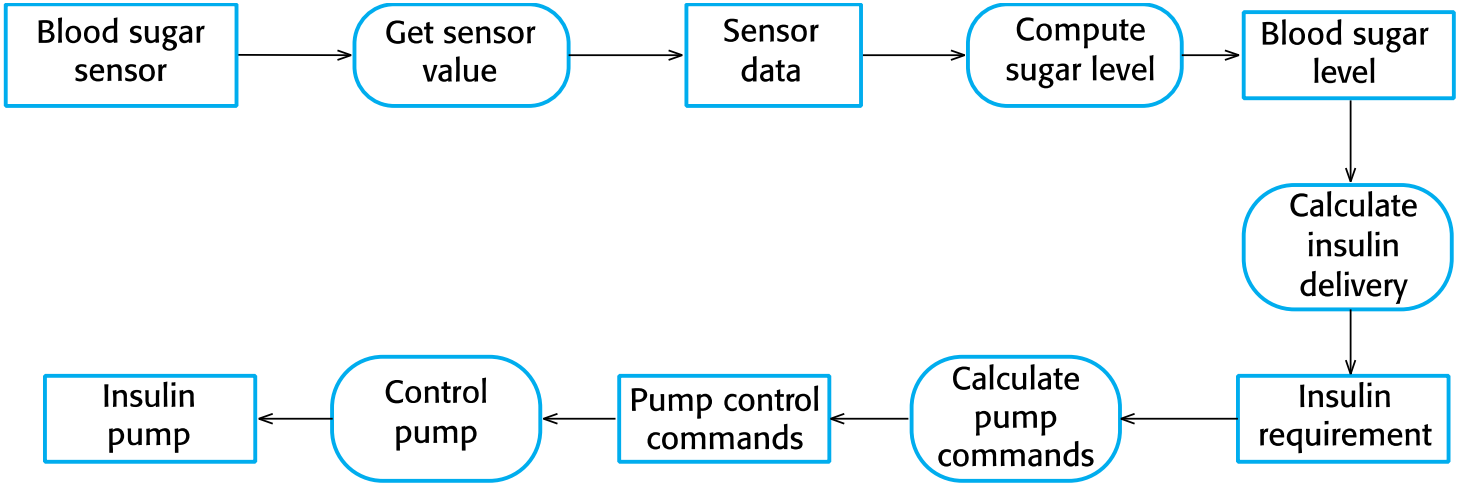
- Two types of stimulus from environment:
  - **Data** : Some data arrives that must be processed by the system.
  - **Events** : Some event happens that triggers system processing. Events may have associated data.

- Behavioral models
  - **Data-driven model**
  - **Event-driven model**
  - **State machine model**

Data-Driven Models

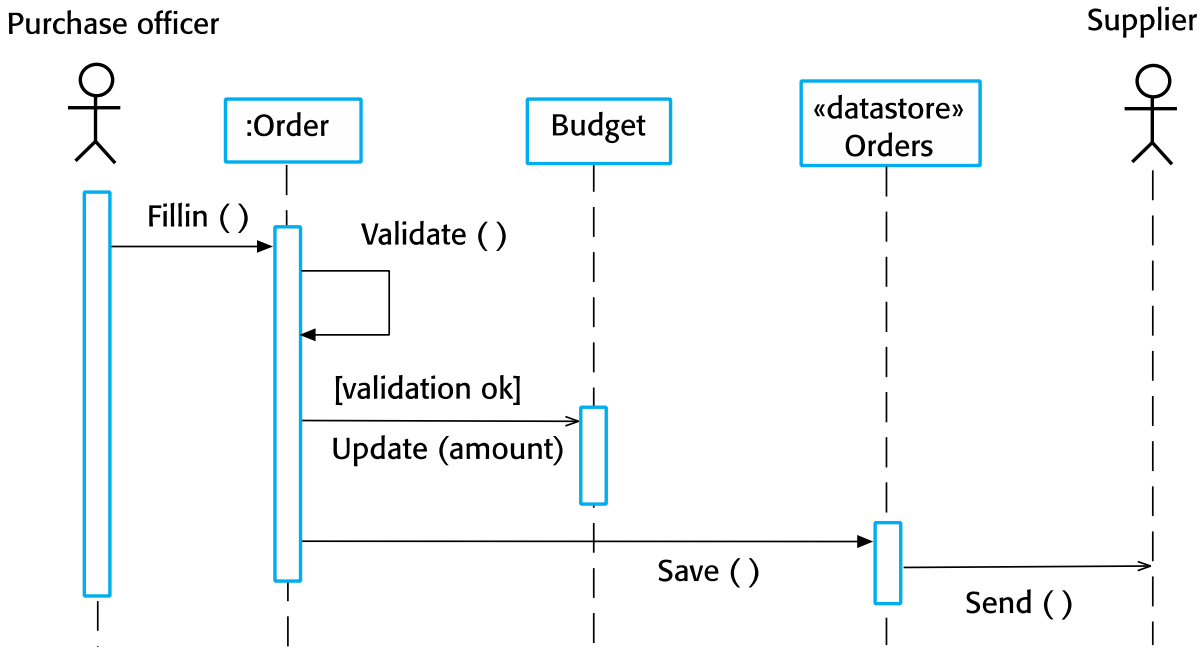
- **Data-processing systems** are primarily driven by data.
  - Controlled by the data input to the system, with relatively little external event processing
  - Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
  - **Data flow diagram**, **Activity diagram**

- Example : Insulin Pump's operations



Event-Driven Models

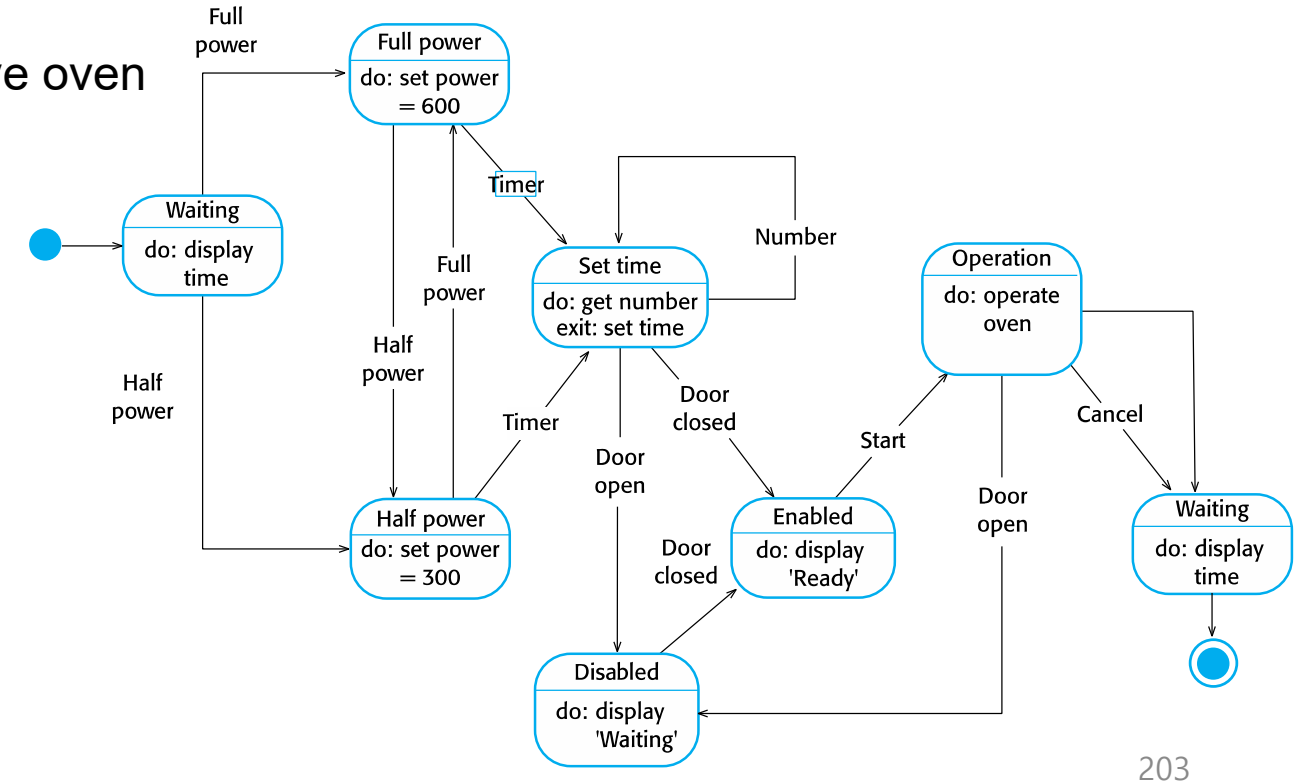
- Real-time systems are often **event-driven**, with minimal data processing.
  - Event-driven modeling shows how a system responds to external and internal events.
    - Assume that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another
  - **Sequence diagram**



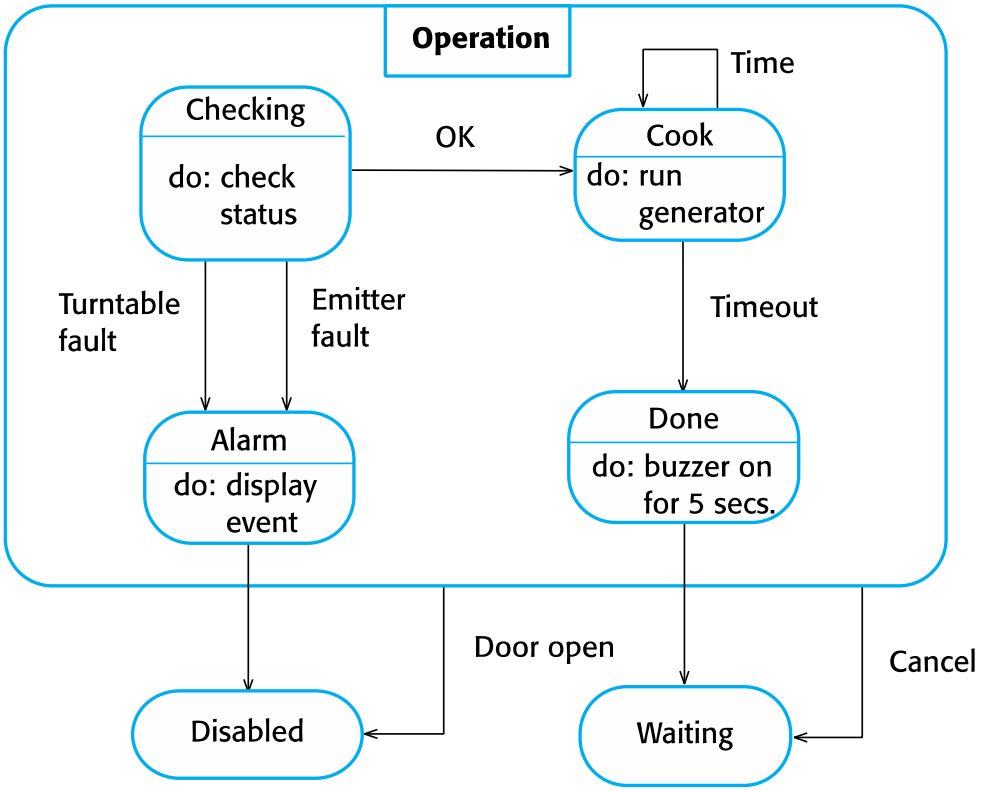
State Machine Models

- **State machine models** model the behaviour of the system in response to external and internal events.
  - Show system states as nodes and events as arcs between these nodes.
  - When an event occurs, the system moves from one state to another.
  - **Statecharts**

- Example : Microwave oven



Microwave Oven - Operations State



States and Stimuli for Microwave Oven

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Model-Driven Engineering

Model-Driven Engineering

- **Model-driven engineering (MDE)** is a software development approach where models rather than programs are the principal outputs of the development process.
  - The programs executing on a hardware/software platform are **generated automatically from the models**.
  - Software engineers no longer should be concerned with programming language details or the specifics of execution platforms.

- **MDE** is still at an early stage of development.
  - Advantages
    - Allows systems to be considered at higher levels of abstraction
    - Generating code automatically means that it is cheaper to adapt systems to new platforms.
  - Disadvantages
    - Models for abstraction and not necessarily right for implementation.
    - Savings from generating code may be outweighed by the costs of developing translators for new platforms.

Model-Driven Architecture

- **Model-driven architecture (MDA)** is a model-focused approach to software design and implementation.
  - The precursor of more general model-driven engineering
  - Models at different levels of abstraction are created.
    - Generate a working program without manual intervention from a high-level platform independent model
    - CIM, PIM, and PSM
  - Often use a subset of UML models to describe a system

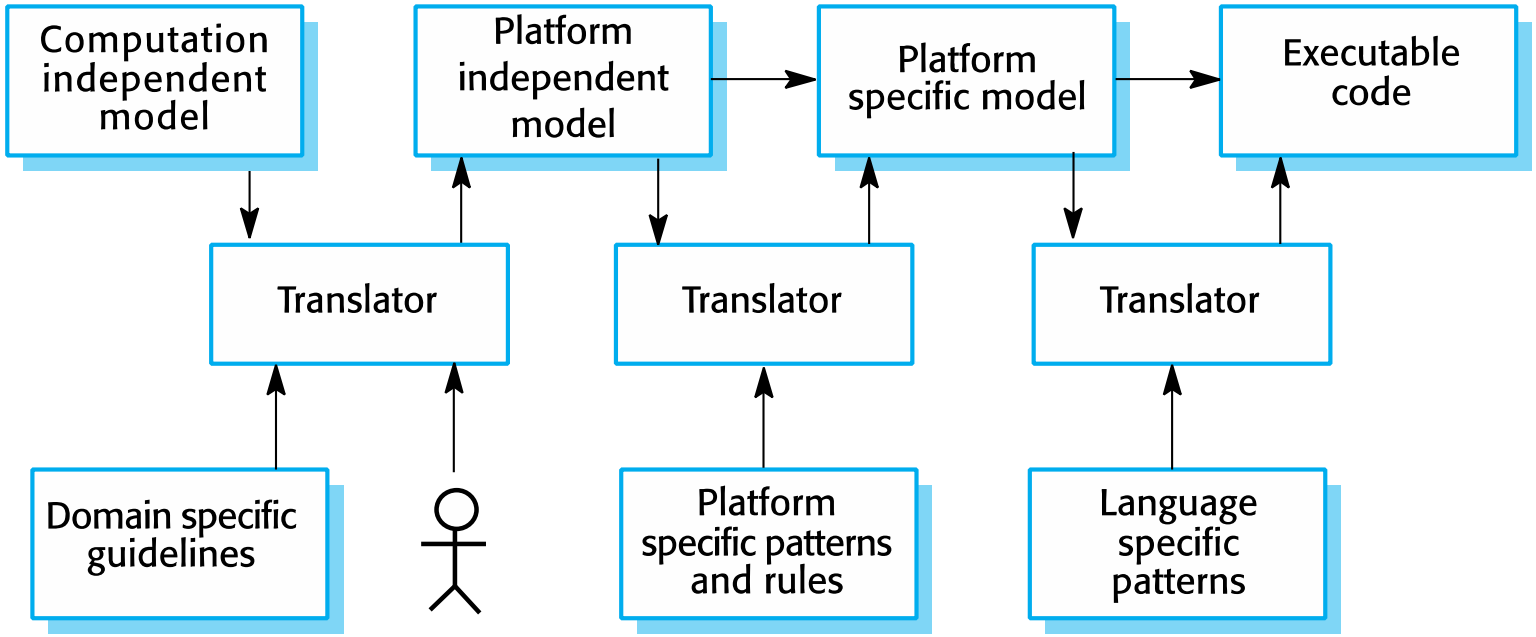
Types of Models in MDA

- **Computation independent model (CIM)**
  - Models the important domain abstractions used in a system
  - CIMs are sometimes called domain models.

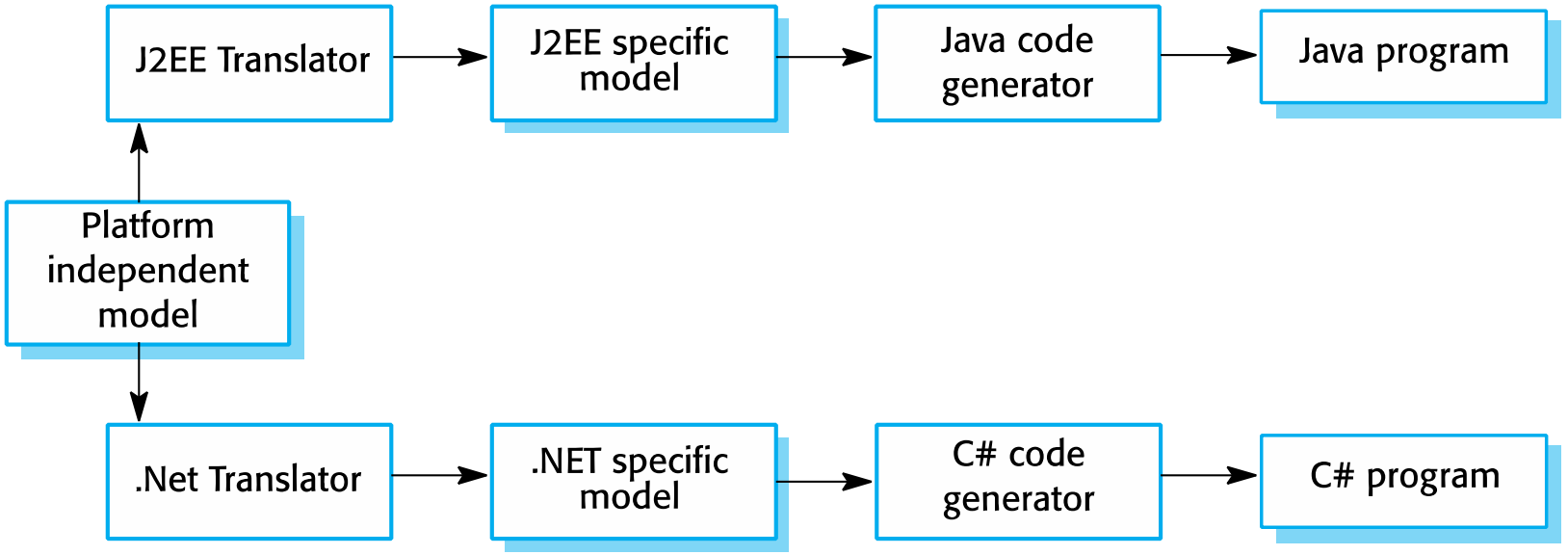
- **Platform independent model (PIM)**
  - Models the operation of the system without reference to its implementation.
  - PIMs are usually described using UML models that show the static system structure and how they respond to external and internal events.

- **Platform specific models (PSM)**
  - Transformations of the platform-independent model into a separate PSM for each application platform.
  - In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

MDA Transformations



Multiple Platform-Specific Models



Adoption of MDA

- Limitations on adopting MDE/MDA
  - **Specialized tool support** is required to convert models from one level to another
  - There is limited tool availability and organizations may require tool adaptation and customization to their environment

- Models are a good way of facilitating discussions about a software design.
  - However, the abstractions that are useful for discussions may not be the right abstractions for implementation.
  - For most complex systems, implementation is not the major problem – requirements engineering, security and dependability, integration with legacy systems and testing are all more significant.

- The arguments for platform-independence are only valid for large, long-lifetime systems.
  - For software products and information systems, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.

Agile Methods and MDA

- The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto.
  - Few agile developers feel comfortable with model-driven engineering.

- If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.

Key Points

- A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behaviour.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.

Key Points

- Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system's behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

Chapter 6. Architectural Design

Topics Covered

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

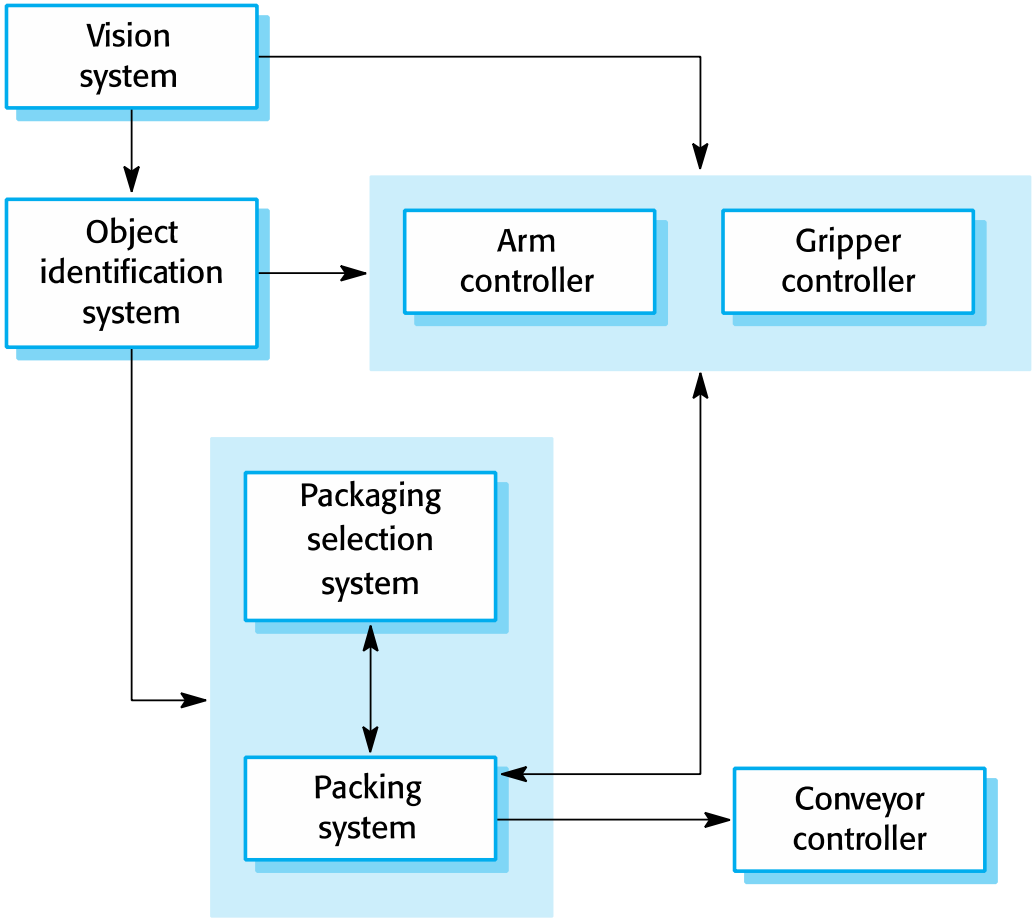
Architectural Design

- **Architectural design** is concerned with understanding how a software system should be organized and designing the overall structure of that system.
  - A critical link between requirements engineering and design
  - Identifies the main structural components in a system and the relationships between them.

- **Architecture model** describes how the system is organized as a set of communicating components.

- Agility and Architecture
  - It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
  - Refactoring the system architecture is usually expensive, because it affects so many components in the system.

The Architecture of Packing Robot Control System



Architectural Abstraction

- Architecture in the small
  - Concerned with the architecture of individual programs
  - Concerned with the way that an individual program is decomposed into components

- Architecture in the large
  - Concerned with the architecture of complex enterprise systems that include other systems, programs and program components
    - Enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of Architectural Design

- 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 simply
  - The most frequently used method for documenting software architectures
  - But, lack of semantics do not show the types of relationships between entities nor the visible properties of entities in the architecture.
  - The semantics of architectural models depend on how the models are used.

- **Box and Line Diagrams**
  - Very abstract - 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.

- **Extensions of UML models**
  - Extending Component diagram
  - Class diagram, Component diagram, Composite structure diagram
  - Not widely used yet

Use of Architectural Models

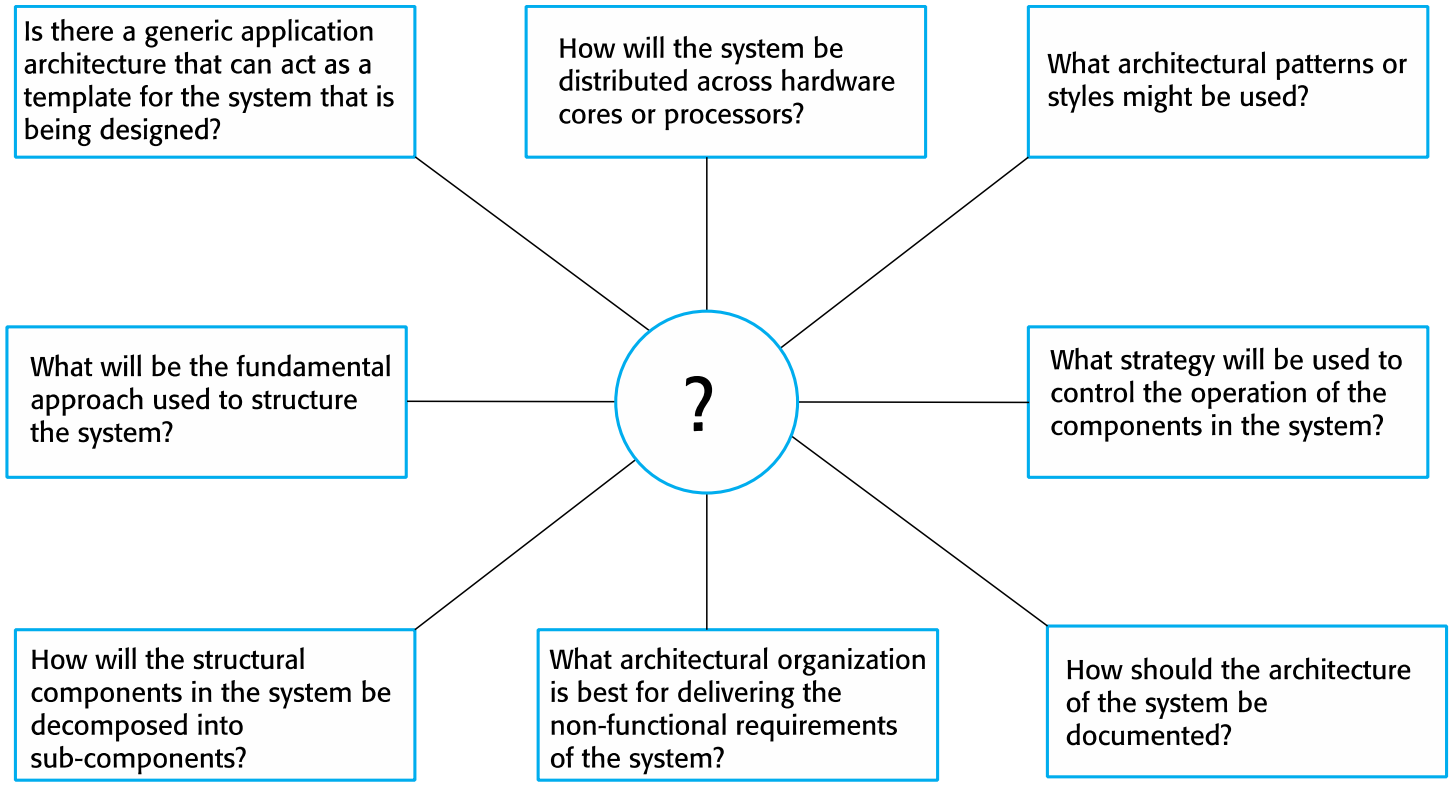
- As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail.
  - Stakeholders can relate to it and understand an abstract view of the system. 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
  - To produce a complete system model that shows the different components in a system, their interfaces and their connections.

Architectural Design Decisions

Architectural Design Decisions

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



Architecture and System Characteristics

- Performance
  - Localize critical operations and minimize communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localize 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.

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**'.
  - Capture the essence of an architecture and can be instantiated in different ways.

Architectural Views

Architectural Views

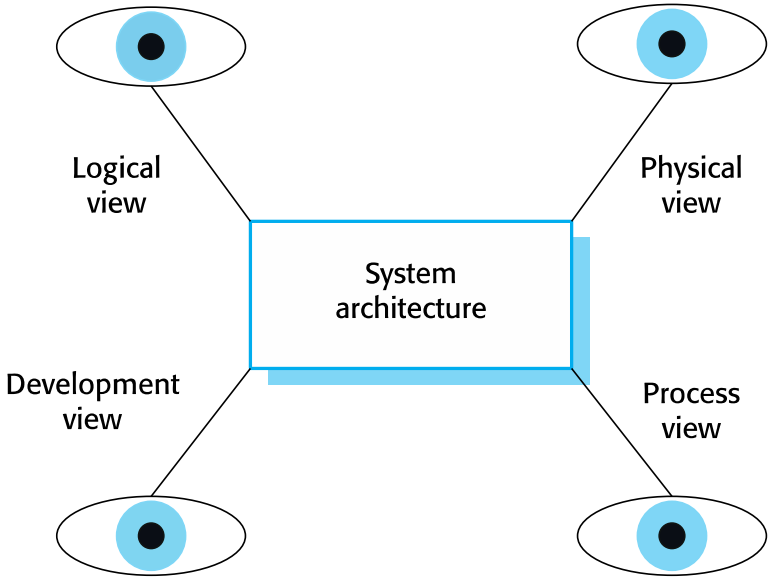
- Each architectural model only shows **one view** showing
  - How a system is decomposed into modules,
  - How the run-time processes interact, or
  - Which system components are distributed across a network.

- We need **multiple views** of the software architecture for both design and documentation purposes.
  - What views are useful when designing and documenting a system's architecture?
  - What notations should be used for describing architectural models?

4 + 1 View Model of Software Architecture

- Logical view : shows the key abstractions in the system as objects or object classes
- Process view : shows how, at run-time, the system is composed of interacting processes
- Development view : shows how the software is decomposed for development
- Physical view : shows the system hardware and how software components are distributed across the processors in the system

- Related 4 views with use cases or scenarios (+1)

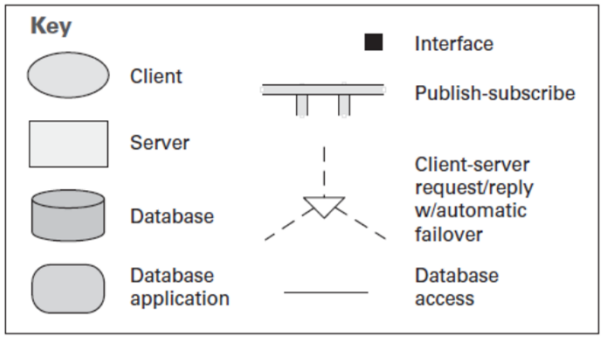


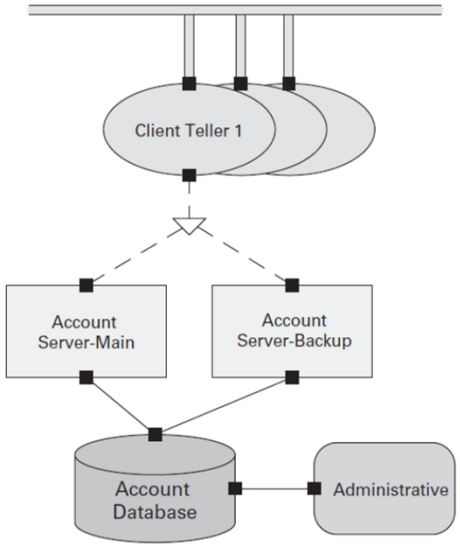
Representing Architectural Views

- **Unified Modeling Language** (UML) is a candidate notation for describing and documenting system architectures.
  - Component diagram, Package diagram, Class diagram, etc.
  - However, UML does not include abstractions appropriate for high-level system description.

- **Architectural description languages (ADLs)** have been developed.
  - But, are not widely used.

- **Naive diagrams** have been widely used.
  - Example : C&C View





Architectural Patterns

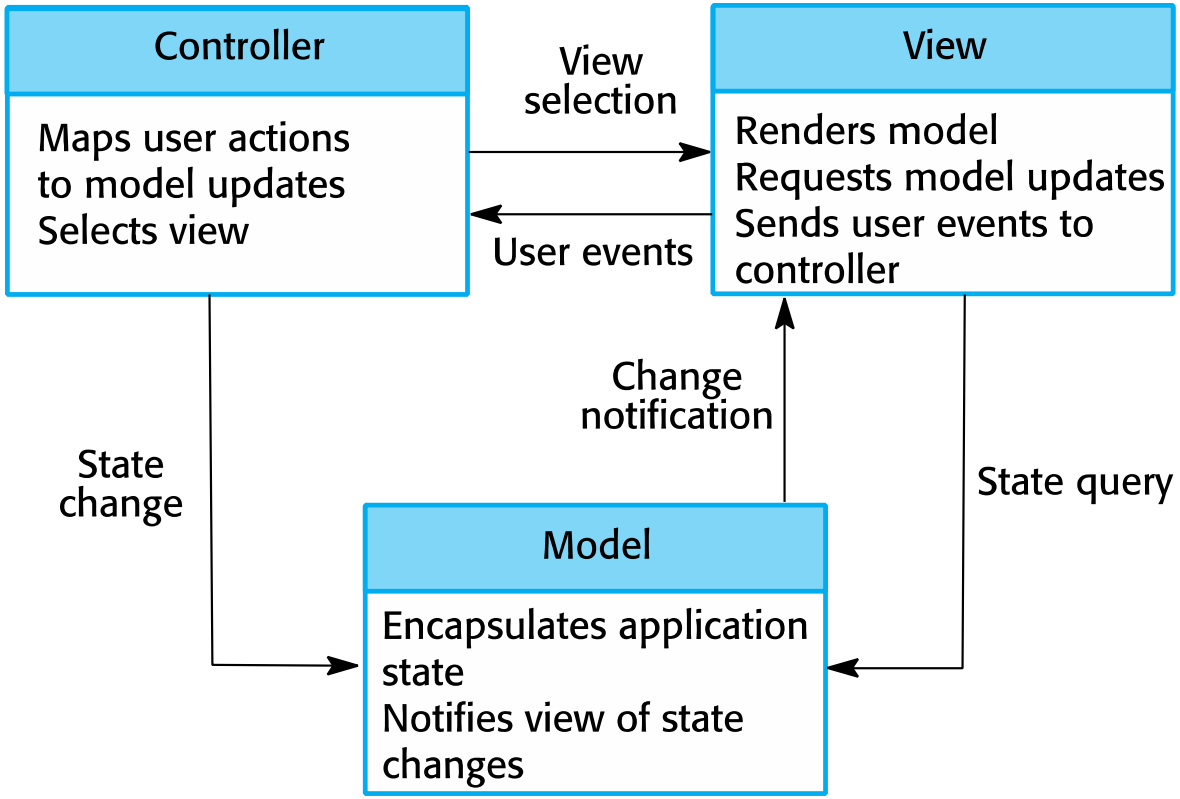
Architectural Patterns

- **Architectural pattern** is a stylized description of good design practice, which has been tried and tested in different environments.
  - Include information about when they are and when the are not useful.
  - Example:
    - **MVC (Model-View-Controller)**
    - **Layered**
    - **Repository**
    - **Client-Server**
    - **Pipe & Filter**
    - etc.

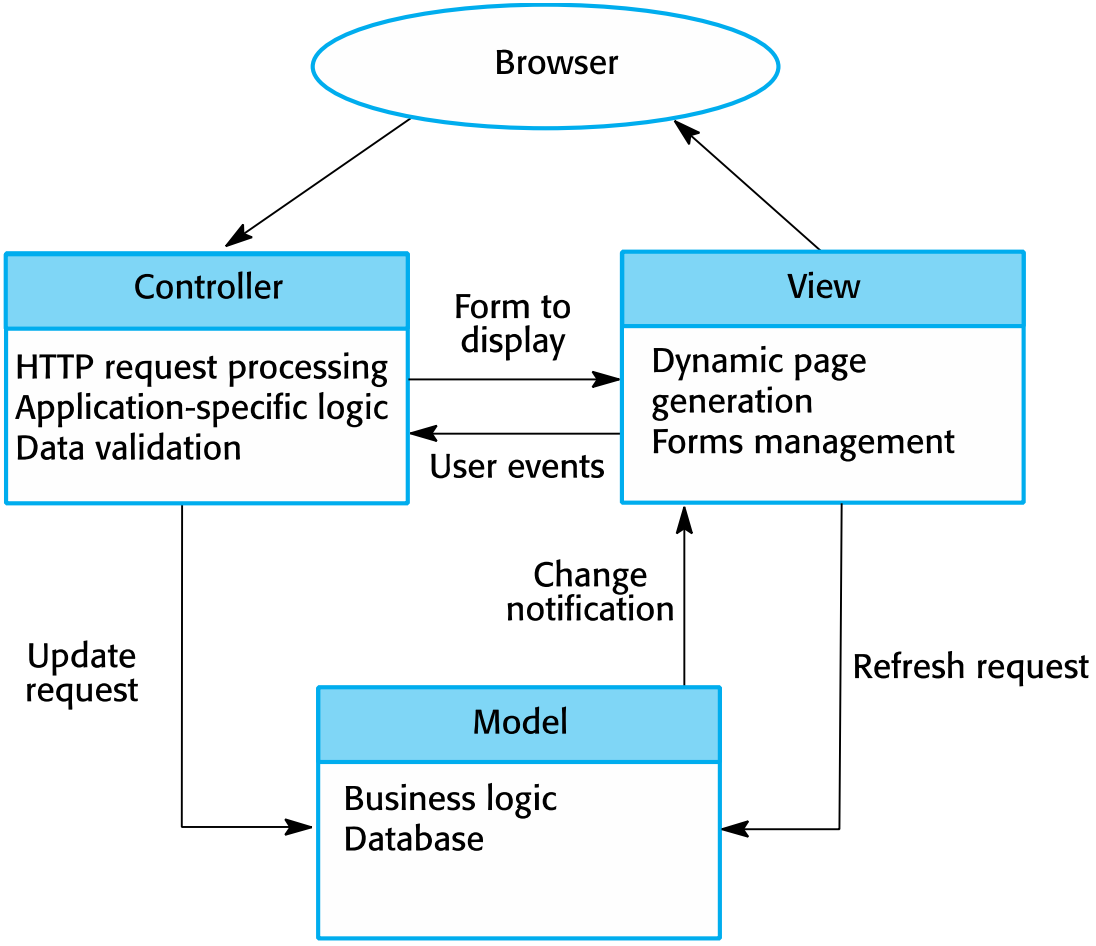
The Model-View-Controller (MVC) Pattern

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.

Organization of the MVC



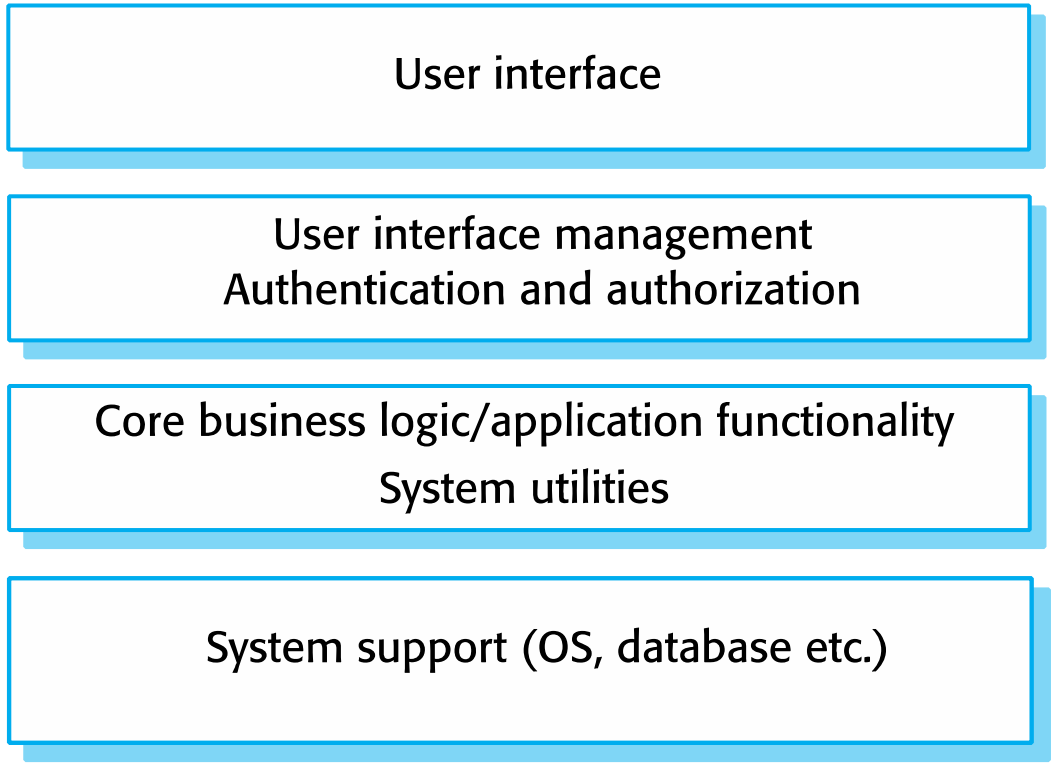
Example : Web Application Architecture



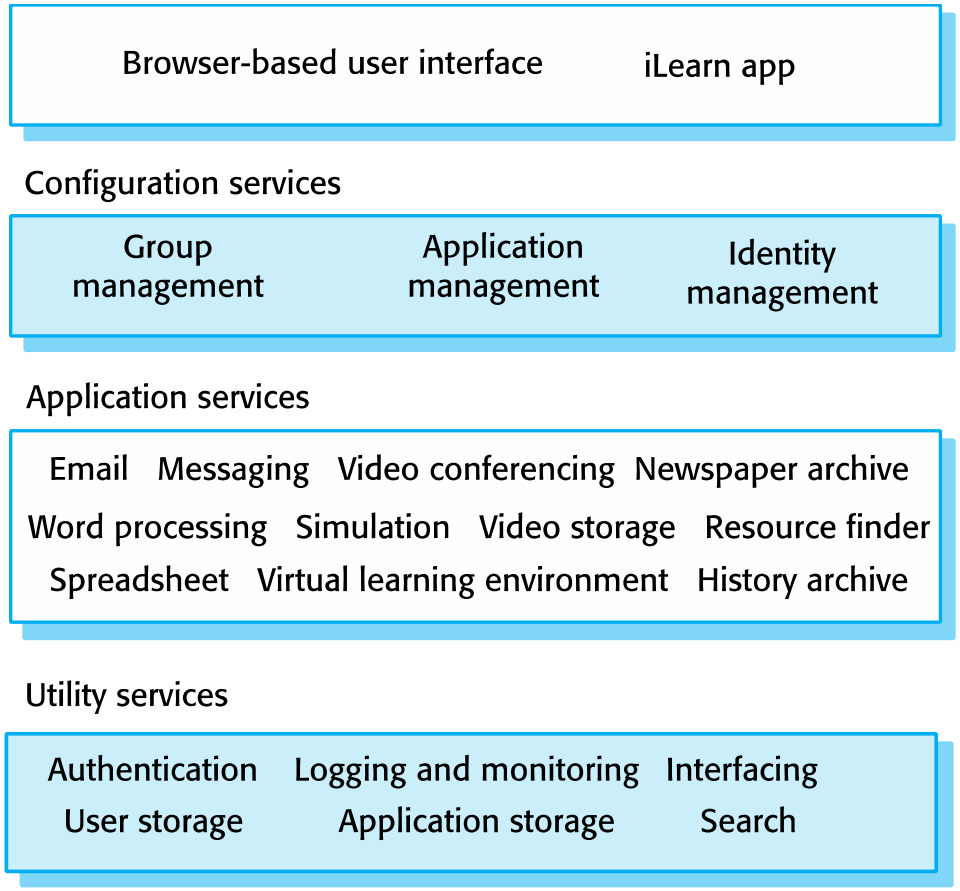
The Layered Architecture Pattern

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.

A Generic Layered Architecture



Example : The iLearn System

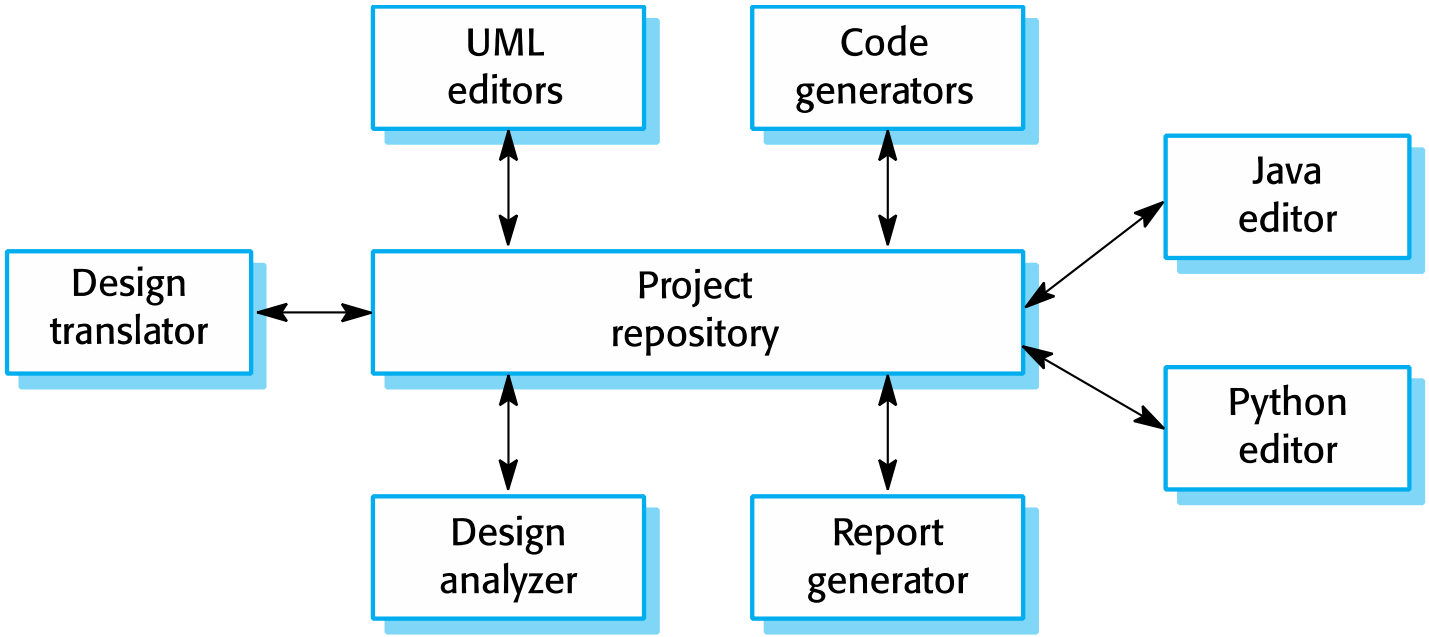


The Repository Pattern

- When large amounts of data are to be shared, the repository model of sharing is most commonly used as an efficient data sharing mechanism.
  - Shared data is held in a central database or repository and may be accessed by all sub-systems

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.

Example : IDE

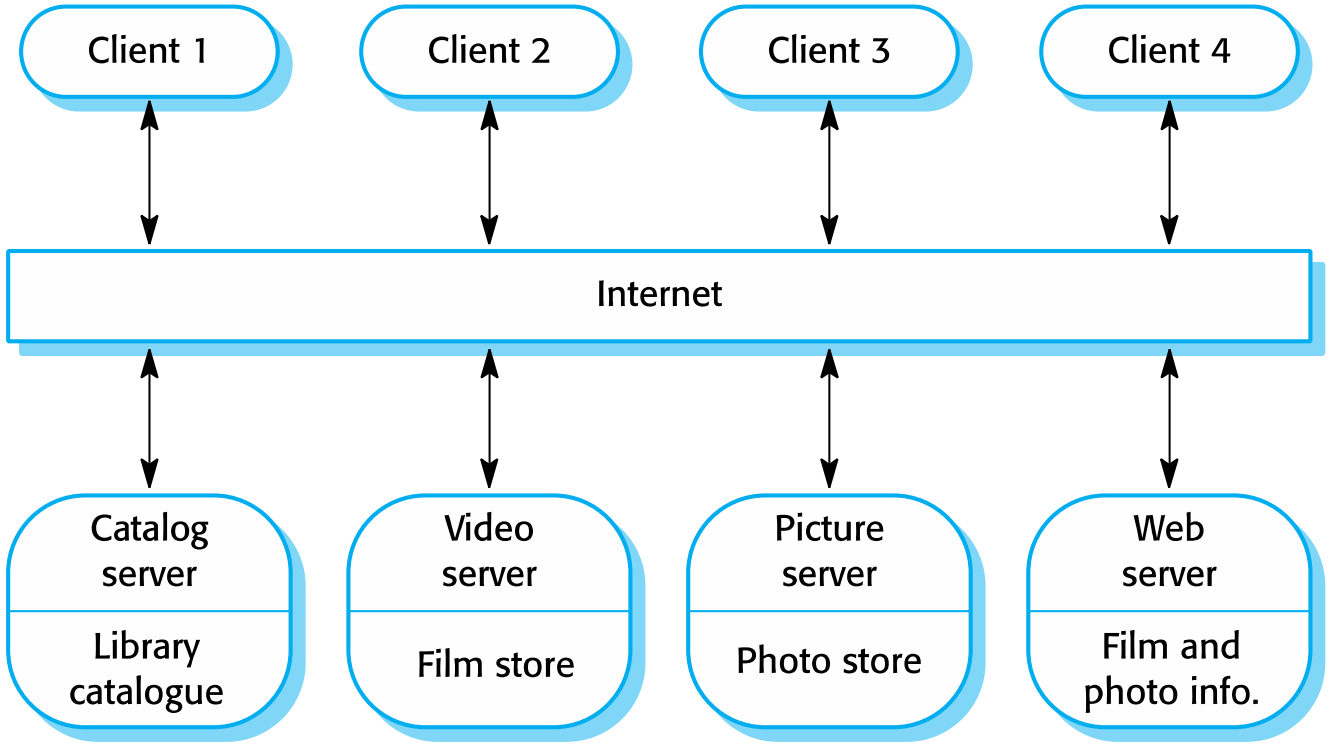

UML editors
Code generators
Java editor
Design translator
Project repository
Python editor
Design analyzer
Report generator

The Client-Server Pattern

- A distributed system model which showing how data and processing is distributed across a range of components
  - A set of stand-alone servers which provide specific services such as printing, data management, etc.
  - A set of clients which call on these services
  - Network which allows clients to access server

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.

Example : Film Library

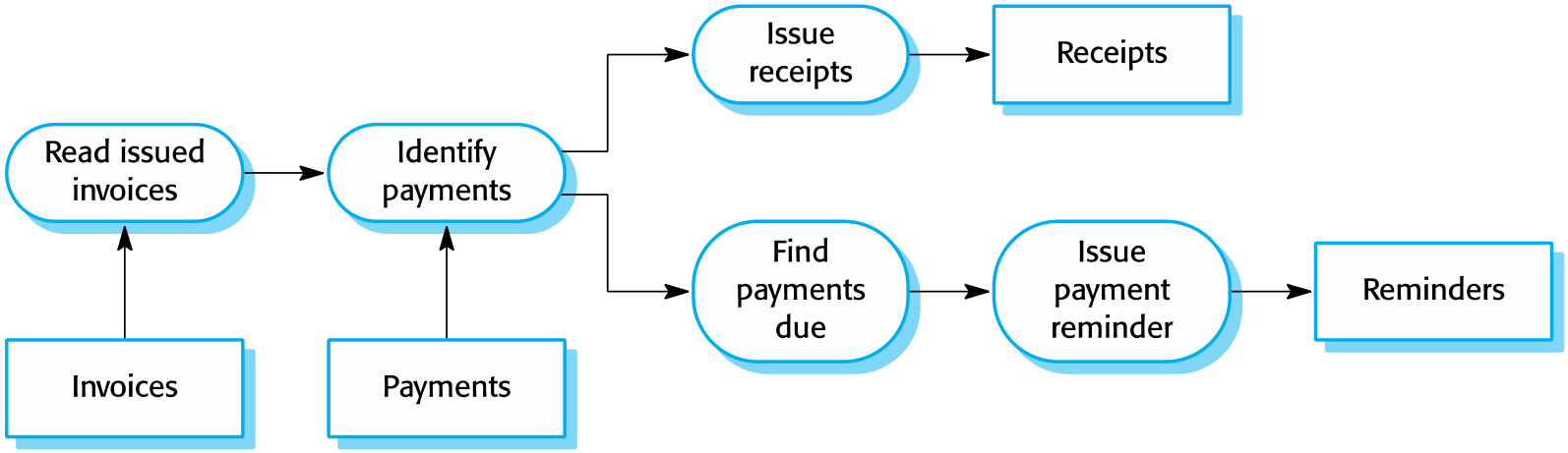


The Pipe and Filter Pattern

- Functional transformations processing inputs to produce outputs can be referred to as a pipe and filter model.
  - Variants are very common.
    - Example : When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
  - Not really suitable for interactive systems.

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.

Example : Payments System



Application Architectures

Application Architectures

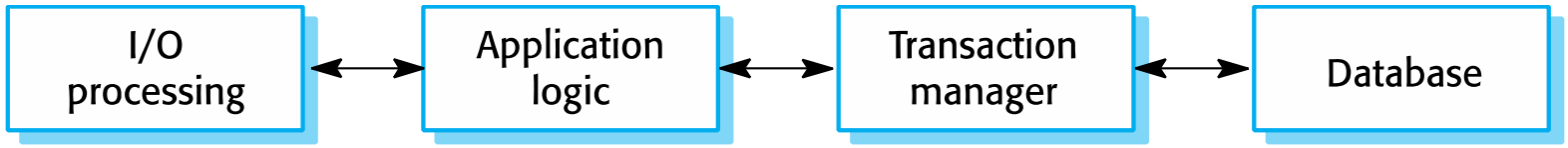
- **Application architecture**
  - An architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements
  - As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.

- Application Types
  - Data processing applications
    - Process data in batches without explicit user intervention during the processing
  - Transaction processing applications
    - 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

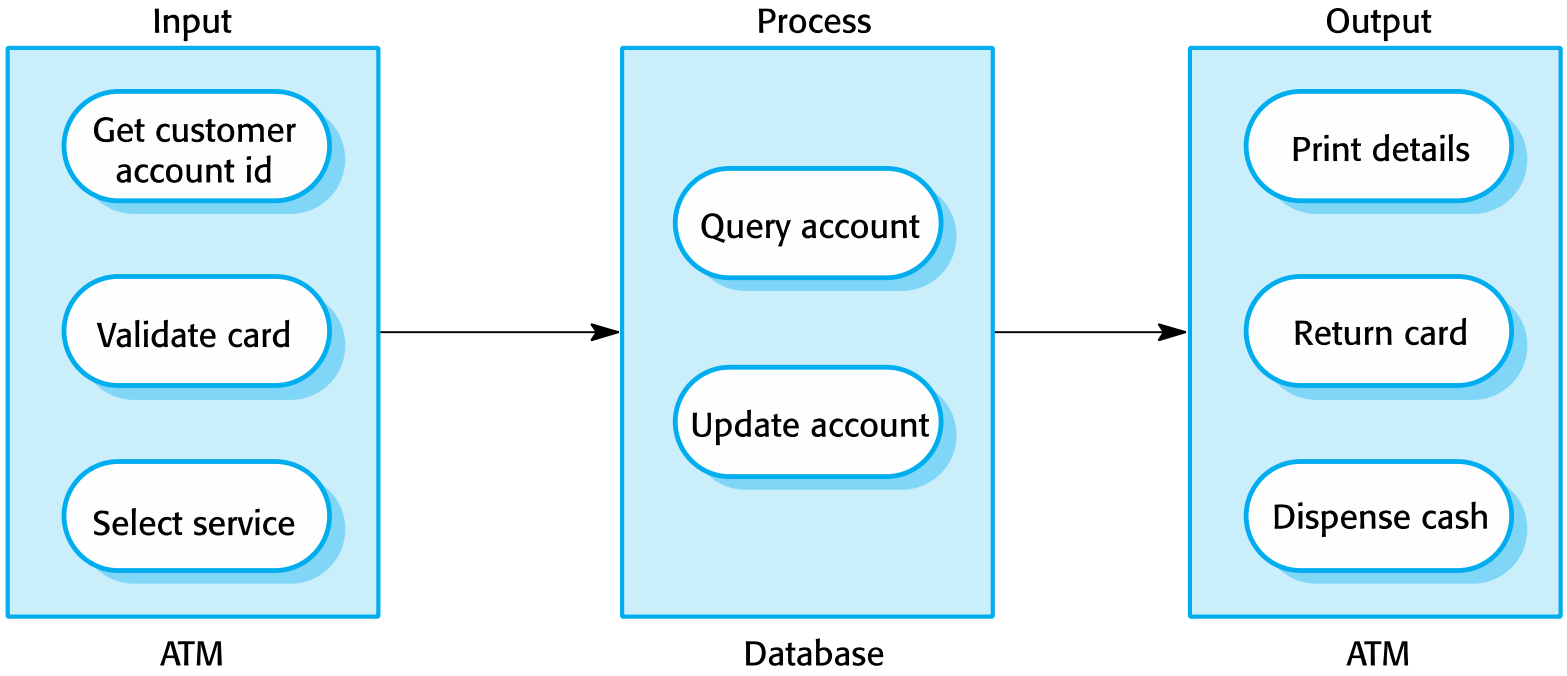
Transaction Processing Systems

- **Transaction Processing Systems** process user requests for information from a database, or process requests to update the database.
  - Users make asynchronous requests for service which are then processed by a transaction manager.
  - A transaction is any coherent sequence of operations that satisfies a goal.
  - Example :
    - Find the times of flights from London to Paris

  - A typical structure of the TPS applications :



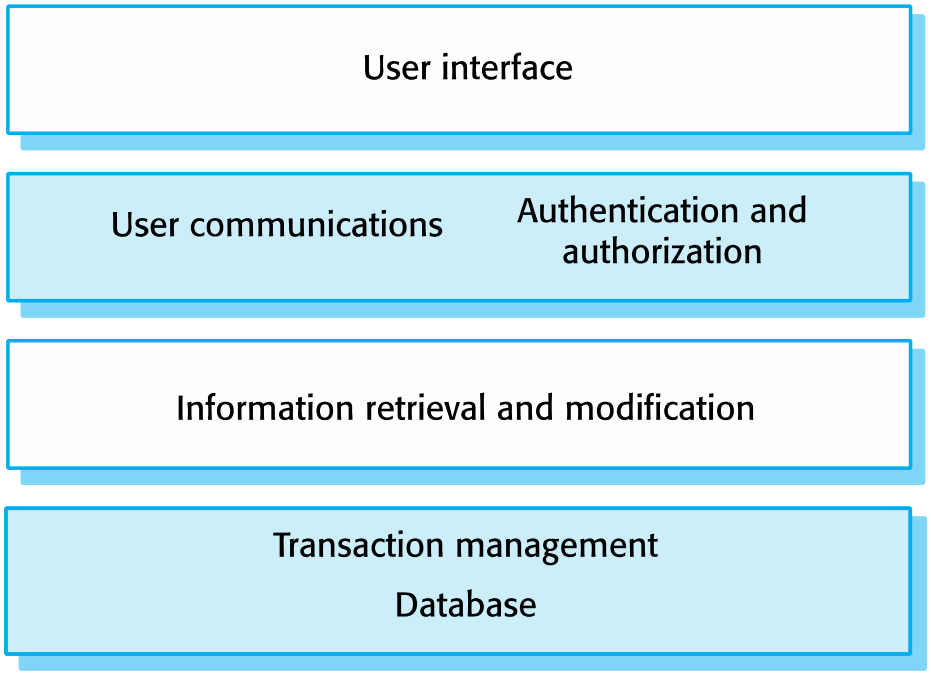
Example : an ATM System



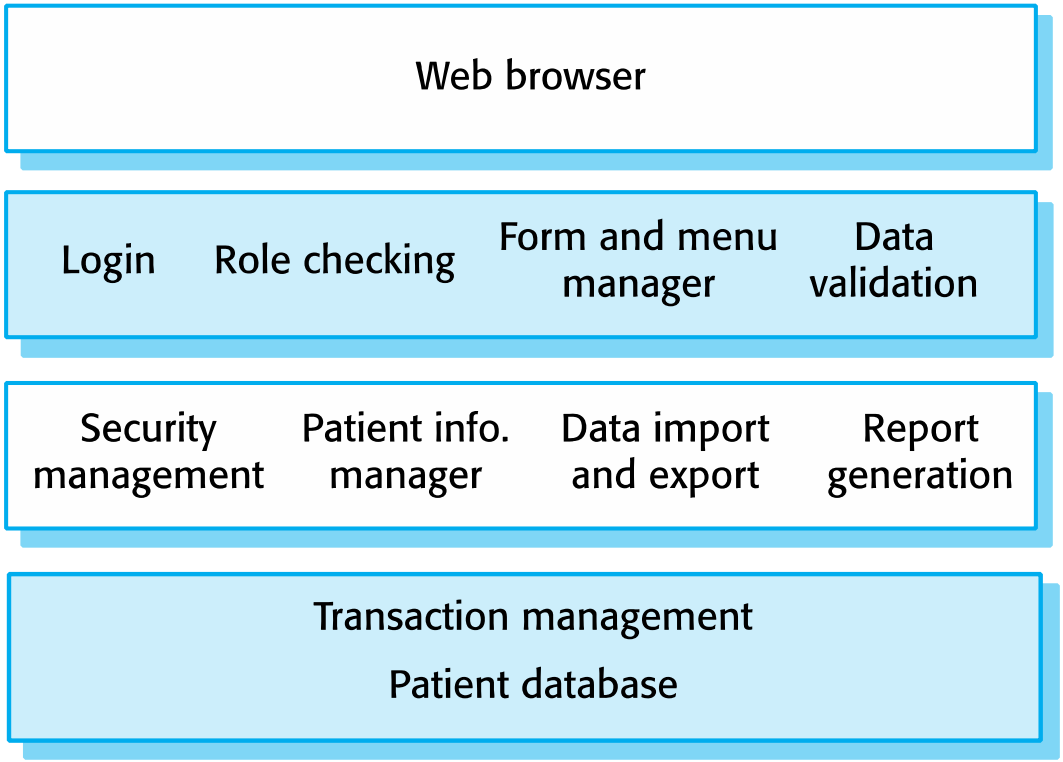
Information Systems Architecture

- Information systems have a generic architecture that can be organized as a layered architecture.
  - Also **transaction-based systems** as interaction with these systems generally involves database transactions.

- Layers include
  - User interface
  - User communications
  - Information retrieval
  - System database



Example : the Mentcare System



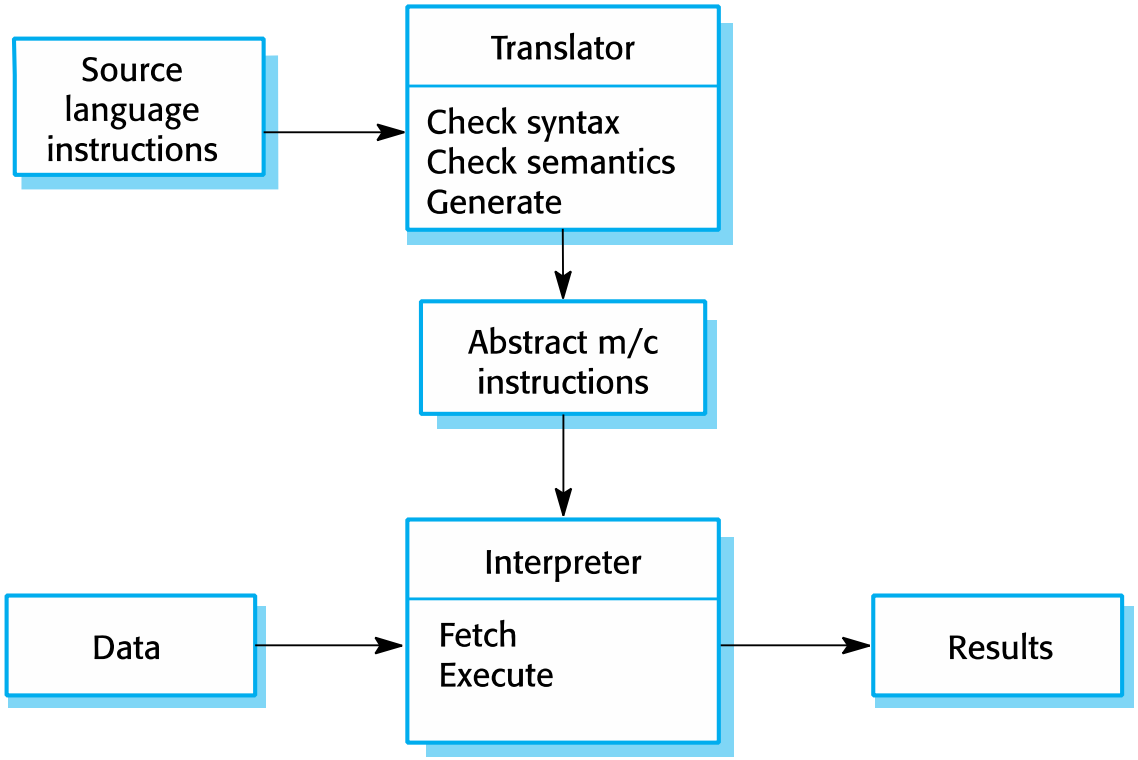
Web-Based Information Systems

- Web-based systems implement user interfaces using a web browser.
  - 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.
    - 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.

- Web-based information systems are often implemented as multi-tier client server/architectures.
  - Web server : Responsible for all user communications, with the user interface implemented using a web browser.
  - Application server : Responsible for implementing application-specific logic as well as information storage and retrieval requests.
  - Database server : Moves information to and from the database and handles transaction management.

Language Processing Systems

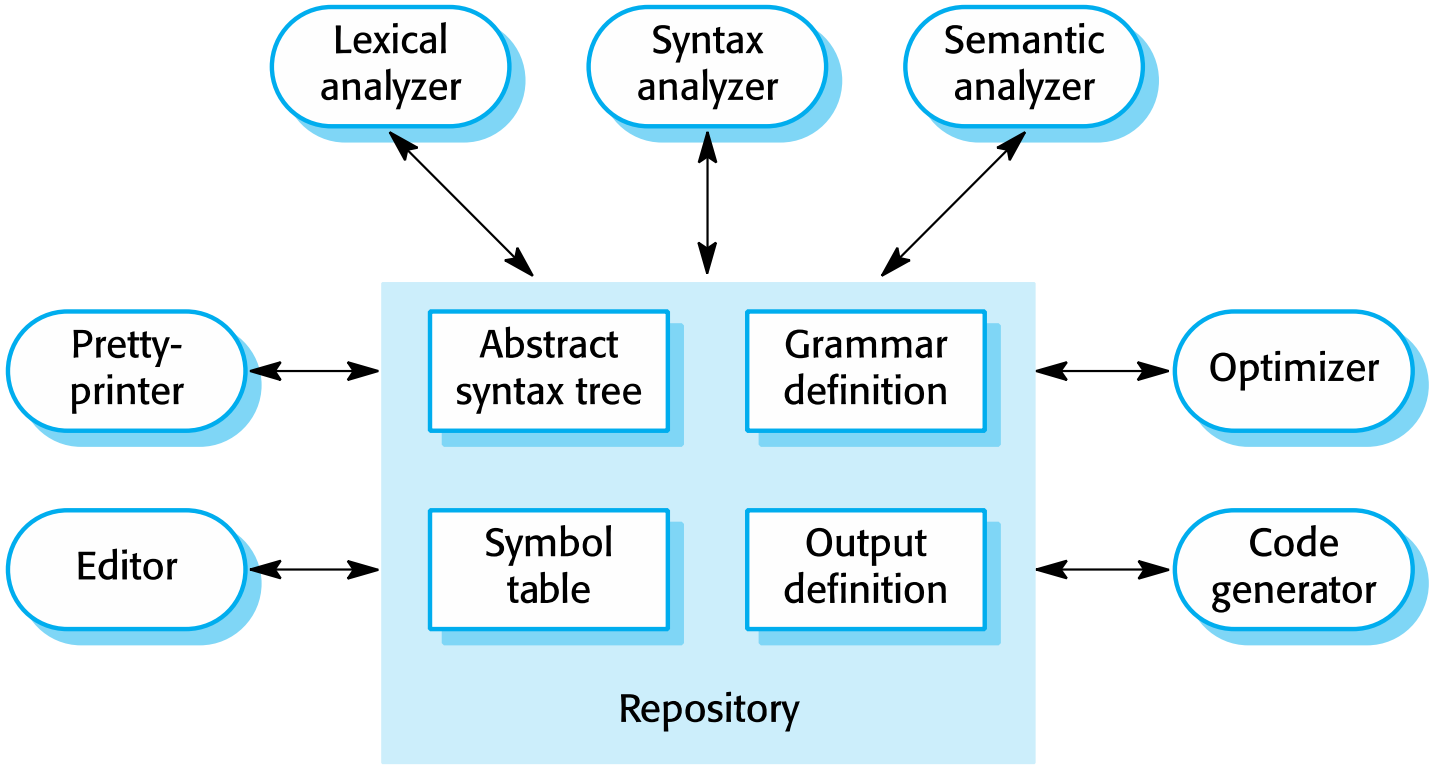
- **Language Processing Systems** accept a natural or artificial language as input and generate some other representation of that language.
  - May include an interpreter to act on the instructions in the language that is being processed.
  - Meta-case tools process tool descriptions, method rules, etc and generate tools.



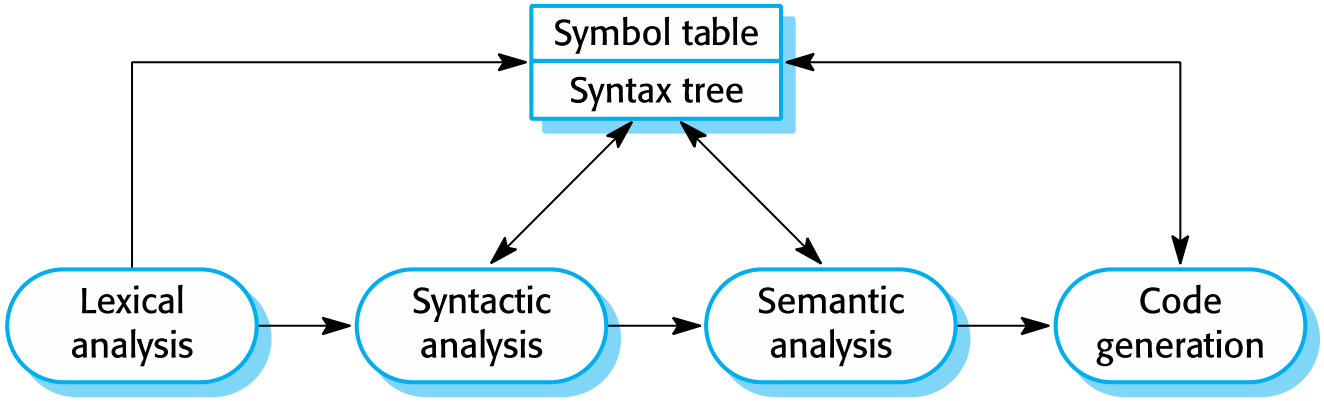
Compiler Components

- Compiler components for language processing systems
  - Lexical analyzer : Takes input language tokens and converts them to an internal form.
  - Symbol table : Holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
  - Syntax analyzer : Checks the syntax of the language being translated.
  - Syntax tree : An internal structure representing the program being compiled
  - Semantic analyzer : Uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
  - Code generator : 'walks' the syntax tree and generates abstract machine code.

A Repository Architecture for a Language Processing System



A Pipe and Filter Architecture for Compilers



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 views such 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.

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 7. Design and Implementation

Topics Covered

- Object-oriented design using UML
- Design patterns
- Implementation issues
- Open source development

Design and Implementation

- **Software design and implementation**
  - The stage at which an executable software system is developed

- Software design and implementation activities are often inter-leaved.
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
  - Implementation is the process of realizing the design as a program.

Build or Buy

- It is possible to buy **off-the-shelf systems (COTS)** that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.

- The design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.
  - Requires different ways to develop software.

Object-Oriented Design Using UML

An Object-Oriented Design Process

- **Structured object-oriented design processes** (such as **UP**)
  - Involve developing a number of different system models.
  - Require a lot of effort for development and maintenance of these models and, this may not be cost-effective for small systems.
  - However, for large systems developed by different groups, design models are an important communication mechanism.

- There are a variety of different object-oriented design processes.

- **Common activities** in all OO design processes
  1. Define the context and modes of use of the system
  2. Design the system architecture
  3. Identify the principal system objects
  4. Develop design models
  5. Specify object interfaces

1. System Context and Interactions

- Understanding the relationships between the software that is being designed and its external environment is essential for deciding
  - How to provide the required system functionality.
  - How to structure the system to communicate with its environment.

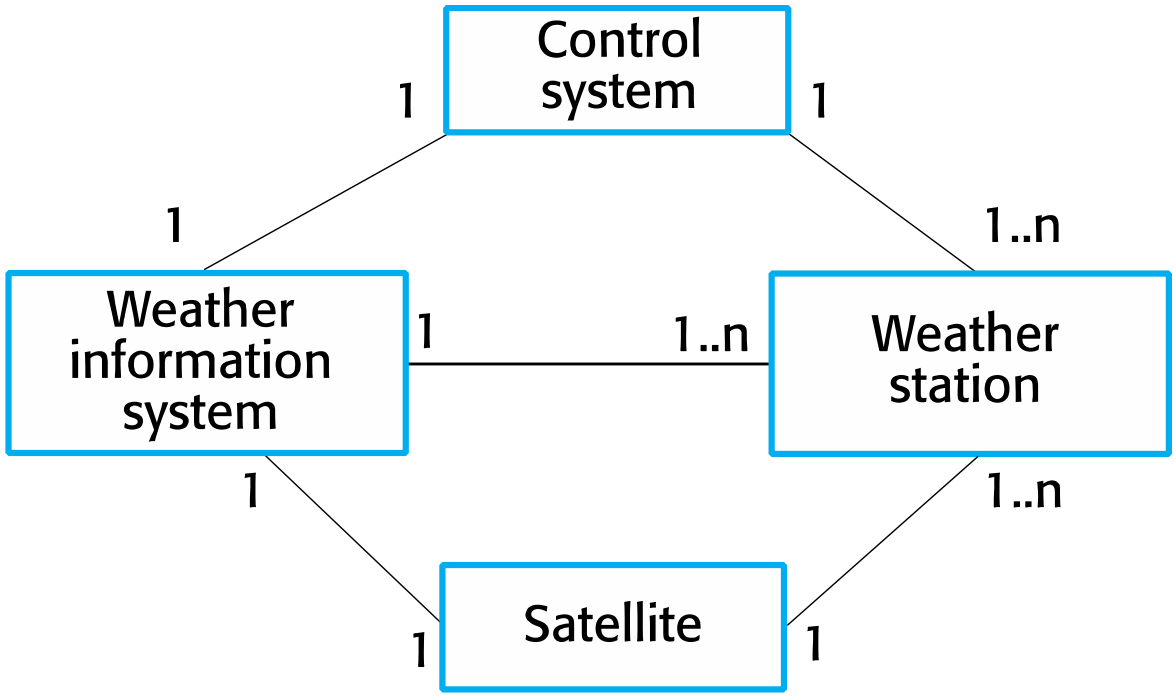
- Understanding of the context establish the boundaries of the system.
  - **Setting the system boundaries** helps you decide what features are implemented in the system being designed and what features are in other associated systems.

- **System context model**
  - Structural model : System context
  - Demonstrates the other systems in the environment of the system being developed

- **Interaction model**
  - Dynamic model : Use-case
  - Shows how the system interacts with its environment as it is used

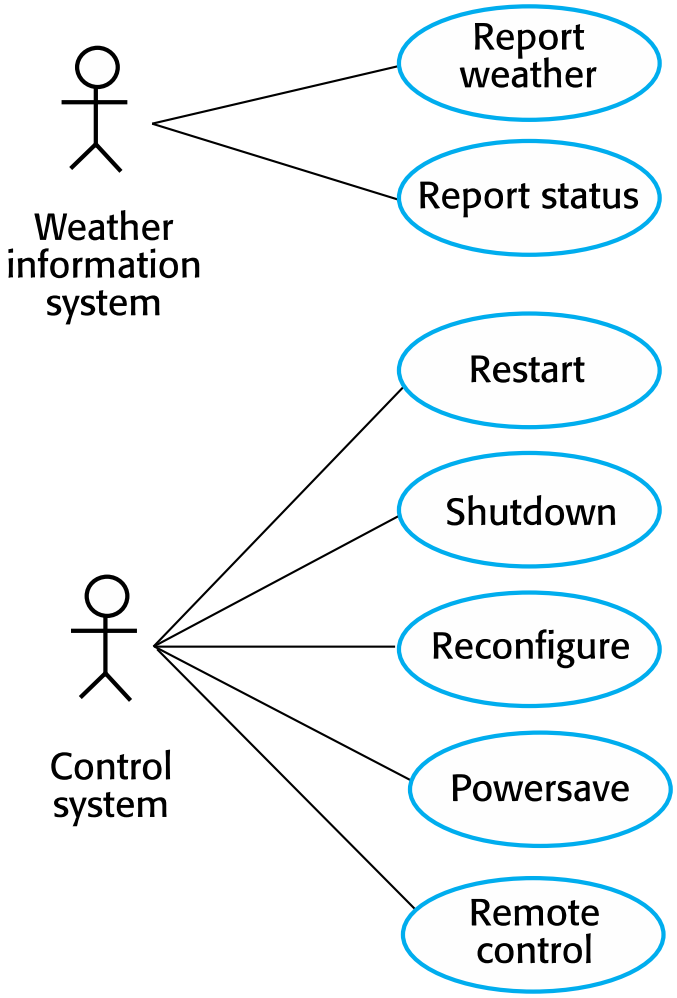
The Weather Station : System Context

- Structural model



The Weather Station : Use-Case

- Dynamic model



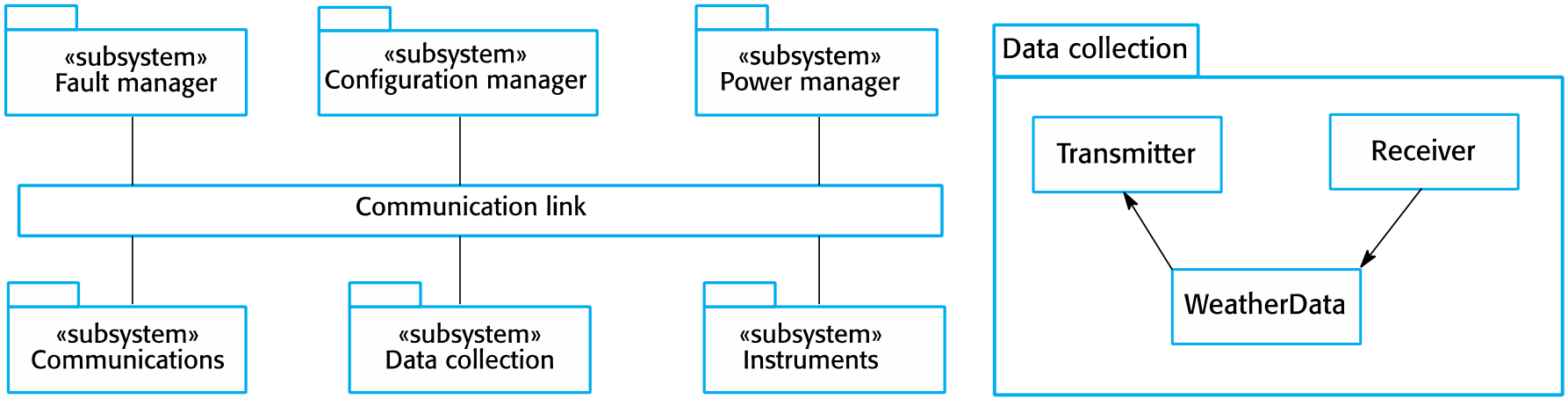
Use Case Description for “Report Weather”

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

2. Architectural Design

- **Identify the major components** that make up the system and their interactions.
  - Organize the components using an **architectural pattern** such as a layered or client-server model, if it needs.

  - Example : The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.



3. Object Class Identification

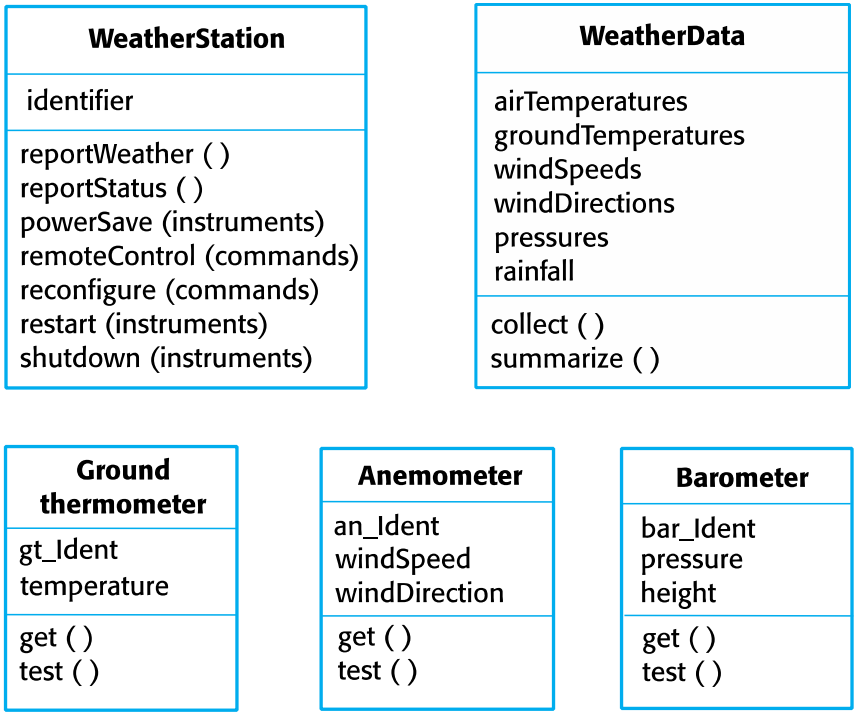
- **Identifying object classes** is a difficult part of object-oriented design.
  - There is no 'magic formula' for object identification.
  - It relies on the skill, experience and domain knowledge of system designers.

- **Object identification** is an iterative process.

- Approaches to object identification
  - Use a grammatical approach based on a natural language description of the system.
    - Base the identification on tangible things in the application domain
  - Use a behavioural approach.
    - Identify objects based on what participates in what behaviour
  - Use a scenario-based analysis.
    - The objects, attributes, and methods in each scenario are identified

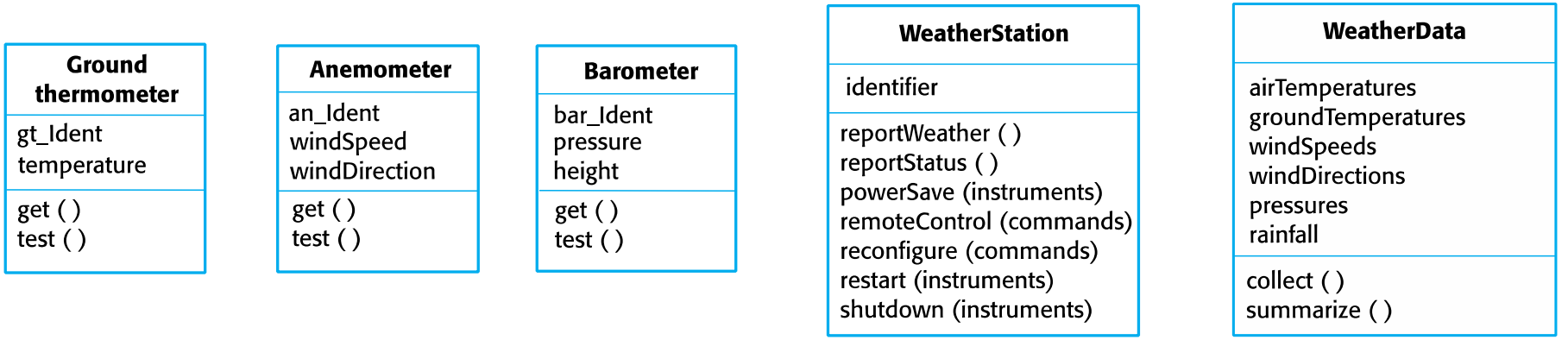
The Weather Station : Object Classes

- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Ground thermometer, Anemometer, Barometer
    - 'Hardware' objects related to the instruments in the system.
  - Weather station
    - The basic interface of the weather station to its environment.
    - It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.



The Weather Station : Object Classes

- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Ground thermometer, Anemometer, Barometer
    - 'Hardware' objects related to the instruments in the system.
  - Weather station
    - The basic interface of the weather station to its environment.
    - It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.



4. Design Models

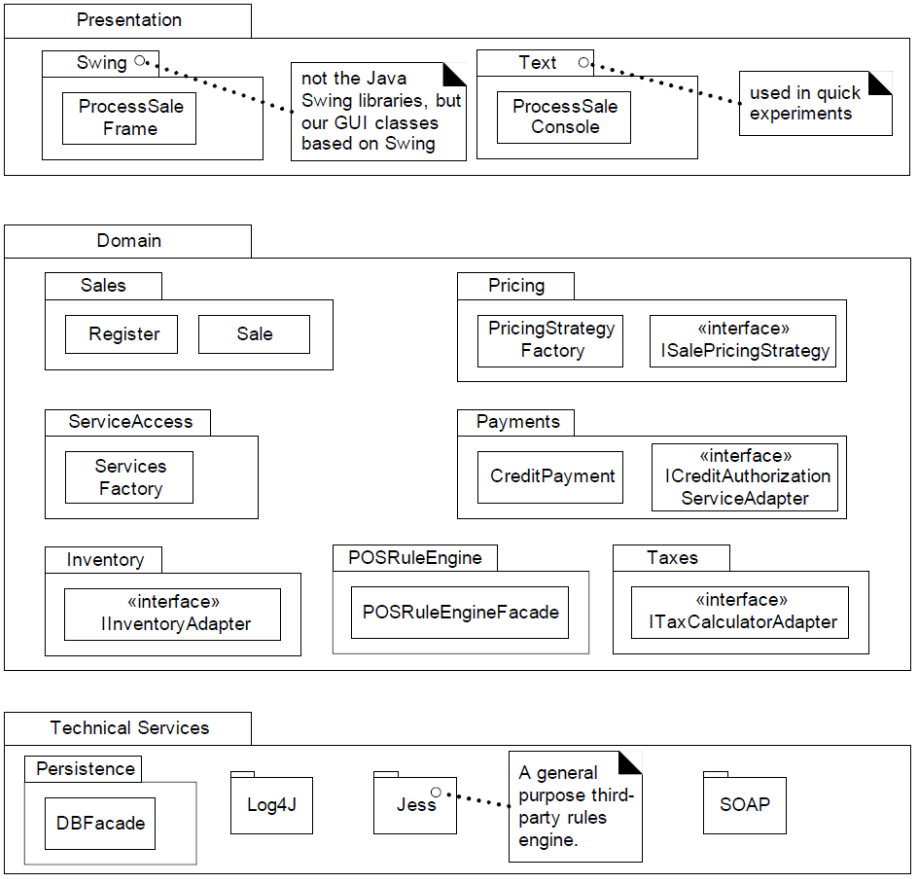
- **Design models** show the **objects and object classes** and **relationships** between these entities.

- Two types of design models
  - **Structural model**
    - Describe the static structure of the system in terms of object classes and relationships
    - Class diagram, Object diagram, Package diagram
  - **Dynamic model**
    - Describe the dynamic interactions between objects
    - Sequence diagram, Communication diagram, Statechart diagram

- Various design models
  - Subsystem model
  - Sequence model
  - State machine model
  - Use-case model
  - Aggregation model, Generalisation models
  - etc.

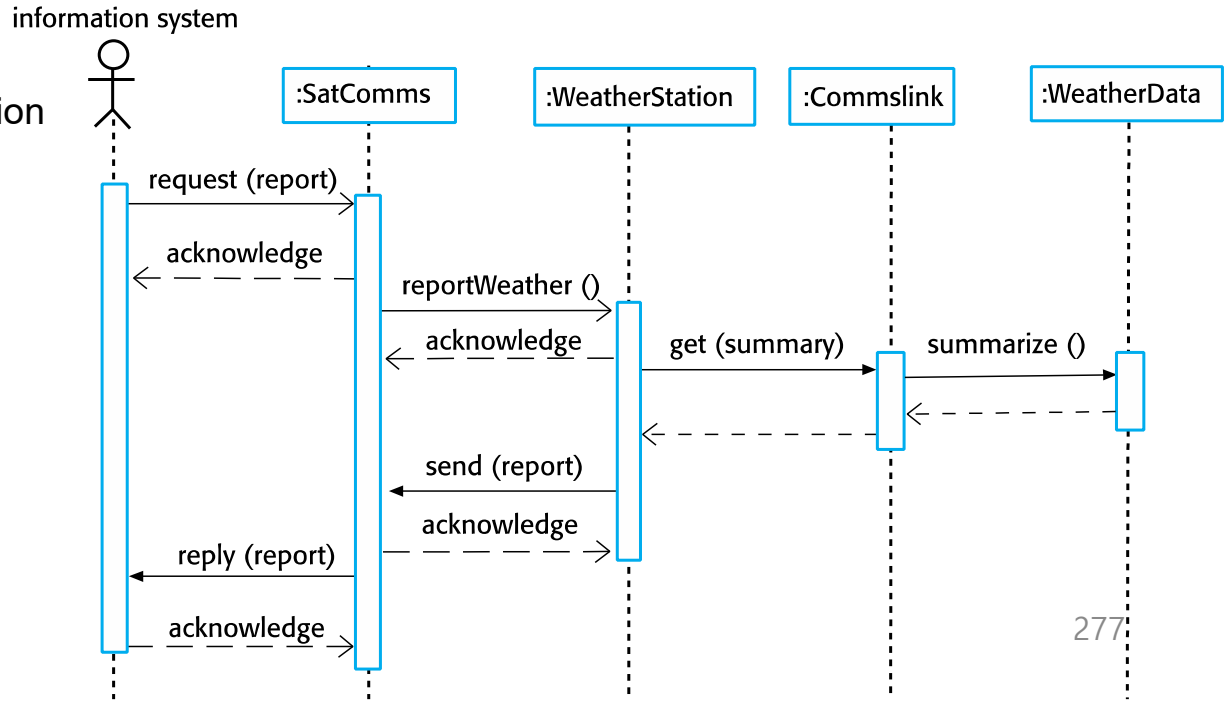
Subsystem Models

- **Subsystem Models** shows how the design is organized into logically related groups of objects.
  - Logical model
  - The UML package diagram are often used.
  - The actual organization of objects in the system may be different.



Sequence Models

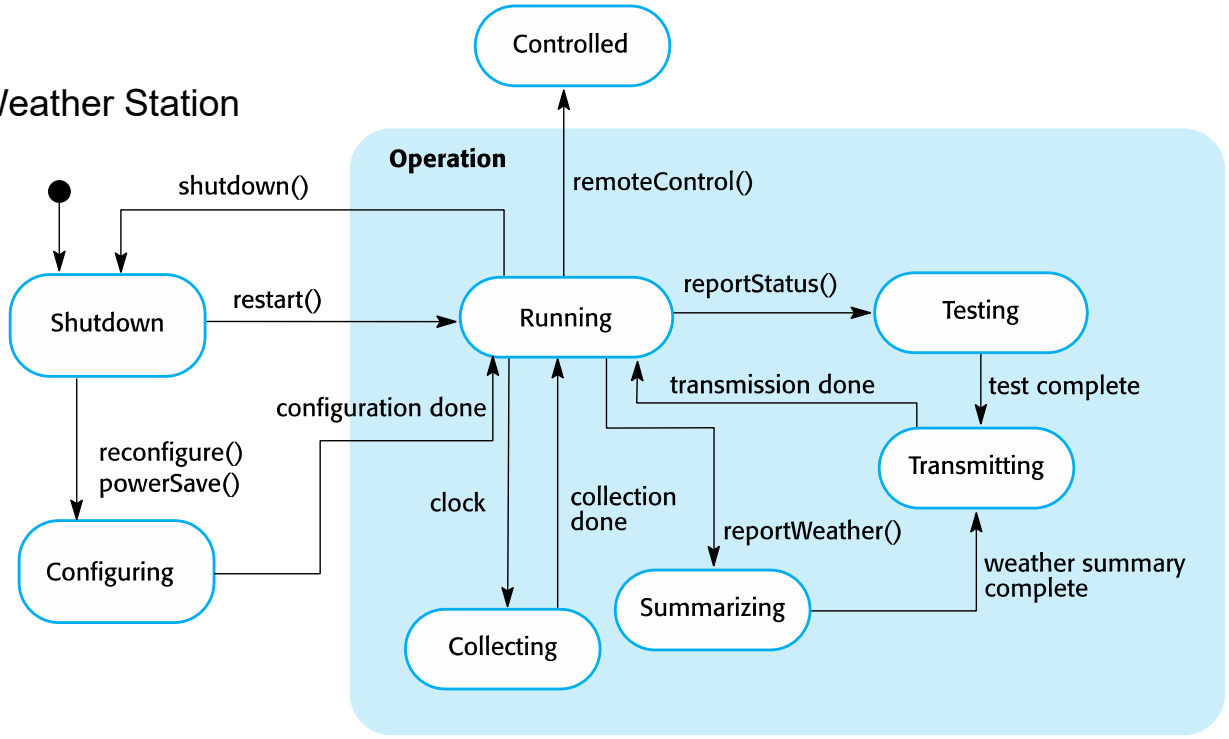
- **Sequence models** show the sequence of object interactions that take place
  - The UML Sequence diagrams are used.
    - Objects are arranged horizontally across the top.
    - Time is represented vertically so models are read top to bottom.
    - Interactions are represented by labelled arrows.
    - Different styles of arrow represent different types of interaction.
    - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.
  - Example:
    - SD for Data Collection



State Diagrams

- **State diagrams** are used to show how objects respond to different service requests and the state transitions triggered by these requests.
  - The UML Statecharts diagram is used.
  - State diagrams are useful high-level models of a system or an object's run-time behavior.
  - Not usually need a state diagram for all of the objects in the system.

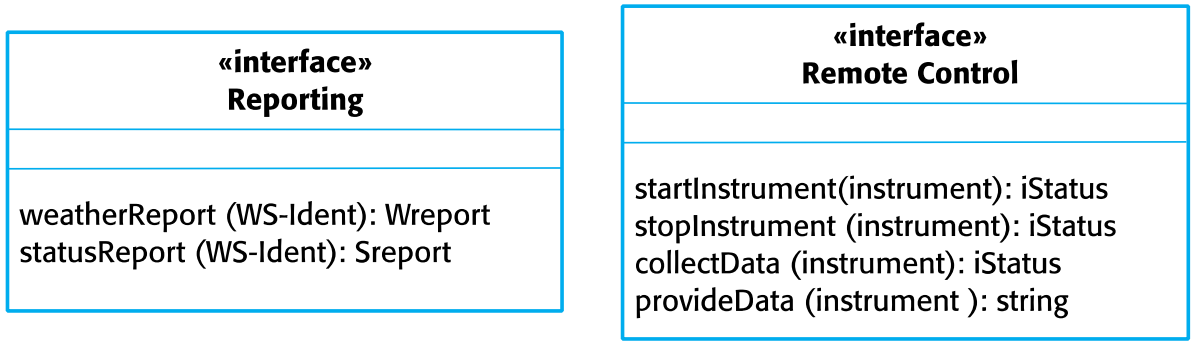
  - Example
    - State diagram for Weather Station



5. Interface Specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
  - Designers should avoid designing the interface representation, but should hide this in the object itself.
  - Objects may have several interfaces which are viewpoints on the methods provided.

- The UML uses **class diagrams** for interface specification.
  - Example
    - Interface specification (Class diagram) for Weather Station



Design Patterns

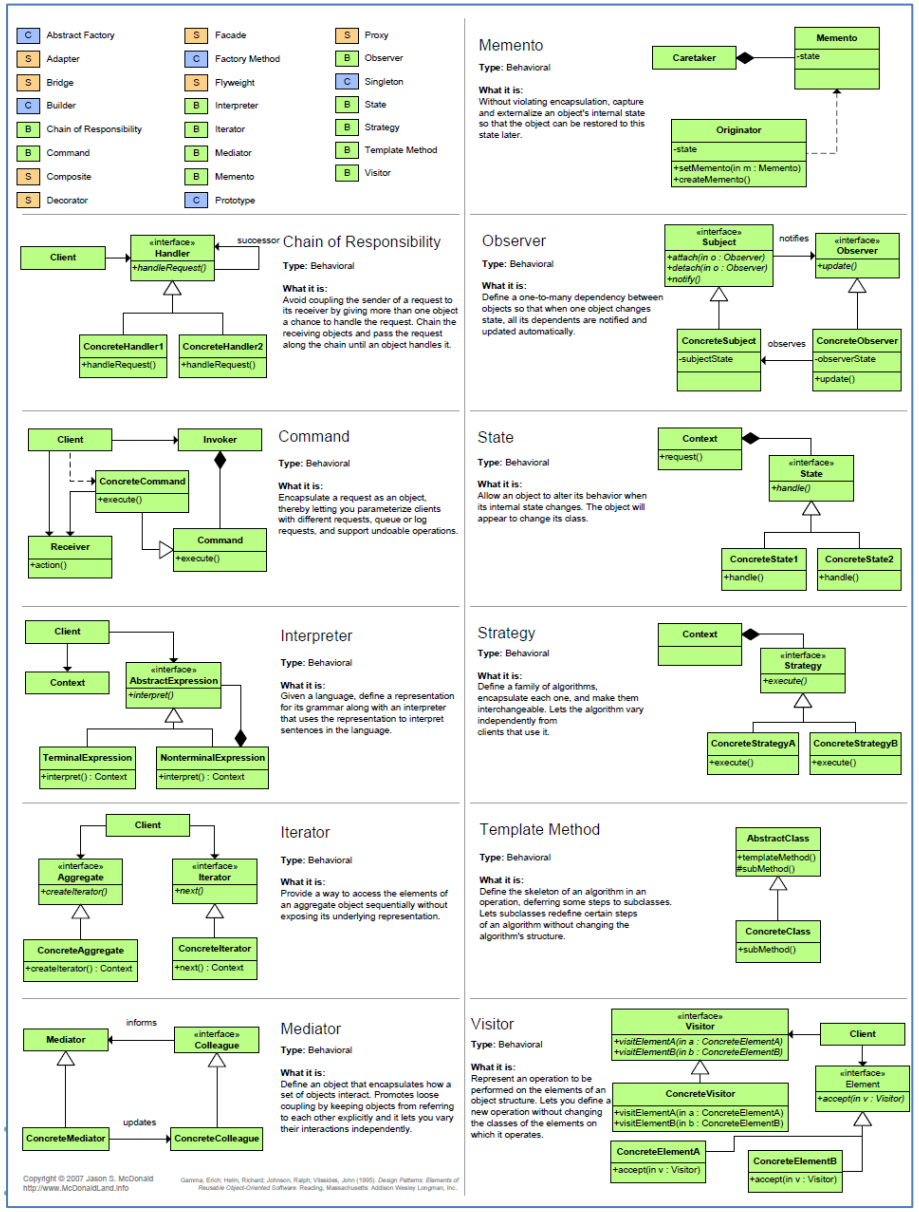
Design Patterns

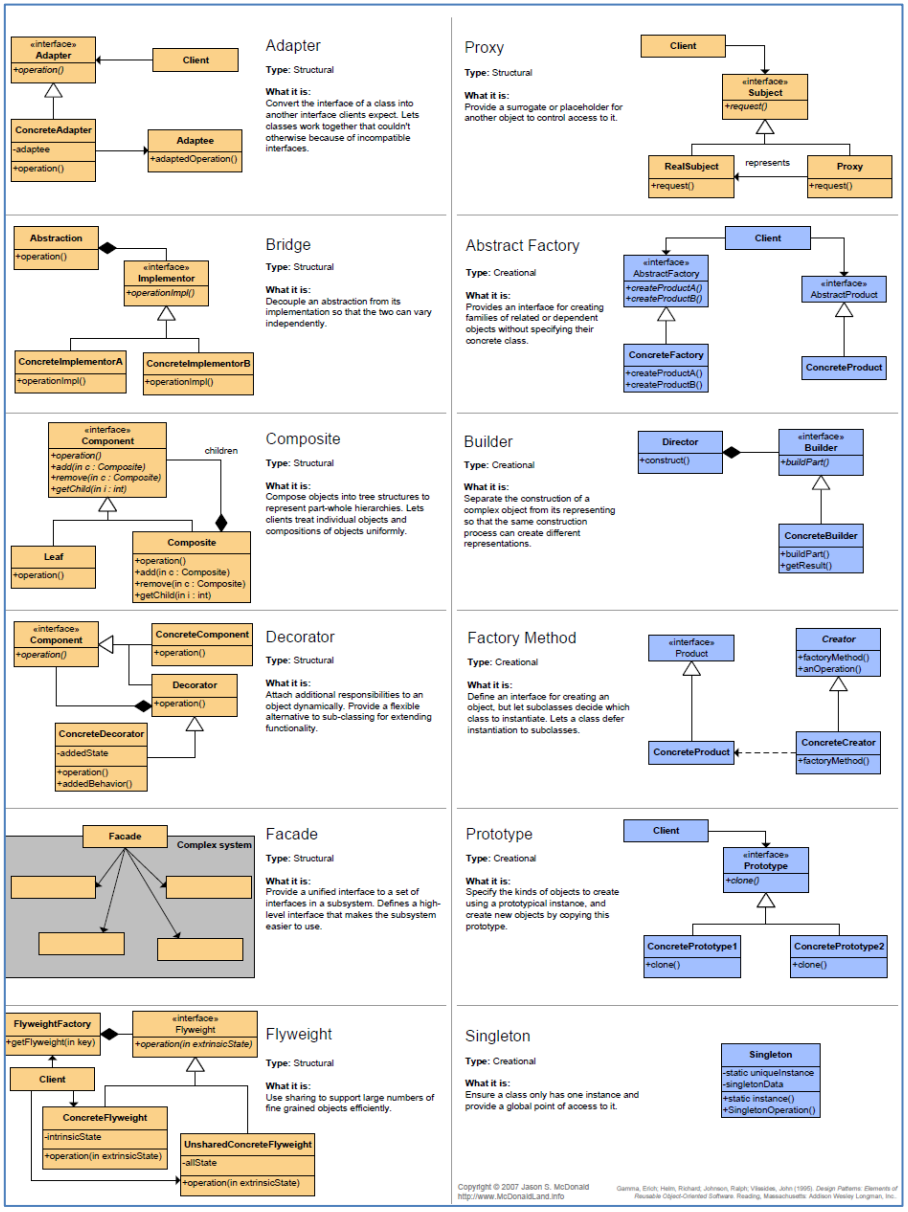
- **Design pattern** is a way to describe best practices, good designs, and capture experience in a way that it is possible for others to **reuse** this experience.
  - Descriptions of the problem and the essence of its solution
  - Sufficiently abstract to be reused in different settings.
  - Pattern descriptions usually make use of **object-oriented characteristics** such as inheritance and polymorphism.
  - **23 design patters of GoF** are widely used.

- Elements of patterns

Element	Description
Name :	A meaningful pattern identifier
Problem description	
Solution description	Not a concrete design, but a template for a design solution that can be instantiated in different ways.
Consequences:	The results and trade-offs of applying the pattern

23 Design Patterns of GoF

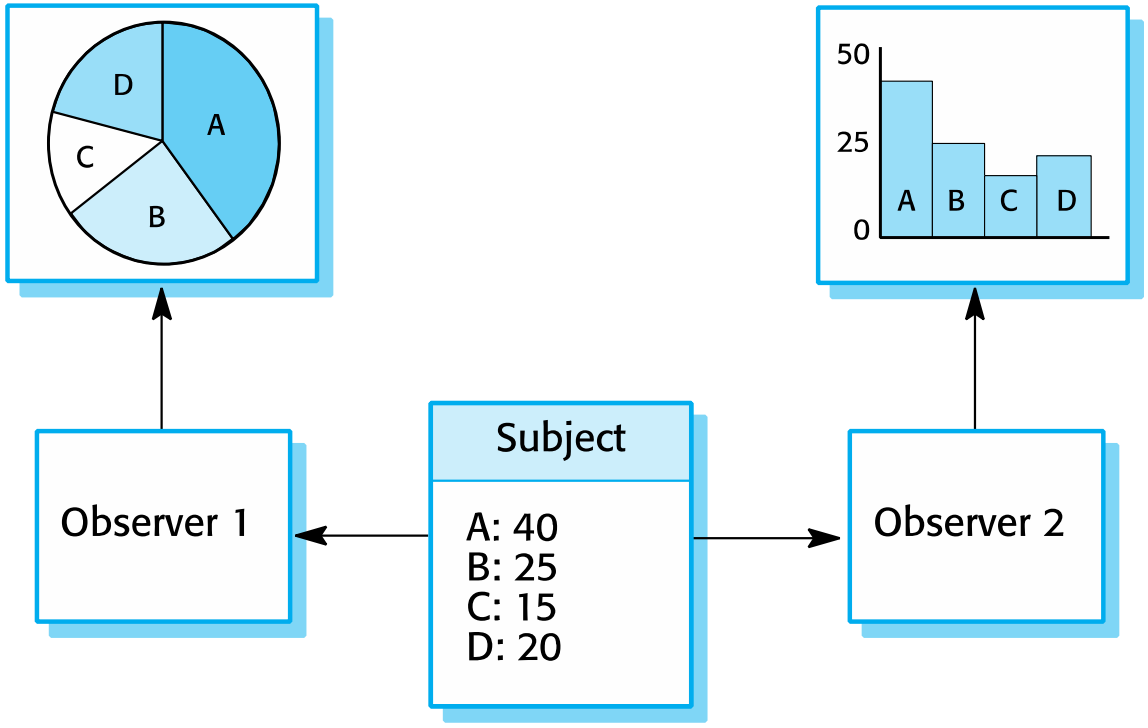




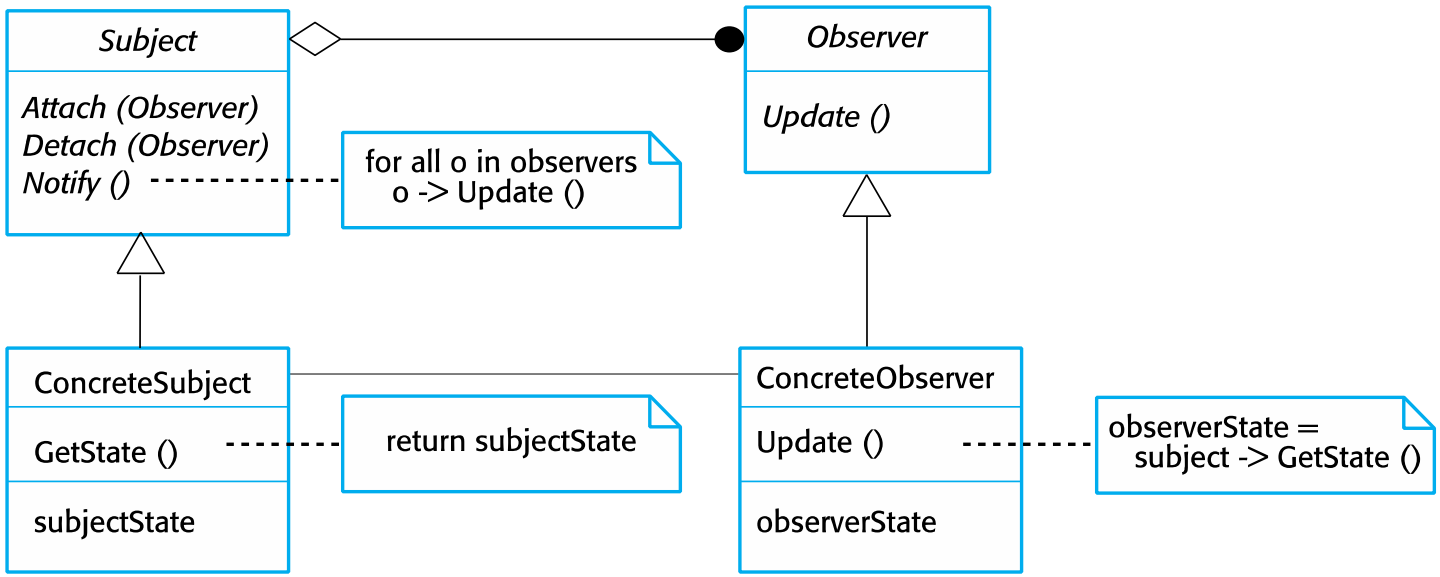
The Observer Pattern

Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Multiple Displays Using the Observer Pattern



A UML Model for the Observer Pattern



Design Problems

- To use design patterns, we need to recognize problems and associated patterns which can be applied to.
  - Example design patterns
    - Observer pattern : Tell several objects that the state of some other object has changed.
    - Façade pattern : Tidy up the interfaces to a number of related objects that have often been developed incrementally.
    - Iterator pattern : Provide a standard way of accessing elements in a collection, irrespective of how the collection is implemented.
    - Decorator pattern : Allow for the possibility of extending the functionality of an existing class at run-time.

Implementation Issues

Implementation Issues

- **Implementation issues** that are often not covered in programming
  - **Reuse**
    - Most modern software is constructed by reusing existing components or systems.
    - When you are developing software, you should make as much use as possible of existing code.
  - **Configuration management**
    - During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - **Host-target development**
    - Production software does not usually execute on the same computer as the software development environment.
    - Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Reuse

- A development approach based on the reuse of existing software emerges.
  - Until 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
    - The only significant reuse or software was the reuse of functions and objects in programming language libraries.

- Reuse levels
  - The abstraction level
    - We don't reuse software directly but use knowledge of successful abstractions in the design of our software. (Architectural and design patterns)
  - The object level
    - We directly reuse objects from a library rather than writing the code. (Programming language libraries)
  - The component level
    - Components are collections of objects and object classes that we reuse in application systems. (Component frameworks)
  - The system level
    - We reuse entire application systems. (COTS)

Reuse Costs

- The costs of **the time spent in looking for** software to reuse and **assessing** whether it meets your needs
  - Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.

- The costs of **adapting and configuring** the reusable software components or systems to reflect the requirements of the system that you are developing

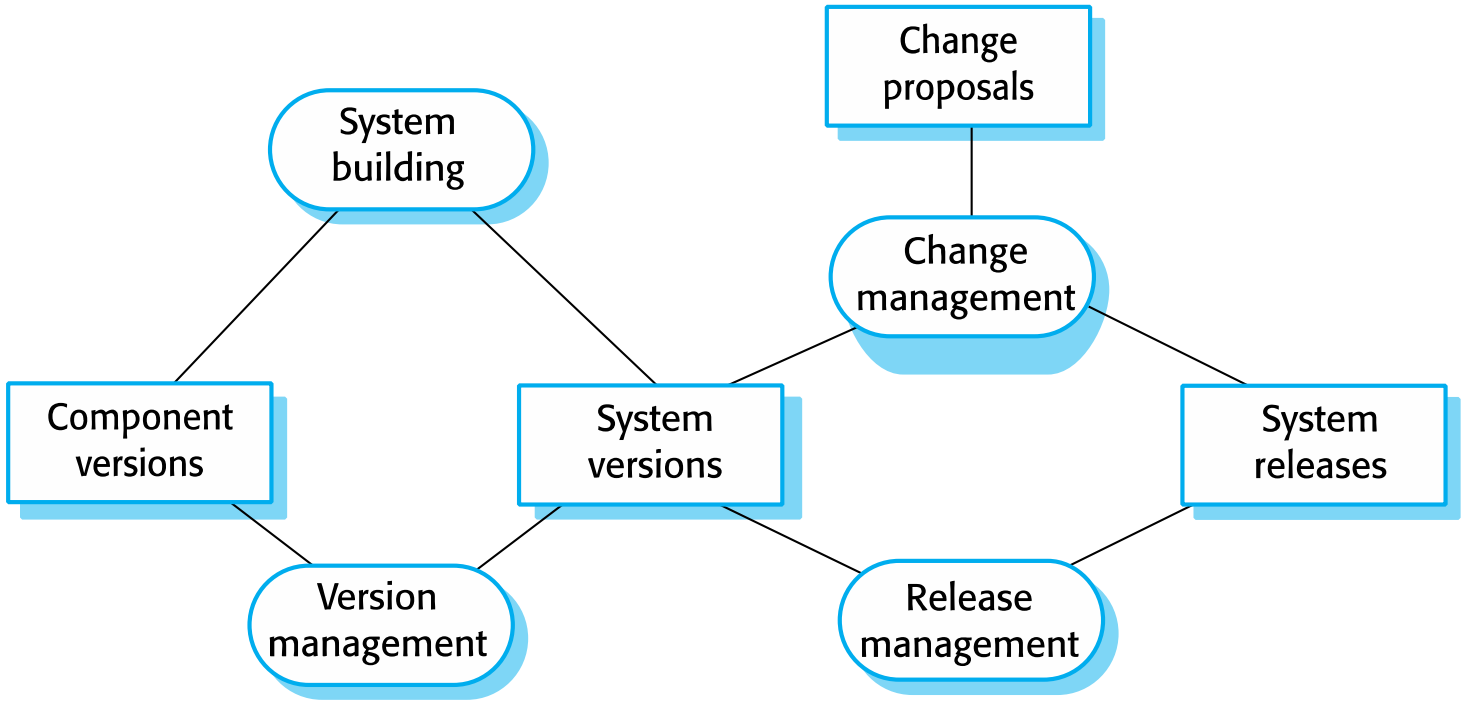
- The costs of **integrating reusable software elements** with each other and with the new code that you have developed

Configuration Management

- **Configuration management** is the general process of managing a changing software system.
  - To support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system. (Chapter 25)

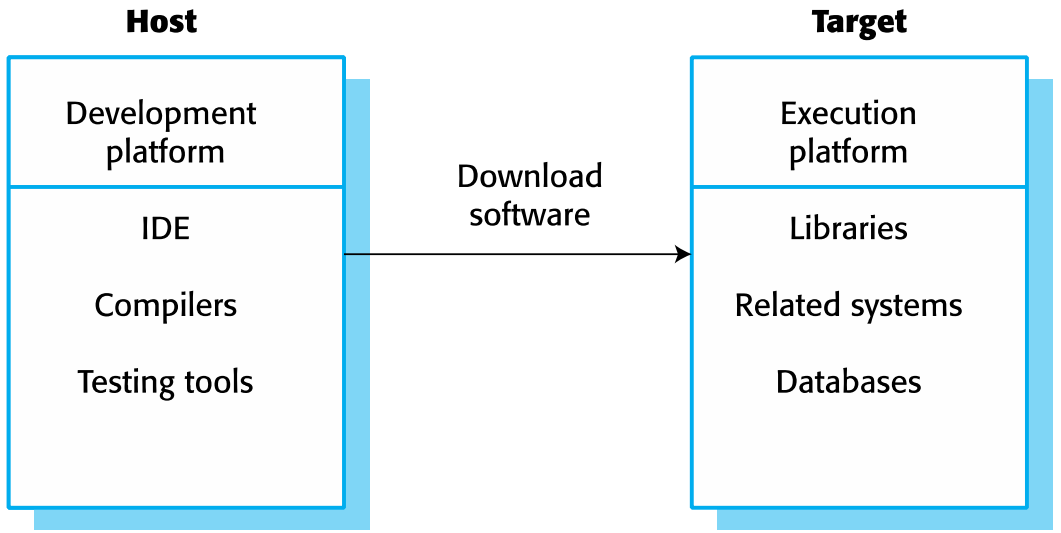
- Configuration Management Activities
  - **Version management**
    - Support is provided to keep track of the different versions of software components.
    - Version management systems include facilities to coordinate development by several programmers.
  - **System integration**
    - Support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
  - **Problem tracking**
    - Support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Interaction of Configuration Management Tools



Host-Target Development

- Most software is developed on a computer (the host) but runs on a separate machine (the target).
  - Development platform and Execution platform
    - A platform is more than just hardware.
    - Includes the installed operating system and other supporting software such as database management systems or, interactive development(environments for development platforms).
  - Development platform usually has different installed software than execution platform.
    - May have different architectures.
  - Host-Target development



Tools for Host-Target Development

- **Tools for development platforms**
  - Integrated compiler and syntax-directed editing system : create, edit and compile code
  - Language debugging system
  - Graphical editing tools : such as UML tools
  - Testing tools : such as JUnit that can automatically run a set of tests on a new version of a program.
  - Project support tools : organize codes for different development projects

- **IDE (Integrated Development Environments)**
  - A set of software tools that supports different aspects of software development, within some common framework and user interface
  - IDEs are created to support development in a specific programming language such as Java.

Deployment Factors

- Implementation should be concerned with deployment factors.
  - If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
  - High availability systems may require components to be deployed on more than one platform. In the event of platform failure, an alternative implementation of the component is available.
  - If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

Open Source Development

Open Source Development

- **Open source development** is a software development approach in which the source code of a software system is published and volunteers are invited to participate in the development process.
  - Rooted on the Free Software Foundation (www.fsf.org)
    - Advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
  - Uses the Internet to recruit a much larger population of volunteer developers.
    - Many of them are also users of the code.

- Popular examples of open source systems
  - The Linux operating system
  - Java
  - The Apache web server
  - The mySQL database management system

Open Source Issues

- **Questions** on open sources :
  - "Should the product that is being developed make use of open source components?"
  - "Should we use an open source approach for the software's development?"

- **Business** with opens source
  - More and more product companies are using an open source approach to development.
  - Their business model is not reliant on selling a software product but on selling support for that product.
  - They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Open Source Licensing

- Fundamental principle of open source
  - **“Source code should be freely available.”**

- It does not mean that anyone can do as they wish with that code.
  - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
  - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
  - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

License Models

- The GNU General Public License (**GPL**).
  - So-called ‘reciprocal’ license
  - If you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU Lesser General Public License (**LGPL**)
  - A variant of the GPL license
  - You can write components that link to open source code without having to publish the source of these components.
- The Berkley Standard Distribution (**BSD**) License
  - Non-reciprocal license
  - You are not obliged to re-publish any changes or modifications made to open source code.
  - You can include the code in proprietary systems that are sold.

License Management

- Establish a system for maintaining information about open-source components that are downloaded and used.
- Be aware of the different types of licenses and understand how a component is licensed before it is used.
- Be aware of evolution pathways for components.
- Educate people about open source.
- Have auditing systems in place.
- Participate in the open source community.

Key Points

- Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

Key Points

- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

Chapter 8. Software Testing

Topics Covered

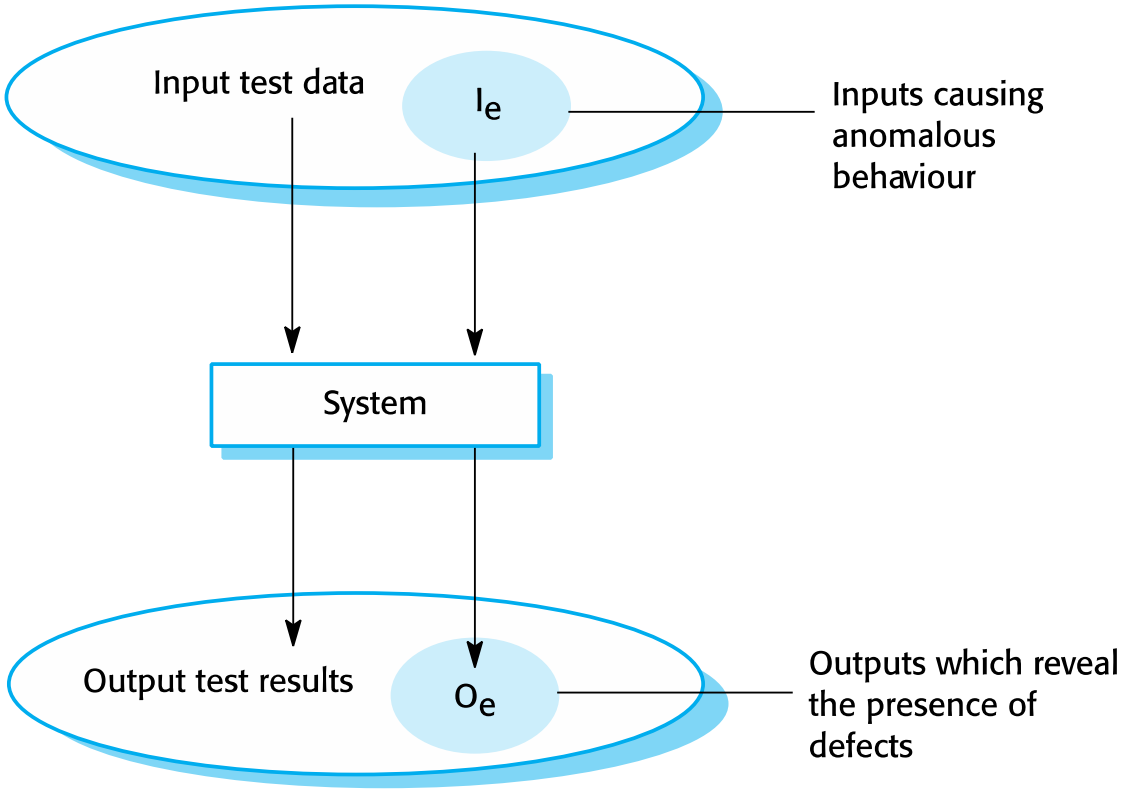
- Development testing
- Test-driven development
- Release testing
- User testing

Program Testing

- **Testing** executes a program using artificial data.
  - Check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
  - Can reveal the presence of errors, but NOT their absence.
  - A part of **V&V (Verification and Validation)** process.

- Testing goals
  - **Validation testing**
    - To demonstrate to the developer and the customer that the software meets its requirements.
    - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
    - A successful test shows that the system operates as intended.
  - **Defect (Verification) testing**
    - To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
    - The test cases are designed to expose defects.
    - The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

An Input-Output Model of Program Testing



Verification vs. Validation

- **Verification**: "*Are we building the product right*".
  - The software should conform to its specification.

- **Validation**: "*Are we building the right product*".
  - The software should do what the user really requires.

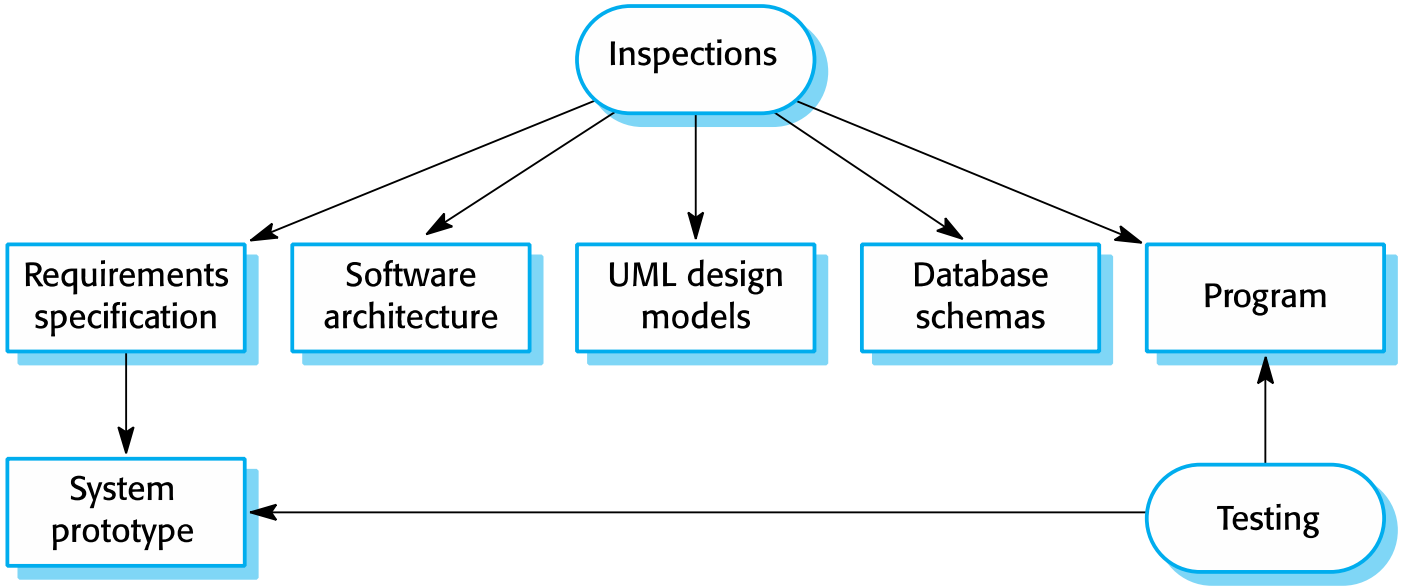
V&V Confidence

- Aim of V&V is to establish confidence that the system is '**fit for purpose**'.

- **V&V confidence** depends on
  - Software purpose
    - The level of confidence depends on how critical the software is to an organisation.
  - User expectations
    - Users may have low expectations of certain kinds of software.
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program.

Inspections and Testing

- Two popular techniques for software V&V
  - **Inspections** : Concerned with analysis of the static system representation to discover problems
    - Static verification
    - May be supplement by tool-based document and code analysis
  - **Testing** : Concerned with exercising and observing product behaviour
    - Dynamic verification
    - The system is executed with test data and its operational behaviour is observed.



Software Inspections

- People examine the source representation to discover anomalies and defects.
  - Not require execution of a system, so may be used before implementation.
  - May be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
  - An effective technique for discovering program errors

- **Advantages** of Inspections
  - During testing, errors can mask (hide) other errors.
    - Because inspection is a static process, you don't have to be concerned with interactions between errors.
  - Incomplete versions of a system can be inspected without additional costs.
    - If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
  - Inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

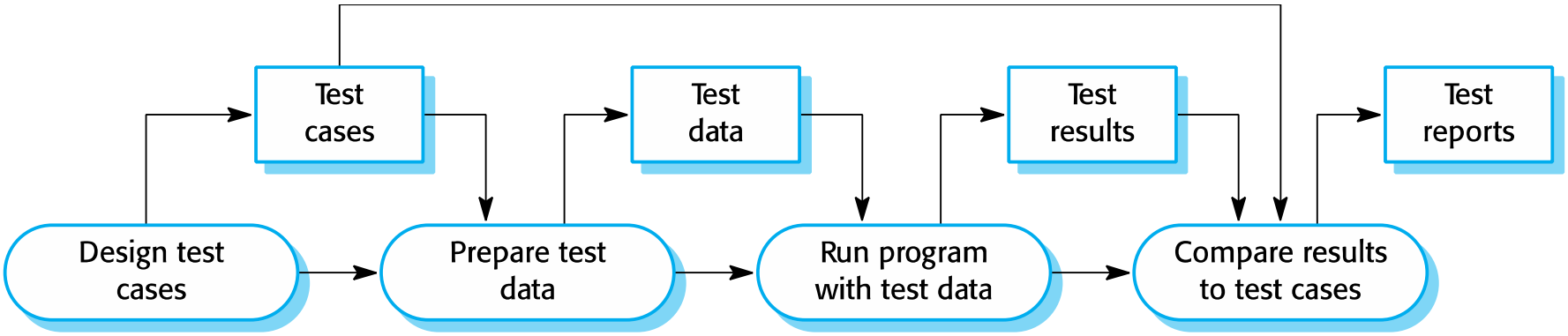
Inspections vs. Testing

- Inspections and testing are **complementary** and not opposing verification techniques.
  - Both should be used during the V&V process.
  - Inspections can check conformance with specifications and standards, but not conformance with the customer's real requirements.
  - Inspections cannot check some non-functional characteristics such as performance, usability, etc.

Software Testing

- Software testing stages
  - **Development testing**
    - The system is tested during development to discover bugs and defects.
  - **Release testing**
    - A separate testing team test a complete version of the system, before it is released to users.
  - **User testing**
    - Users or potential users of a system test the system in their own environment.

- Software testing process



Development Testing

Development Testing

- **Development testing** includes all testing activities that are carried out by the team developing the system.
  - **Unit testing**
    - Individual program units or object classes are tested.
    - Should focus on testing the functionality of objects or methods.
  - **Component testing**
    - Several individual units are integrated to create composite components.
    - Should focus on testing component interfaces.
  - **System testing**
    - Some or all components in a system are integrated and the system is tested as a whole.
    - Should focus on testing component interactions.

Unit Testing

- **Unit testing** is the process of testing individual components in isolation.
  - Defect testing

- Units may be:
  - **Individual functions or methods** within an object
  - **Object classes** with several attributes and methods
  - **Composite components** with defined interfaces used to access their functionality.

Unit Testing : Object Class

- Complete test coverage of a **class** involves
  - Testing all operations associated with an object.
  - Setting and interrogating all object attributes.
  - Exercising the object in all possible states.

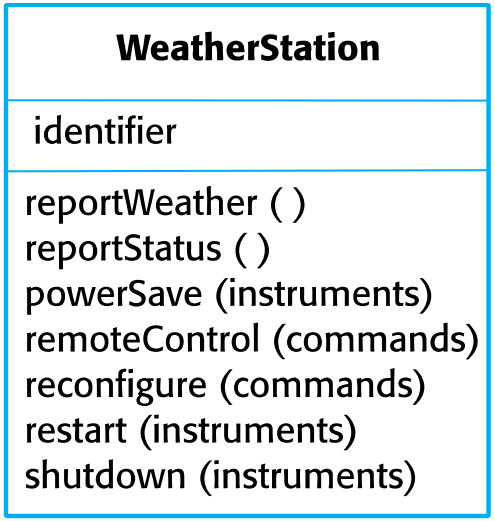
- **Inheritance** makes it more difficult to design object class tests.
  - Since the information to be tested is not localized.

The Weather Station : Object Unit Testing

- Need to define test cases for all operations.
  - Such as reportWeather, calibrate, test, startup and shutdown

- **State model** can identify sequences of state transitions to be tested and the event sequences to cause these transitions.

- For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running



Automated Testing

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.

- **Unit testing frameworks** provide generic test classes that you extend to create specific test cases.
  - Can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.
  - JUnit, xUnit, etc.

- Components (stages) of automated testing frameworks
  - Setup part : Initialize the system with the test case, namely the inputs and expected outputs.
  - Call part : Call the object or method to be tested.
  - Assertion part : Compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful, if false, then it has failed.

Developing Unit Test Cases

- Two types of unit test cases
  - **Positive**
    - Reflect normal operation of a program.
    - Should show that the component works as expected.
  - **Negative**
    - Based on testing experience of where common problems arise.
    - Use abnormal inputs to check that these are properly processed and do not crash the component.

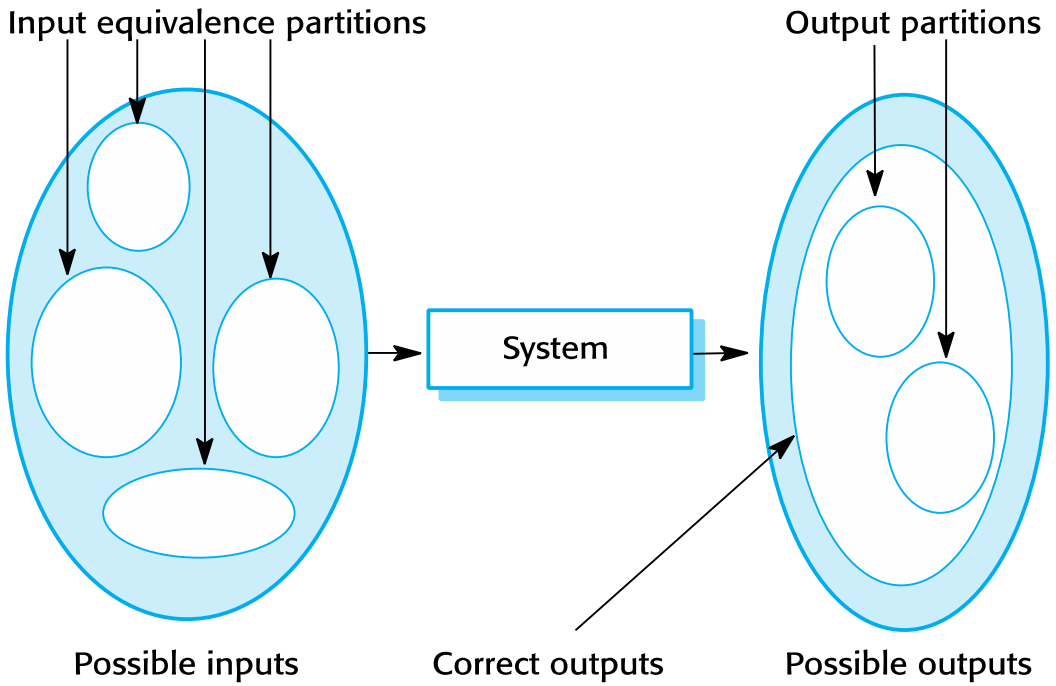
Testing Strategies

- **Partition testing**
  - Identify groups of inputs that have common characteristics and should be processed in the same way.
  - Choose tests from within each of these groups.

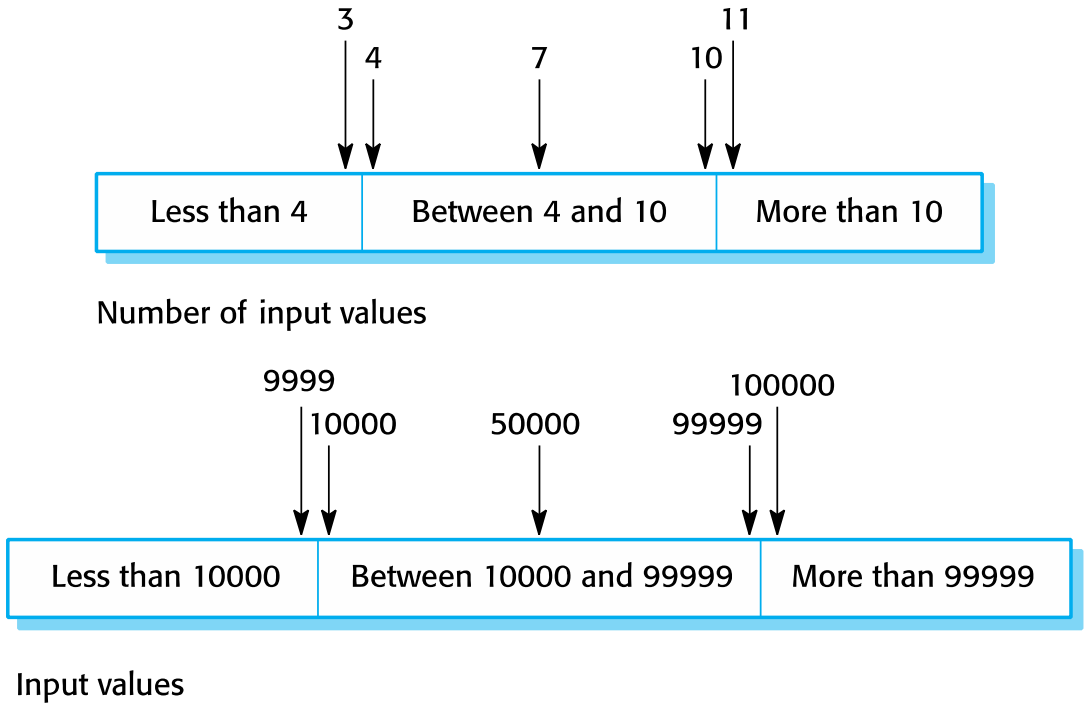
- **Guideline-based testing**
  - Use testing guidelines to choose test cases.
  - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.
    - Brute-force testing

Partition Testing

- Input data and output results often fall into different classes where all members of a class are related.
  - Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
  - Test cases should be chosen from each partition.



Equivalence Partitions with Boundary Value Analysis



Testing with Guidelines

- **General testing guidelines**
  - Choose inputs that force the system to generate all error messages.
  - Design inputs that cause input buffers to overflow.
  - Repeat the same input or series of inputs numerous times.
  - Force invalid outputs to be generated.
  - Force computation results to be too large or too small.

- For example, testing software with sequences which have only a single value can be guided:
  - Use sequences of different sizes in different tests.
  - Derive tests so that the first, middle and last elements of the sequence are accessed.
  - Test with sequences of zero length.

Component Testing

- **Software components** are often composite components that are made up of several interacting objects.
  - Can access the functionality of these objects through the defined **component interface**.

- **Component testing** is the testing of composite components.
  - Focus on showing that the component interface behaves according to its specification.
  - Assume that unit tests on the individual objects within the component have been completed.

  - Component testing ≈ Interface testing ≠ Interaction testing ≈ Integration testing

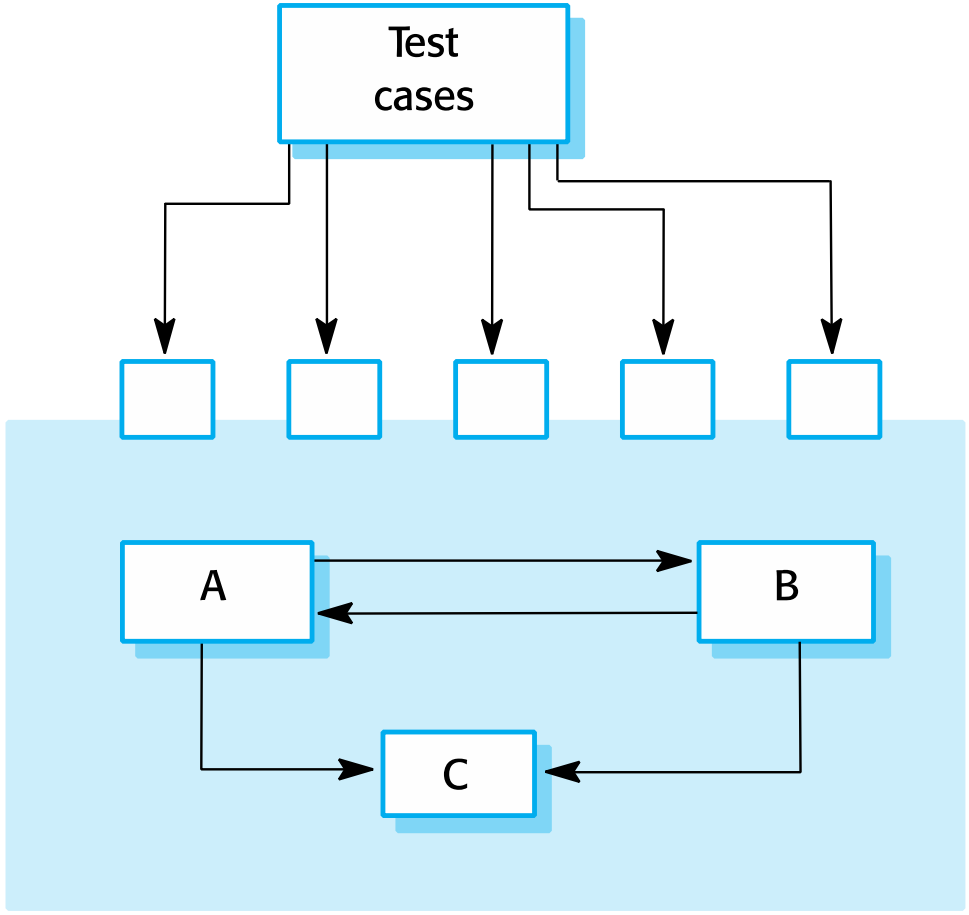
Interface Testing

- To detect faults due to interface errors or invalid assumptions about interfaces.

- Interface types
  - Parameter interfaces : Data passed from one method or procedure to another.
  - Shared memory interfaces : Block of memory is shared between procedures or functions.
  - Procedural interfaces : Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - Message passing interfaces : Sub-systems request services from other sub-systems

- Interface errors
  - Interface misuse
    - A calling component calls another component and makes an error in its use of its interface, e.g., parameters in the wrong order.
  - Interface misunderstanding
    - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
  - Timing errors
    - The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface Testing



Interface Testing Guidelines

- Interface Testing Guidelines
  - Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
  - Always test pointer parameters with null pointers.
  - Design tests which cause the component to fail.
  - Use stress testing in message passing systems.
  - In shared memory systems, vary the order in which components are activated.

System Testing

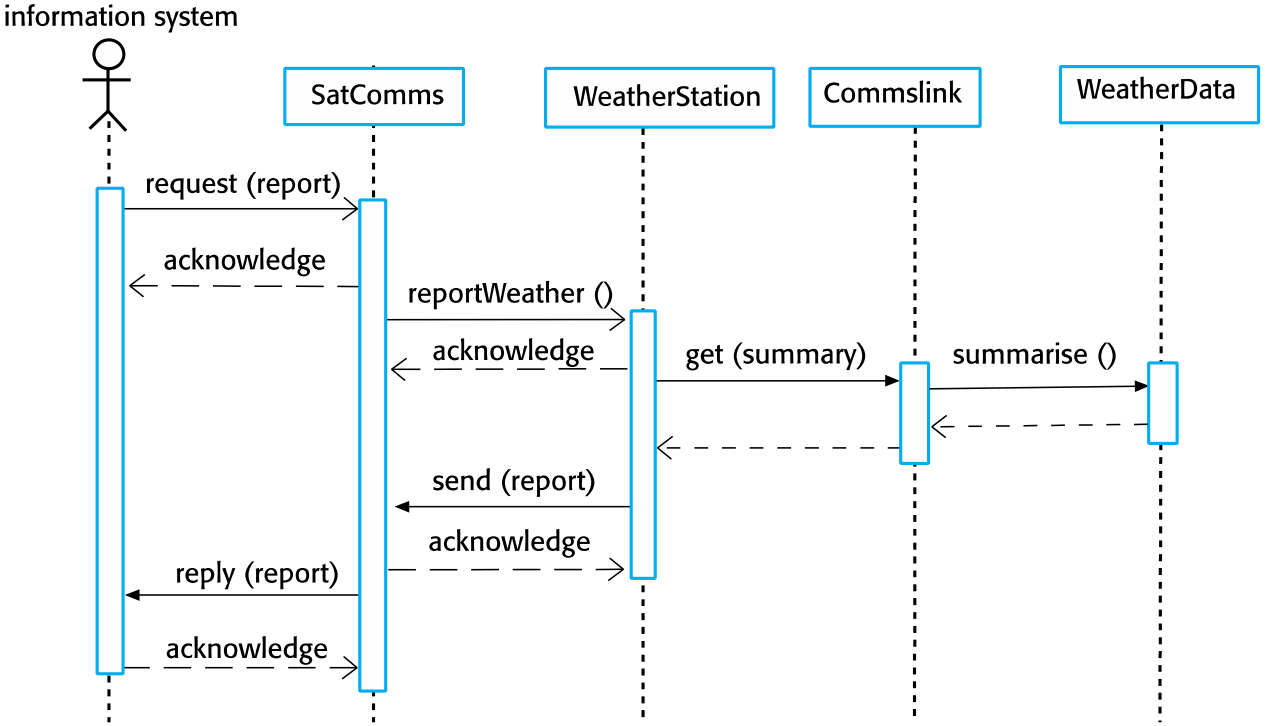
- **System testing** during development involves integrating components to create a version of the system and then testing the integrated system.
  - The focus is testing the interactions between components. **(Integration testing)**
  - Checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
  - Tests the emergent behavior of a system. **(System testing)**

- System testing is a collective process rather than an individual.
  - Reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
  - Components developed by different team members or sub-teams may be integrated at this stage.
  - System testing may involve a separate testing team with no involvement from designers and programmers.

Developing System Test Cases

- **Use-cases** and **Sequence diagrams** can be used as a basis.
  - Each use case usually involves several system components so testing the use case forces these interactions to occur.
  - Sequence diagrams associated with the use case documents the components and interactions that are being tested.

- Example : Collecting weather data SD



Testing Policies

- Exhaustive system testing is impossible.
  - **Testing policies** define a required system test coverage.

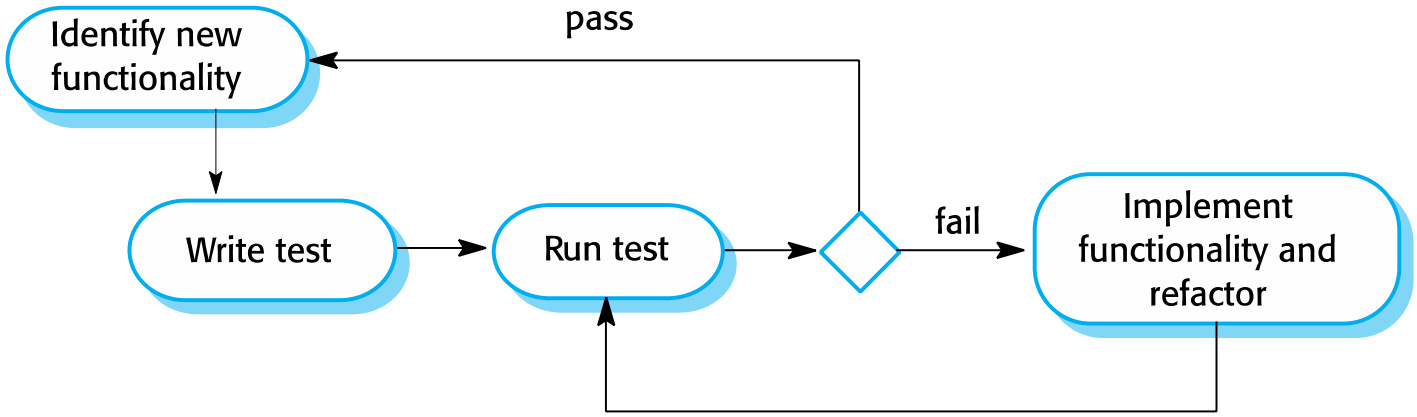
- Examples of testing policies
  - "All system functions that are accessed through menus should be tested."
  - "Combinations of functions accessed through the same menu must be tested."
  - "Where user input is provided, all functions must be tested with both correct and incorrect input."

Test-Driven Development

Test-Driven Development

- **Test-driven development (TDD)** is a program development approach inter-leaving testing and code development.
  - Tests are written before code and 'passing' the tests is the critical driver of development.
  - Develop code incrementally, along with a test for that increment.
  - Not move on to the next increment, until the code passes its test.

- TDD was introduced as part of agile methods such as **XP**.
  - However, it can also be used in plan-driven development processes.



TDD Process

- The typical TDD process
  1. Start by identifying the increment of functionality that is required.
     - Should normally be small and implementable in a few lines of code.
  2. Write a test for this functionality and implement this as an automated test.
  3. Run the test, along with all other tests that have been implemented.
     - Initially, we have not implemented the functionality so the new test will fail.
  4. Implement the functionality and re-run the test.
  5. Once all tests run successfully, move on to implementing the next chunk of functionality.

Benefits of TDD

- **Code coverage**
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- **Regression testing**
  - A regression test suite is developed incrementally as a program is developed.
  - Tests the system to check that changes have not 'broken' previously working code through rerunning the tests every time a change is made to the program.
- **Simplified debugging**
  - When a test fails, it should be obvious where the problem lies.
  - The newly written code needs to be checked and modified.
- **System documentation**
  - The tests themselves are a form of documentation that describe what the code should be doing.

Release Testing

Release Testing

- **Release testing** is the process of testing a particular release of a system that is intended for use outside of the development team.
  - To convince the supplier of the system that it is good enough for use.
  - Should show that the system delivers its specified functionality, performance and dependability.
  - Should show the system does not fail during normal use.

- Release testing is usually **a black-box testing** process.
  - Tests are only derived from the system specification.

Release Testing vs. System Testing

- **Release testing** is a form of system testing.

- Important differences are
  - A separate team that has not been involved in the system development should be responsible for release testing.
  - System testing by the development team should focus on discovering bugs in the system (defect/verification testing).
  - The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

- **Performance tests**
  - Involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

- **Stress testing**
  - A form of performance testing where the system is deliberately overloaded to test its failure behavior.

Requirements-Based Testing

- Requirements-based testing examines each requirement to develop test cases.

- Example : Mentcare system

Requirements	If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
	If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.
Test cases	Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
	Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system.
	Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
	Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
	Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

User Testing

User Testing

- **User or customer testing** is a stage in which users or customers provide input and advice on system testing.
  - Influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

- Types of user testing
  - **Alpha testing**
    - Users of the software work with the development team to test the software at the developer's site.
  - **Beta testing**
    - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
  - **Acceptance testing**
    - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.
    - Primarily for custom systems

Agile Methods and Acceptance testing

- There is no separate acceptance testing process.
  - In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
  - Tests are defined by the user/customer and are integrated with other tests in that they run automatically when changes are made.

- Main problem is whether or not the embedded user is ‘typical’ and can represent the interests of all system stakeholders.

Key Points

- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.
- Development testing includes unit testing, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.
- When testing software, you should try to ‘break’ the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.

Key Points

- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- Test-first development is an approach to development where tests are written before the code to be tested.
- Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.

Chapter 9. Software Evolution

Topics Covered

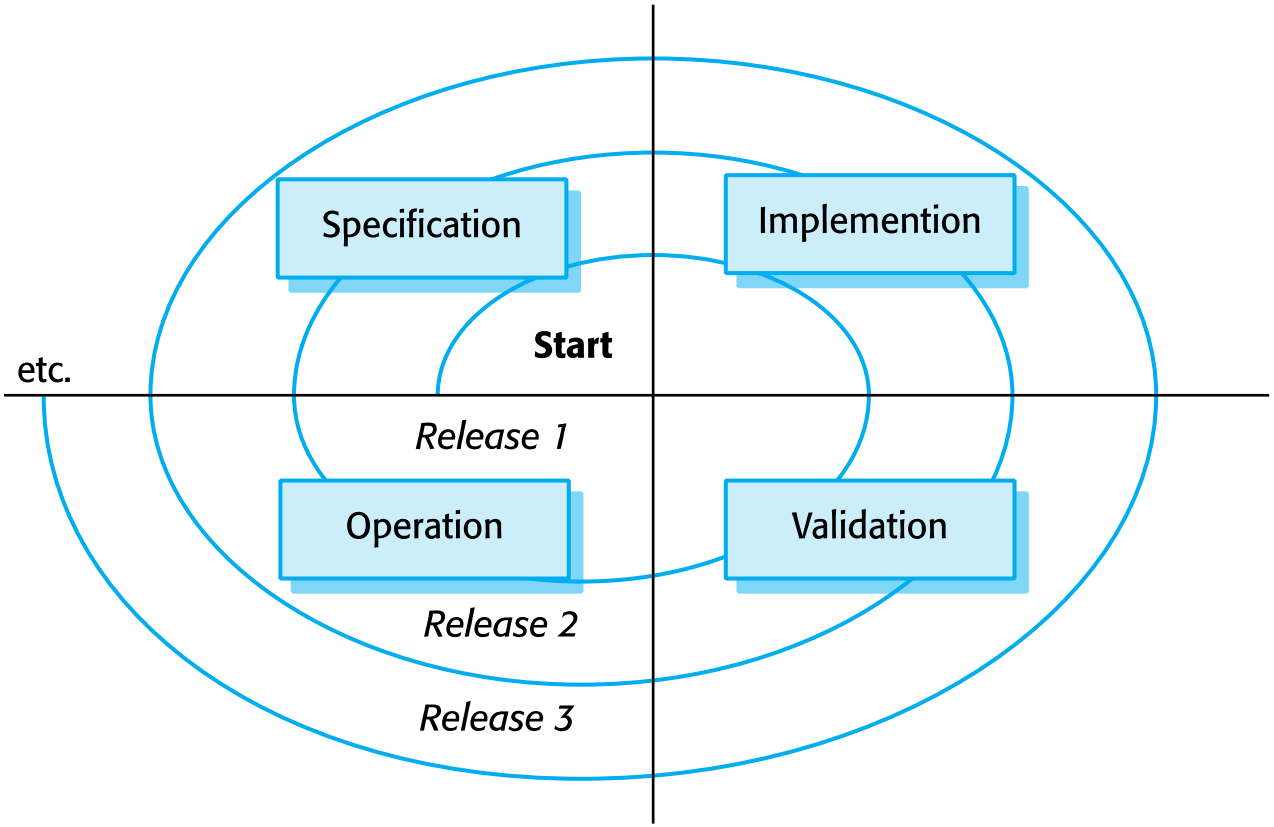
- Evolution processes
- Legacy systems
- Software maintenance

Software Change

- Software change is **inevitable**.
  - New requirements emerge when the software is used.
  - The business environment changes.
  - Errors must be repaired.
  - New computers and equipment is added to the system.
  - The performance or reliability of the system may have to be improved.

- A key problem for all organizations is **implementing and managing change** to their existing software systems.
  - The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

A Spiral Model of Development and Evolution

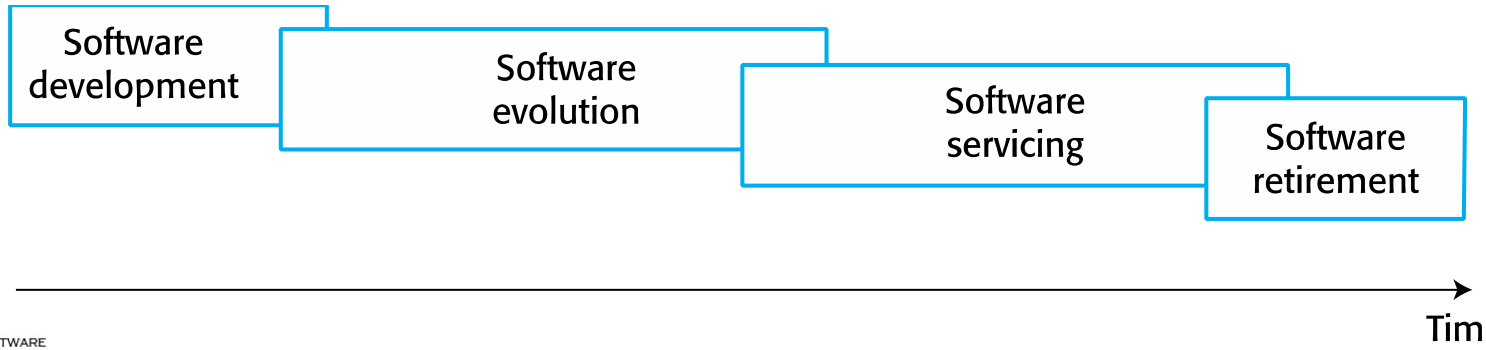


Evolution and Servicing

- **Evolution**
  - The stage in a software system's life cycle, where it is in operational use and is evolving as new requirements are proposed and implemented in the system.

- **Servicing**
  - At this stage, the software remains useful, but the only changes made are those required to keep it operational, i.e., bug fixes and changes to reflect changes in the software's environment.
  - No new functionality is added.

- **Phase-out (Retirement)**
  - The software may still be used but no further changes are made to it.



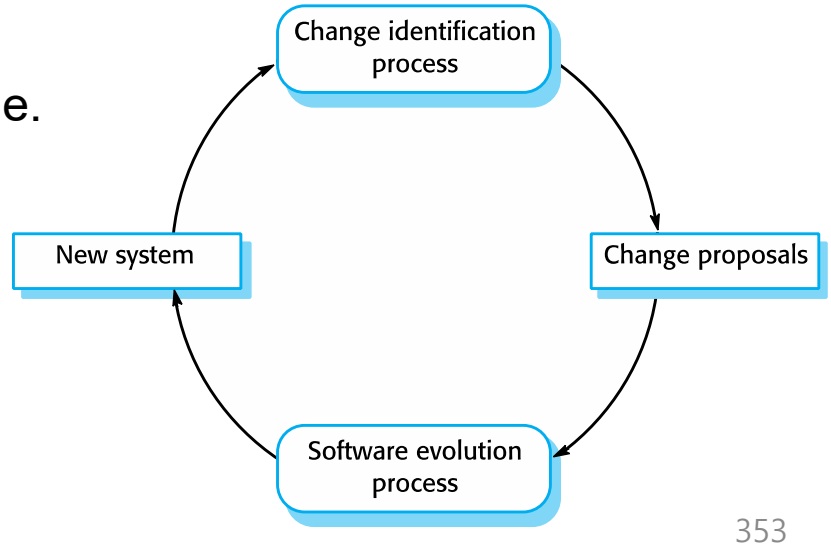
Evolution Processes

Evolution Processes

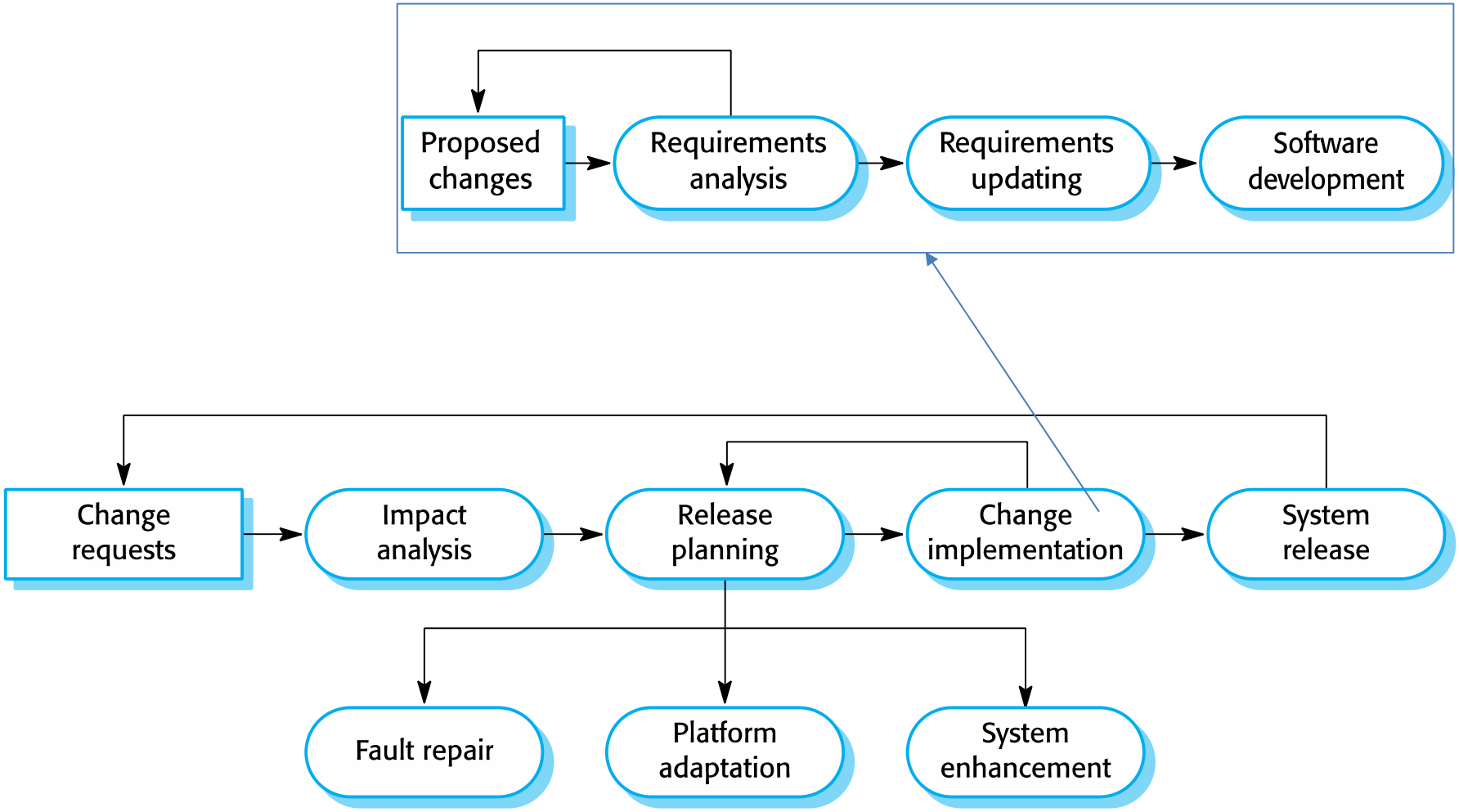
- **Software evolution** processes depend on
  - The type of software being maintained,
  - The development processes used, and
  - The skills and experience of the people involved.

- Proposals for change are the driver for system evolution.
  - Should be linked with components that are affected by the change.
  - Should allow the cost and impact of the change to be estimated.

- Change identification and evolution continues throughout the system lifetime.

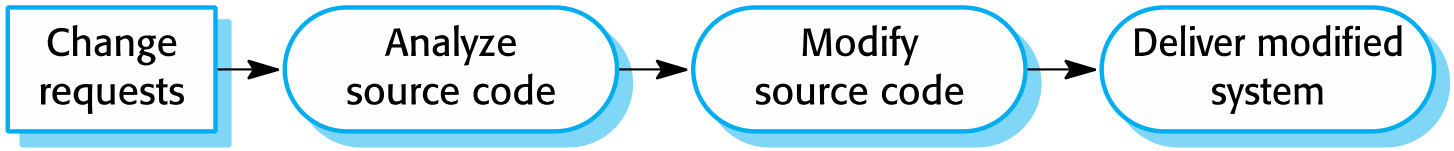


The Software Evolution Process



Urgent Change Requests

- **Urgent changes** may have to be implemented without going through all stages of the software engineering process
  - If a serious system fault must be repaired to allow normal operation to continue
  - If changes to the system's environment (e.g., OS upgrade) have unexpected effects
  - If there are business changes that require a very rapid response (e.g., release of a competing product)



Agile Methods and Evolution

- Agile methods are based on incremental development so the transition from development to evolution is a **seamless** one.
  - Evolution is simply a continuation of the development process based on frequent system releases.
  - Automated regression testing is particularly valuable when changes are made to a system.
  - Changes may be expressed as additional user stories.
- Under the assumption that the Agile development teams have been maintained.
  - Should avoid **handover problems**

Handover Problems

- Where the development team have used an agile approach, but the evolution team is unfamiliar with agile methods and prefer a plan-based approach.
  - The evolution team may expect detailed documentation to support evolution and this is not produced in agile processes.

- Where a plan-based approach has been used for development, but the evolution team prefer to use agile methods.
  - The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.

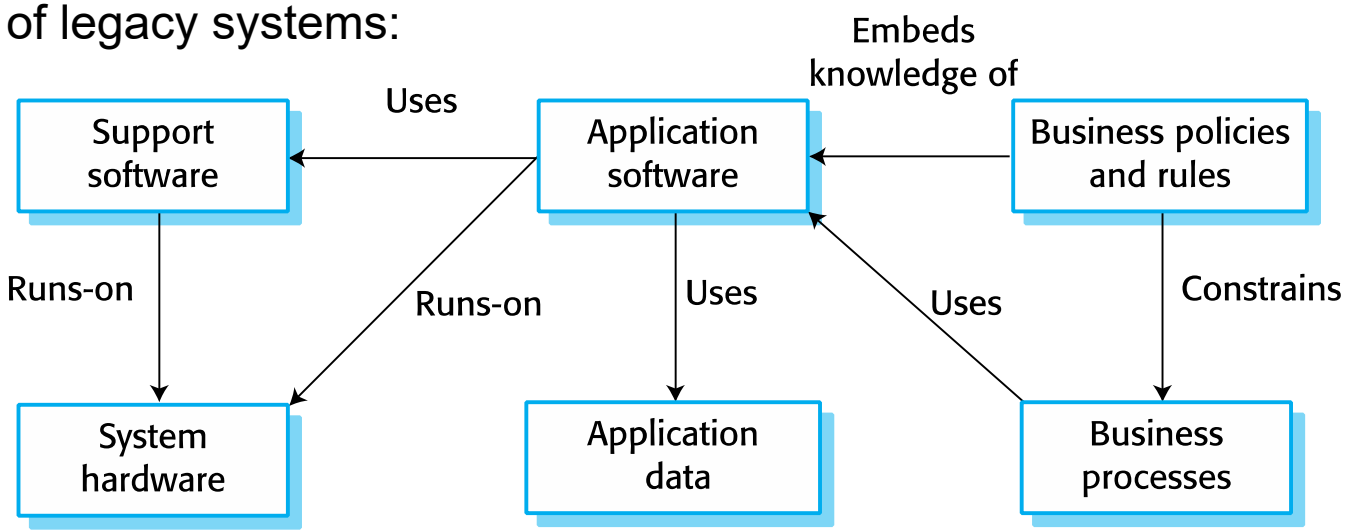
Legacy Systems

Legacy Systems

- **Legacy systems** are older systems that rely on languages and technology that are no longer used for new systems development.
  - May be dependent on older hardware such as mainframe computers.
  - May have associated legacy processes and procedures.

- Legacy systems are broader socio-technical systems.
  - Include hardware, software, libraries and other supporting software and business processes.

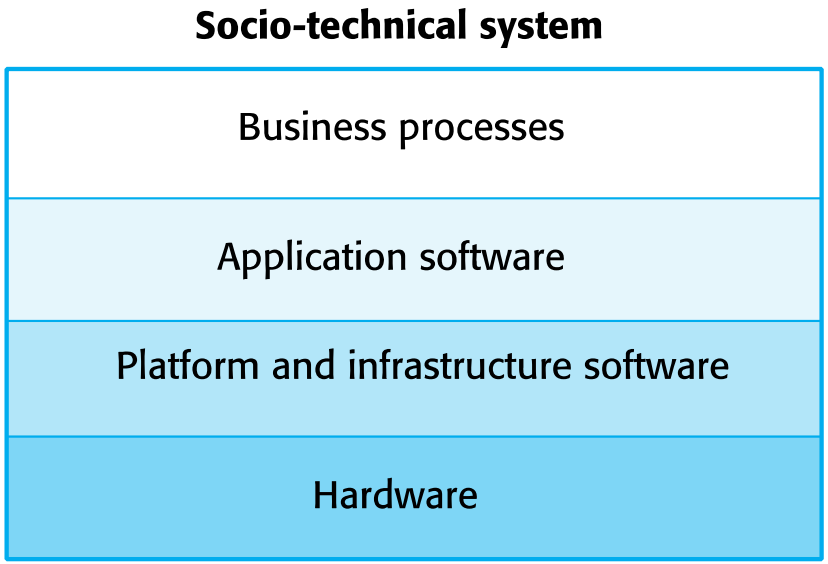
- Elements of legacy systems:



Components of Legacy System Components

Element	Description
System hardware	Legacy systems may have been written for hardware that is no longer available.
Support software	The legacy system may rely on a range of support software, which may be obsolete or unsupported.
Application software	The application system that provides the business services is usually made up of a number of application programs.
Application data	These are data that are processed by the application system. They may be inconsistent, duplicated or held in different databases.
Business processes	These are processes that are used in the business to achieve some business objective. Business processes may be designed around a legacy system and constrained by the functionality that it provides
Business policies and rules	These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

Typical Layers of Legacy Systems



Legacy System Replacement and Change

- **Legacy system replacement** is risky and expensive.
  - Lack of complete system specification
  - Tight integration of system and business processes
  - Undocumented business rules embedded in the legacy system
  - New software development may be late and/or over budget.
  - The system is still in use.

- **Legacy system change** is also expensive.
  - No consistent programming style
  - Use of obsolete programming languages with few people available with these language skills
  - Inadequate system documentation
  - System structure degradation
  - Program optimizations may make them hard to understand
  - Data errors, duplication and inconsistency

Legacy System Management

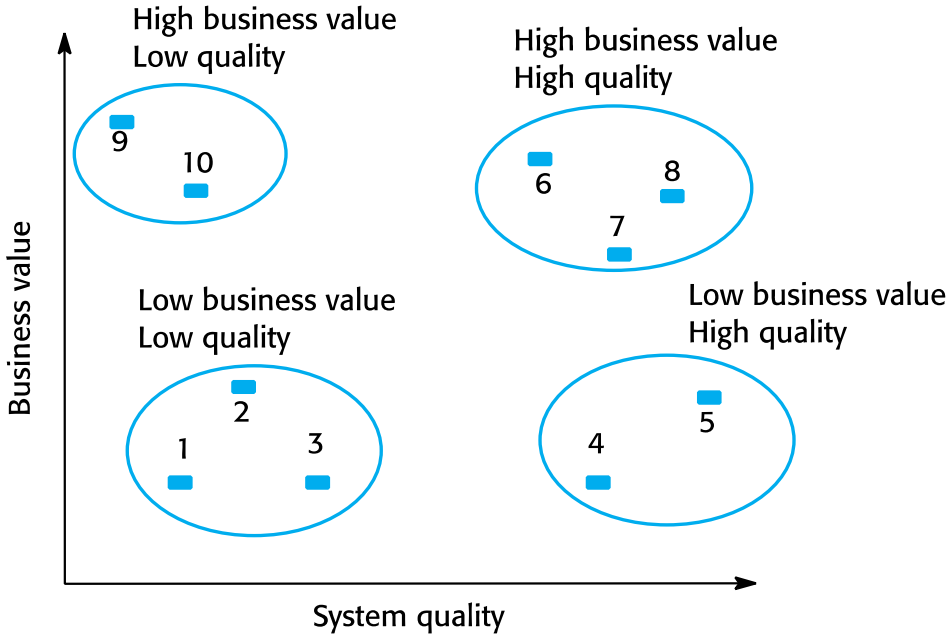
- Organizations relying on legacy systems should decide one belows
  - Scrap the system completely and modify business processes so that it is no longer required, or
  - Continue maintaining the system, or
  - Transform the system by re-engineering to improve its maintainability, or
  - Replace the system with a new system.

- Legacy system assessment
  - Assess the system quality and its business value to choose appropriate strategy.

- Legacy system categories
  - Low quality, low business value
  - Low-quality, high-business value
  - High-quality, low-business value
  - High-quality, high business value

Legacy System Assessment & Categories

- Low quality, low business value
  - These systems should be scrapped.
- Low-quality, high-business value
  - These make an important business contribution but are expensive to maintain.
  - Should be re-engineered or replaced if a suitable system is available.
- High-quality, low-business value
  - Replace with COTS, scrap completely or maintain.
- High-quality, high business value
  - Continue in operation using normal system maintenance.



Business Value Assessment

- Assessment should take different viewpoints into account.
  - System end-users, Business customers, IT managers, Senior managers, etc.
- Interview different stakeholders and collate results.
- Issues in Business Value Assessment
  - The use of the system
    - If systems are only used occasionally or by a small number of people, they may have a low business value.
  - The business processes that are supported
    - A system may have a low business value if it forces the use of inefficient business processes.
  - System dependability
    - If a system is not dependable and the problems directly affect business customers, the system has a low business value.
  - The system outputs
    - If the business depends on system outputs, then the system has a high business value.

System Quality Assessment

- Business process assessment
  - How well does the business process support the current goals of the business?
- Environment assessment
  - How effective is the system's environment and how expensive is it to maintain?
- Application assessment
  - What is the quality of the application software system?

Business Process Assessment

- Use a viewpoint-oriented approach and seek answers from system stakeholders
  - Is there a defined process model and is it followed?
  - Do different parts of the organisation use different processes for the same function?
  - How has the process been adapted?
  - What are the relationships with other business processes and are these necessary?
  - Is the process effectively supported by the legacy application software?

- Example
  - A travel ordering system may have a low business value, because of the widespread use of web-based ordering.

Environment Assessment

Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?
Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?

Application Assessment

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

System Measurement

- **Quantitative data** will help to make an assessment of the quality of the application system.

- Examples are
  - The number of system change requests.
    - The higher this accumulated value, the lower the quality of the system.
  - The number of different user interfaces used by the system.
    - The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.
  - The volume of data used by the system.
    - As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data.
    - Cleaning up old data is a very expensive and time-consuming process

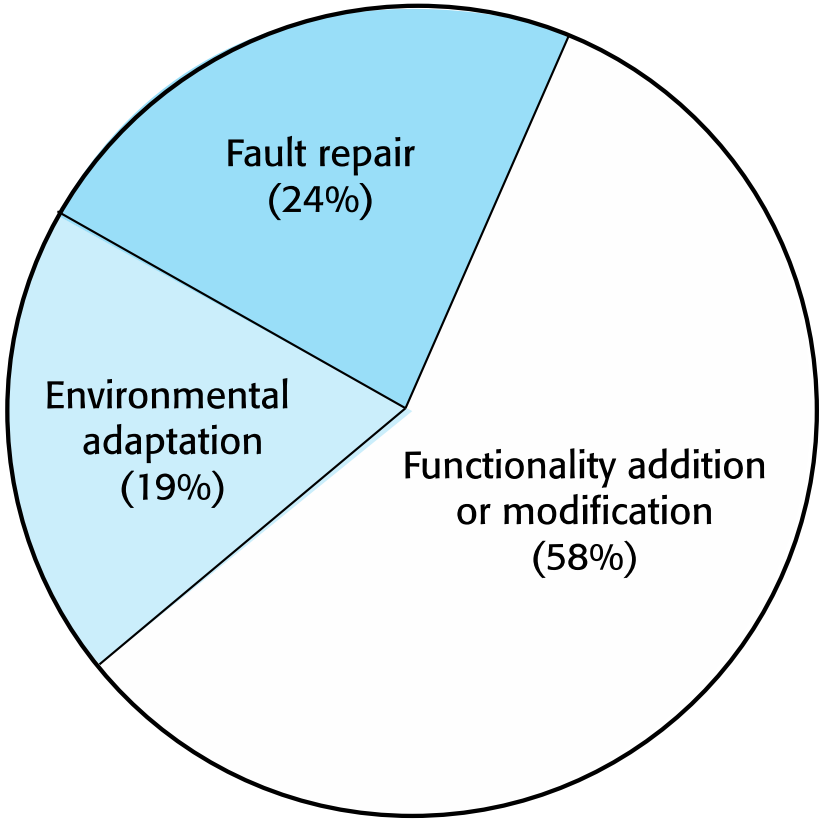
Software Maintenance

Software Maintenance

- **Software maintenance**
  - Modifying a program after it has been put into use.
  - Mostly used for changing custom software.
    - Generic software products are said to evolve to create new versions.
  - Not normally involve major changes to the system's architecture.
  - Changes are implemented by modifying existing components and adding new components to the system.

- Types of maintenance
  - **Fault repairs**
    - Changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements.
  - **Environmental adaptation**
    - Maintenance to adapt software to a different operating environment
    - Changing a system to operate in a different environment (computer, OS, etc.).
  - **Functionality addition and modification**
    - Modifying the system to satisfy new requirements.

Maintenance Effort Distribution

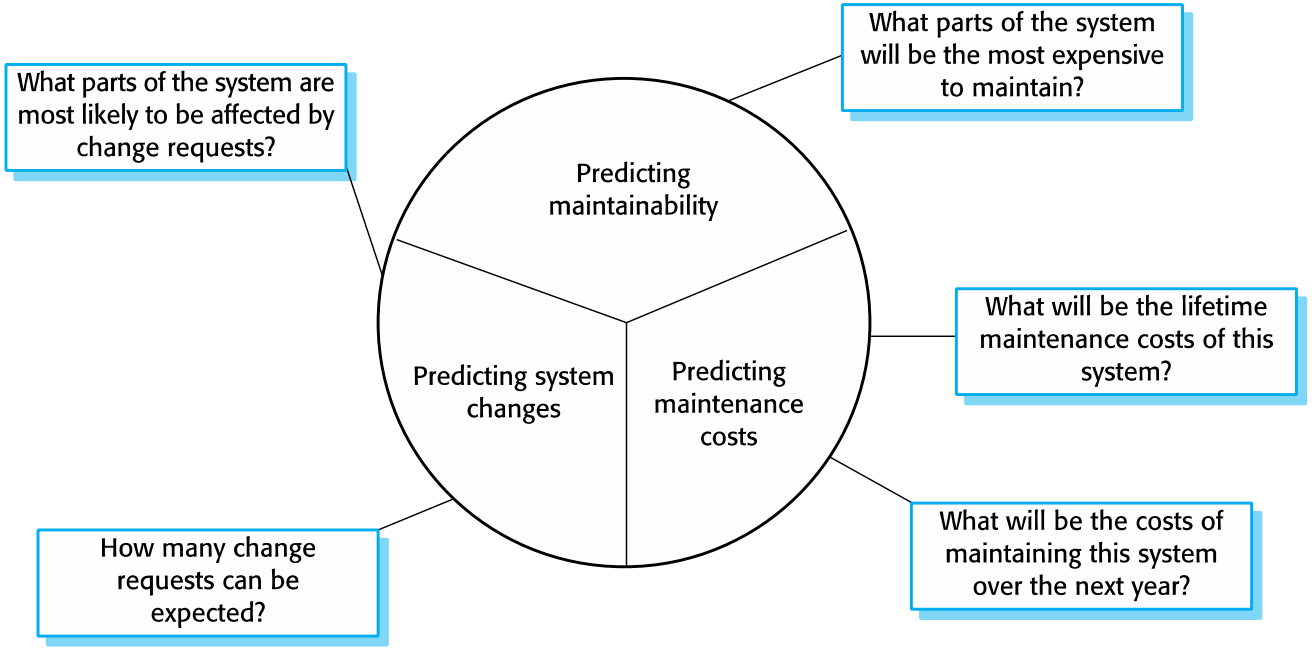


Maintenance Costs

- **Maintenance costs**
  - Usually greater than development costs (2* to 100* depending on the application).
  - Affected by both technical and non-technical factors.
  - Increases as software is maintained.
    - Maintenance corrupts the software structure so makes further maintenance more difficult.
  - Ageing software can have high support costs.
    - Old languages, compilers etc.
    - As programs age, their structure degrades and they become harder to change

Maintenance Prediction

- **Maintenance prediction** is concerned with assessing which parts of the system may cause problems and have high maintenance costs.
  - Change acceptance depends on the maintainability of the components affected by the change.
  - Implementing changes degrades the system and reduces its maintainability.
  - Maintenance costs depend on the number of changes and costs of change depend on maintainability.



Change Prediction

- **Change prediction**
  - Predicting the number of changes requires.
  - Predicting understanding of the relationships between a system and its environment.

- Tightly coupled systems require changes whenever the environment is changed.

- Factors influencing this relationship are
  - Number and complexity of system interfaces
  - Number of inherently volatile system requirements
  - The business processes where the system is used

Metrics for Change Prediction

- **Process metrics** may be used to assess maintainability
  - Number of requests for corrective maintenance
  - Average time required for impact analysis
  - Average time taken to implement a change request
  - Number of outstanding change requests

- **Complexity metrics** of system components may be used to assess maintainability.
  - Complexity of control structures
  - Complexity of data structures
  - Object, method (procedure) and module size

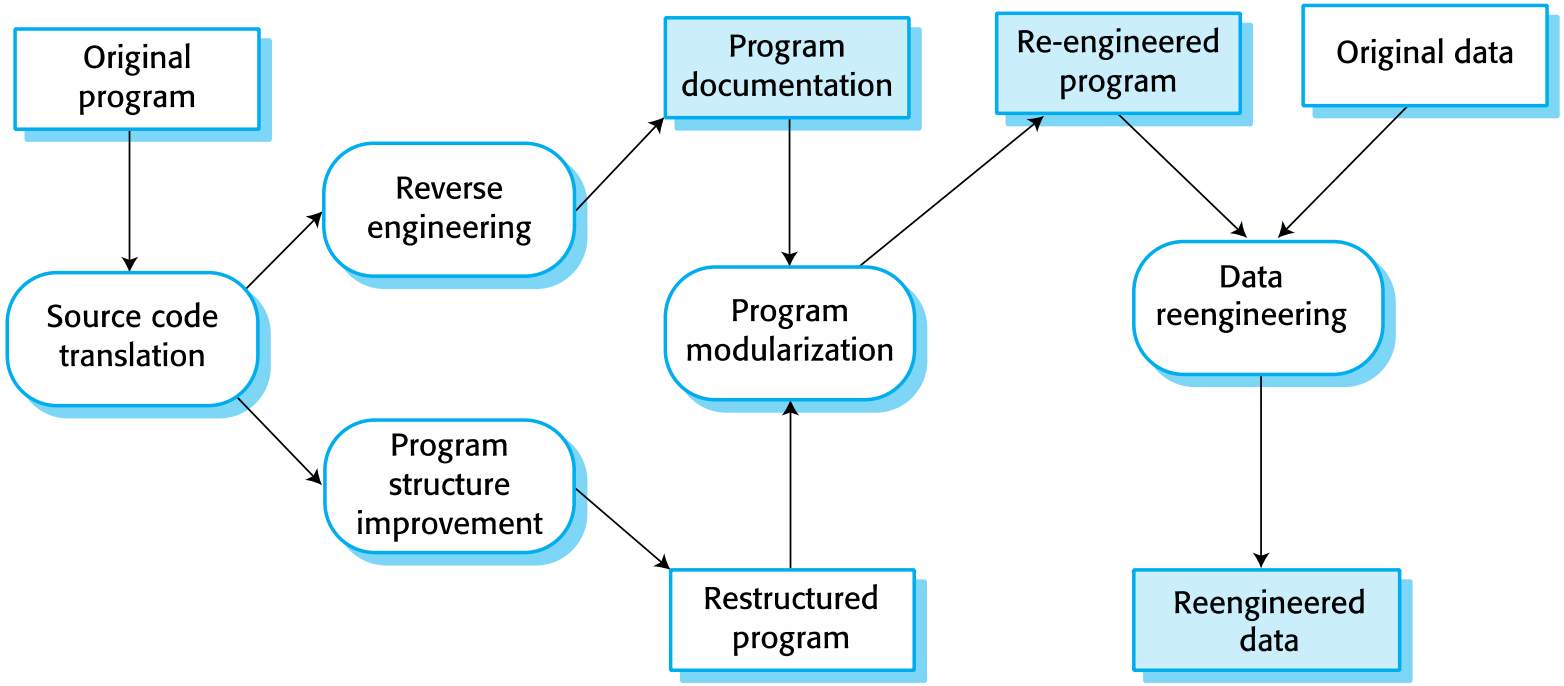
  - Studies have shown that most maintenance effort is spent on a relatively small number of system components.

Software Reengineering

- Restructuring or rewriting part or all of a legacy system without changing its functionality.
  - Applicable where some but not all sub-systems of a larger system require frequent maintenance.
  - Involves adding effort to make them easier to maintain.
    - The system may be re-structured and re-documented.

- Advantages of **reengineering**
  - Reduced risk
    - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
  - Reduced cost
    - The cost of re-engineering is often significantly less than the costs of developing new software.

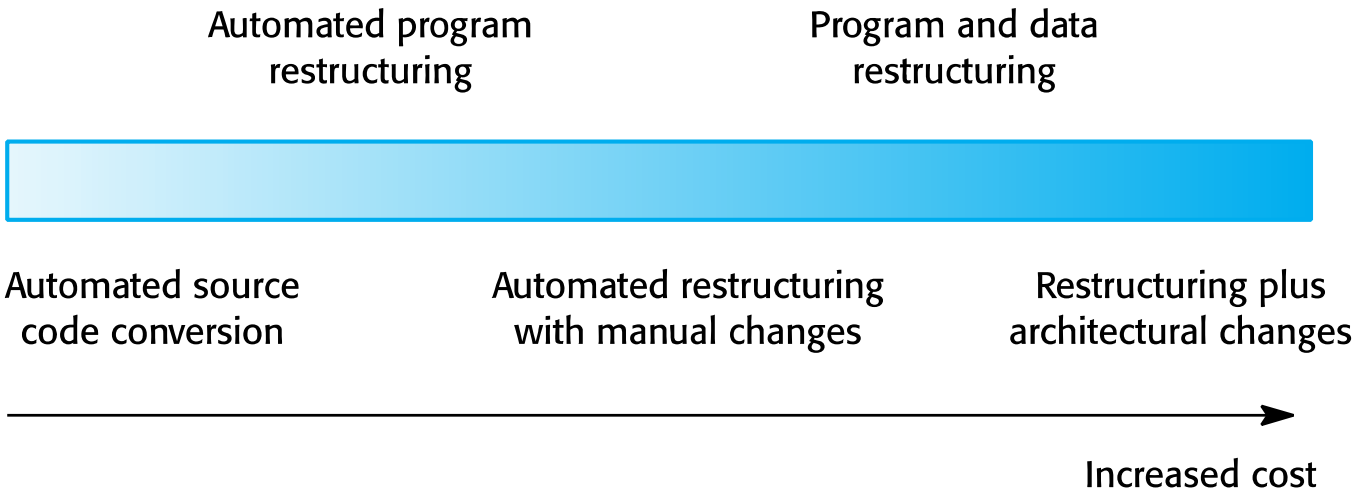
The Reengineering Process



Reengineering Process Activities

- **Source code translation**
  - Convert code to a new language.
- **Reverse engineering**
  - Analyze the program to understand it.
- **Program structure improvement**
  - Restructure automatically for understandability.
- **Program modularization**
  - Reorganize the program structure.
- **Data reengineering**
  - Clean-up and restructure system data.

Steps of Reengineering



Refactoring

- **Refactoring** is the process of making improvements to a program to slow down degradation through change.
  - 'Preventative maintenance' that reduces the problems of future change.
  - Involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.

- When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Refactoring and Reengineering

- **Re-engineering** takes place after a system has been maintained for some time and maintenance costs are increasing.
  - Use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.

- **Refactoring** is a continuous process of improvement throughout the development and evolution process.
  - To avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

‘Bad smells’ in Program Code

- Duplicate code
  - The same or very similar code may be included at different places in a program.
  - This can be removed and implemented as a single method or function that is called as required.
- Long methods
  - If a method is too long, it should be redesigned as a number of shorter methods.
- Switch (case) statements
  - These often involve duplication, where the switch depends on the type of a value.
  - The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
- Data clumping
  - Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program.
  - These can often be replaced with an object that encapsulates all of the data.
- Speculative generality
  - This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Key Points

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.
- Legacy systems are older software systems, developed using obsolete software and hardware technologies, that remain useful for a business.
- It is often cheaper and less risky to maintain a legacy system than to develop a replacement system using modern technology.

Key Points

- The business value of a legacy system and the quality of the application should be assessed to help decide if a system should be replaced, transformed or maintained.
- There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
- Software re-engineering is concerned with re-structuring and re-documenting software to make it easier to understand and change.
- Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.