Software Engineering

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

http://dslab.konkuk.ac.kr

Ver. 2.0 (2010.06)

Introduction

- This lecture provides
  - Part I. Overview
  - Part II. Requirements
  - Part III. Design
  - Part IV. Development
  - Part V. Verification and Validation
  - Part VI. Managing People

- Practice
  - SASD : Analysis → Design → Implementation

Part I. Overview

Chapter 1.

Introduction to Software Engineering

Objectives

- To introduce software engineering
- To explain software engineering's importance
- To answer key questions about software engineering

Software Engineering

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

- Let's define software engineering through 11 FAQs as follows.

FAQs about Software Engineering

1. What is software?
2. What is software engineering?
3. What is the difference between software engineering and computer science?
4. What is the difference between software engineering and system engineering?
5. What is a software process?
6. What is a software process model?
7. What are the costs of software engineering?
8. What are software engineering methods?
9. What is CASE (Computer-Aided Software Engineering) ?
10. What are the attributes of good software?
11. What are the key challenges facing software engineering?

1. What is Software?

- Software is computer programs and associated documentation such as requirements, design models and user manuals

- Software products may be developed for a particular customer or for a general market.
  - Generic : developed to be sold to a range of different customers. e.g. PC software such as Excel or Word
  - Bespoke (custom) : developed for a single customer according to their specification. e.g. Software used in a hospital

2. What is Software Engineering?

- Software engineering is
  - An engineering discipline that is concerned with all aspects of software production.
  - All things concerned with a successful development of software

- Software engineers should
  - adopt a systematic and organised approach
  - use
    - appropriate tools,
    - techniques depending on the problem to be solved,
    - development constraints,
    - resources available.

3. What is the Difference between Software Engineering and Computer Science?

- Computer science is concerned with theory and fundamentals.

- Software engineering is concerned with the practicalities of developing and delivering useful software.

- Computer science theories are insufficient to act as a complete underpinning for software engineering (unlike physics and electrical engineering), since it is practiced/performed by people.

4. 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 system engineering process concerned with developing the software infrastructure, control, applications and databases in the system.

5. What is a Software Process?

- Software process is a set of activities whose goal is the development or evolution of software.

- Generic activities in all software processes
  - Specification : what the system should do
  - Development : production of the software system
  - Validation : checking that the software is really what the customer wants
  - Evolution : changing the software in response to changing demands

6. What is a Software Process Model?

- A simplified representation of a software process, presented from a specific perspective.

- Examples of process perspectives
  - Workflow perspective : sequence of activities
  - Data-flow perspective : information flow
  - Role/action perspective : who does what

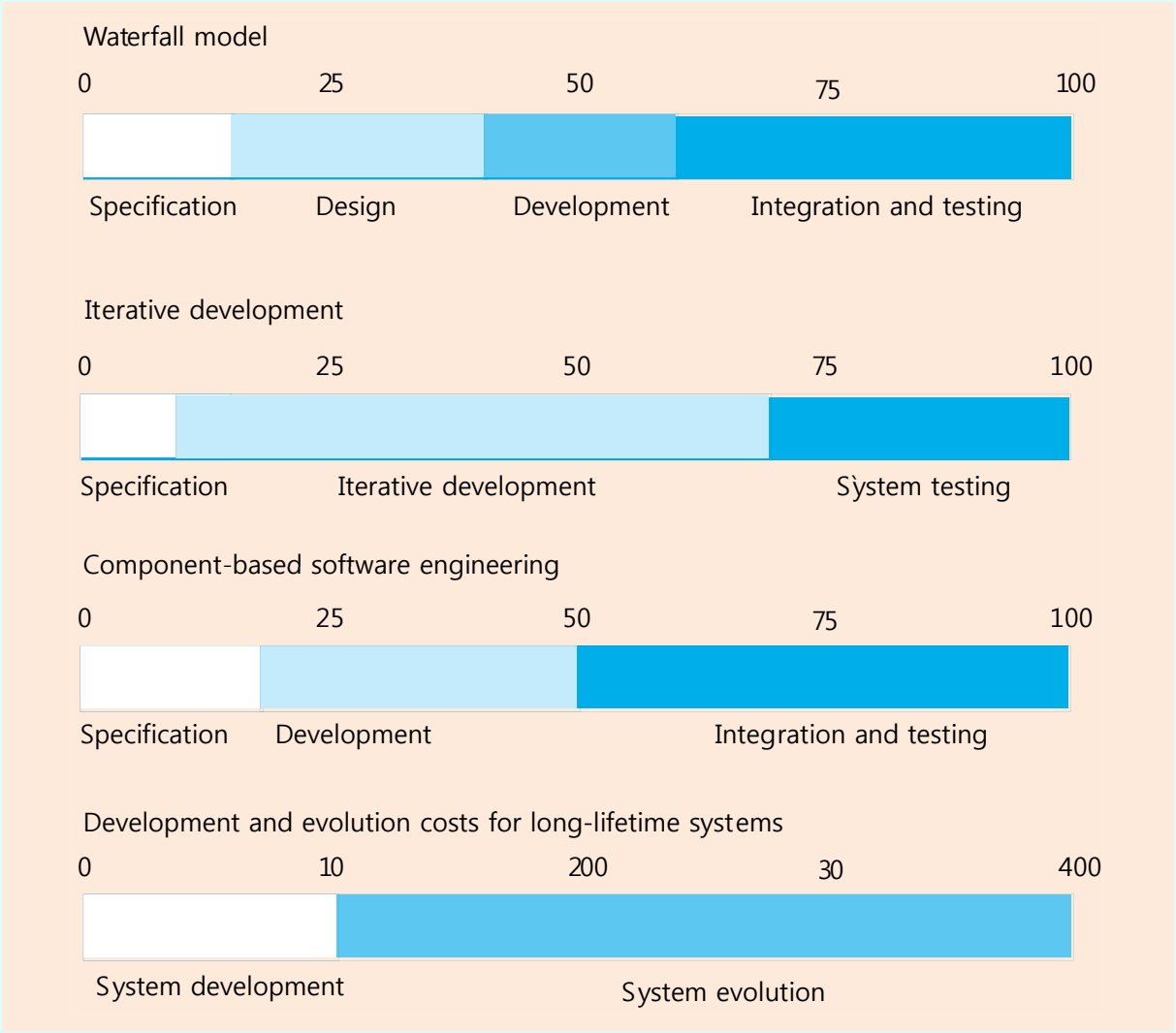
- Generic process models
  - Waterfall
  - Iterative development
  - Component-based software engineering

7. What are the Costs of Software Engineering?

- Development costs are roughly
  - 60% : development costs
  - 40% : testing costs
  - For custom (long-lifetime) software, evolution costs often exceed development costs.

- Costs can vary depending on
  - the type of system being developed
  - the requirements of system attributes (performance and system reliability)

- Therefore, distribution of costs depends on the development model that is used.



Activity-cost distribution varying depending on software process models

8. What are Software Engineering Methods?

- Organized way of producing software according to the process

- Structured approaches to software development which include system models, notations, rules, design advice and process guidance.
  - Model descriptions
    - Descriptions of graphical models which should be produced
  - Rules
    - Constraints applied to system models
  - Recommendations
    - Advice on good design practice
  - Process guidance
    - What activities to follow

9. What is CASE (Computer-Aided Software Engineering) ?

- CASEs are Software systems that are intended to provide automated support for software process activities and software engineering methods.
  - Requirements and design
  - Programming and debugging
  - Testing

10. What are the Attributes of Good Software?

- The good software should
  - deliver the required functionality and performance to the user
  - be maintainable, dependable and acceptable.

- Maintainability
  - Software must evolve to meet changing needs.
- Dependability
  - Software must be trustworthy.
- Efficiency
  - Software should not make wasteful use of system resources.
- Acceptability
  - Software must be accepted by the users for which it was designed.
  - It must be understandable, usable and compatible with other systems.

Summary

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software products consist of developed programs and associated documentation.
- Essential product attributes are maintainability, dependability, efficiency and usability.
- The software process consists of activities that are involved in developing software products. Basic activities are software specification, development, validation and evolution.
- Methods are organized ways of producing software. They include suggestions for the process to be followed, the notations to be used, rules governing the system descriptions produced, and design guidelines.
- CASE tools are software systems which are designed to support routine activities in the software process such as editing design diagrams, checking diagram consistency and keeping track of program tests which have been run.

Chapter 2.

Socio-technical Systems

Objectives

- To explain what a socio-technical system is
- To explain the distinction between a socio-technical system and a computer-based system
- To introduce the concept of emergent system properties
- To explain about system engineering
- To discuss legacy systems and why they are critical to many businesses

What is a System?

- A purposeful collection of inter-related components working together to achieve some common objective
  - May include software, mechanical, electrical and electronic hardware and be operated by people.

- Technical computer-based systems
  - Include hardware and software, but where the operators and operational processes are not normally considered to be part of the system.
  - The system is not self-aware. (e.g. Lap-top, MP3 player, cell phones, etc.)

- Socio-technical systems
  - Systems that include technical systems but also operational processes and people who use and interact with the technical system.
  - Socio-technical systems are governed by organisational policies and rules. (e.g. flight control system, transportation reservation system, etc.)

Characteristics of Socio-technical System

- Emergent properties
  - Properties of the system as a whole, depending on the system components and their relationships
  - Features :
    - non-deterministic
    - complex relationship with organizational objectives

  - Non-deterministic
    - They do not always produce the same output when presented with the same input, because the system's behaviour is partially dependent on human operators.

  - Complex relationships with organisational objectives
    - The extent to which the system supports organisational objectives does not just depend on the system itself.

Emergent Properties

- Emergent properties are a consequence of the relationships between system components.
- Therefore, they can only be assessed and measured once the components have been integrated into a system.

Property	Description
Volume	The volume of a system (the total space occupied) varies depending on how the component assemblies are arranged and connected.
Reliability	System reliability depends on component reliability but unexpected interactions can cause new types of failure and therefore affect the reliability of the system.
Security	The security of the system (its ability to resist attack) is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguards.
Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty and modify or replace these components.
Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators and its operating environment.

Types of Emergent Properties

- Functional properties
  - These appear when all the parts of a system work together to achieve some objective.
  - For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.

- Non-functional properties
  - These relate to the behaviour of the system in its operational environment.
  - Examples are reliability, performance, safety, and security.

Reliability of Systems

- We need to consider the reliability from aspect of systems
  - Even if we have reliable software components,
  - System failures often occur dus to unforeseen interactions between reliable components.
- Therefore, we need system engineering.

- Influences on reliability
  - Hardware reliability
    - What is the probability of a hardware component failing and how long does it take to repair that component?
  - Software reliability
    - How likely is it that a software component will produce an incorrect output.
  - Operator reliability
    - How likely is it that the operator of a system will make an error?

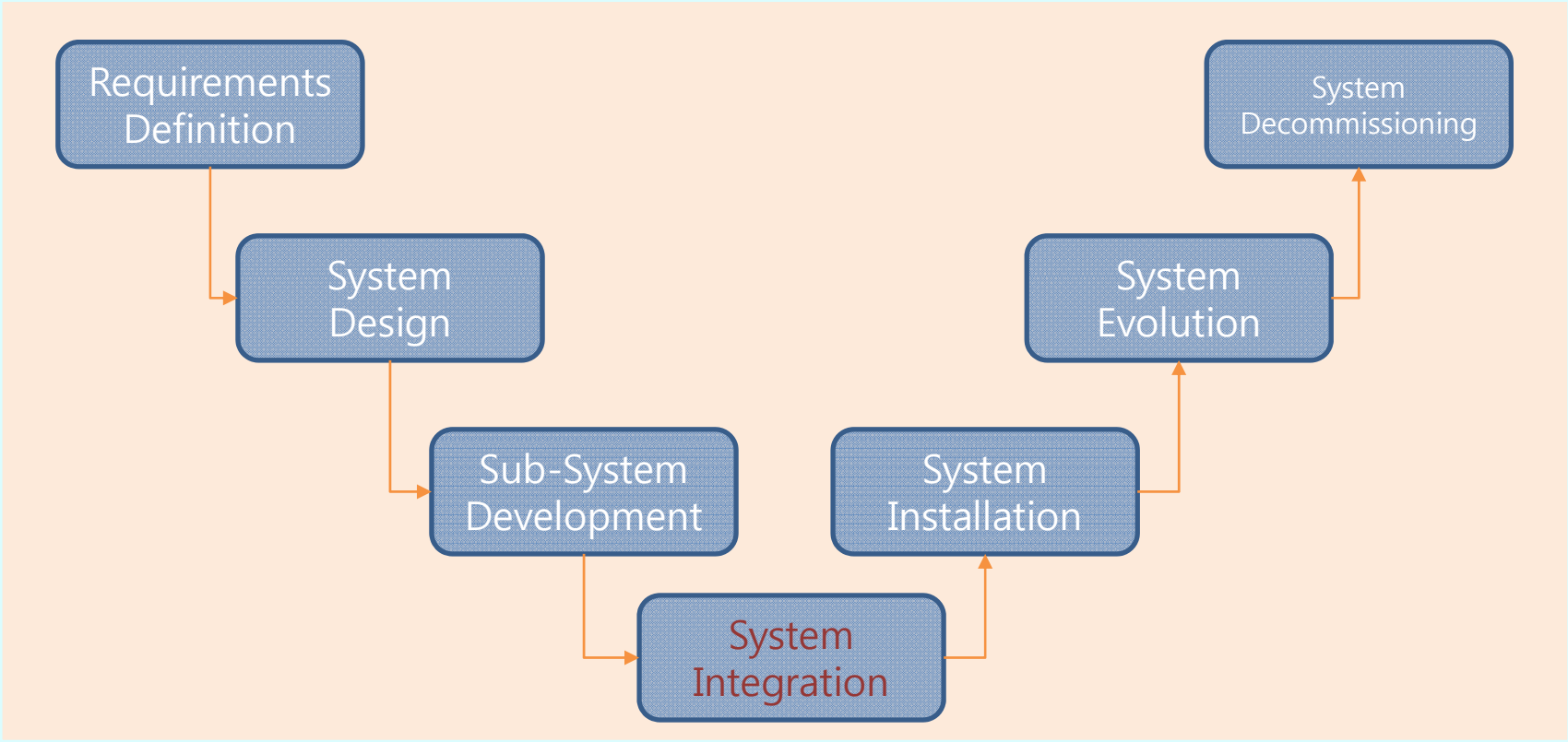
Systems Engineering

- System engineering is concerned with
  - specifying, designing, implementing, validating,
  - deploying and maintaining socio-technical systems

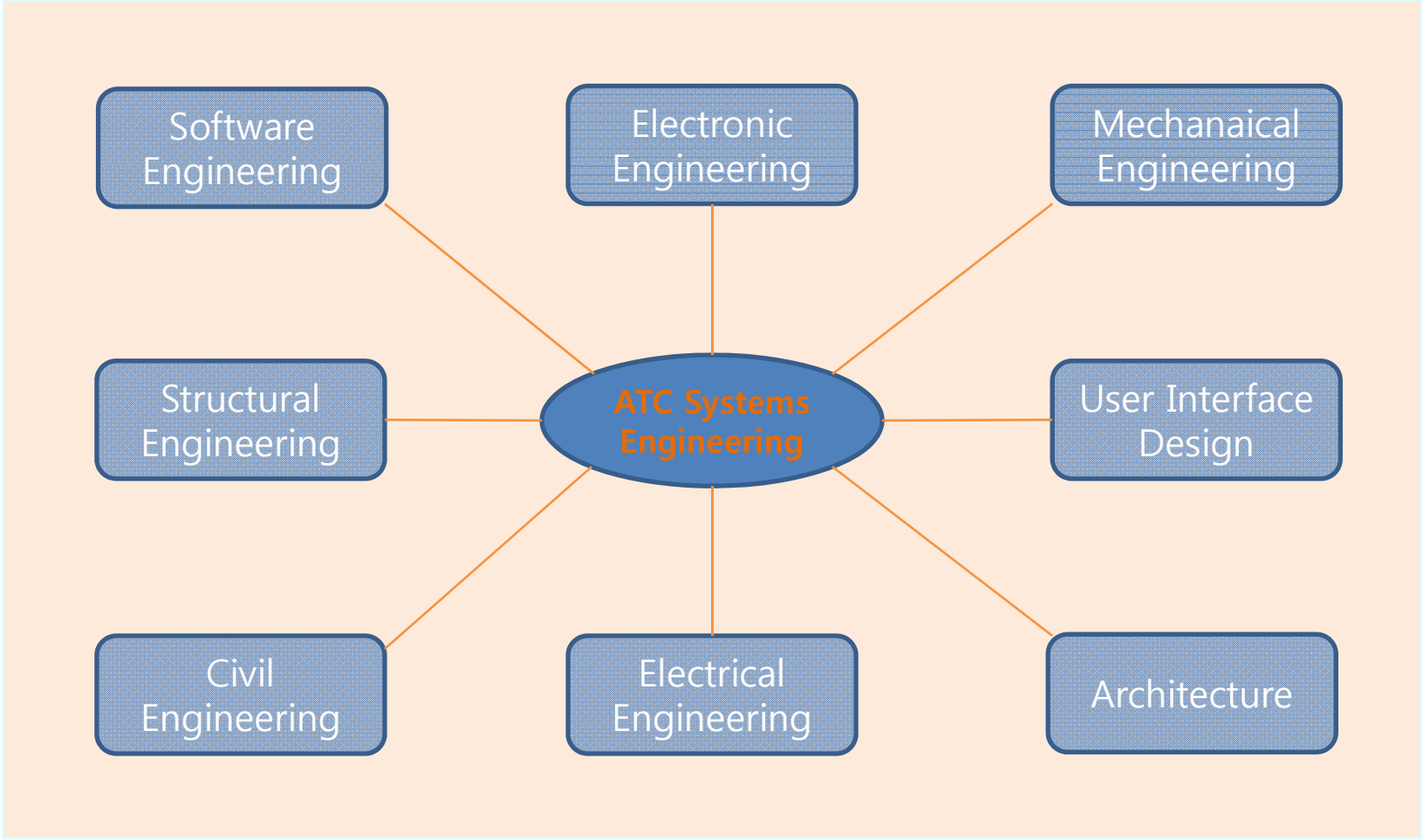
- System engineering is also concerned with
  - services provided by the system,
  - constraints on its construction,
  - operation and the ways in which it is used

Systems Engineering Process

- System engineering process
  - Usually follows a "Waterfall" model.
  - Involves engineers from different disciplines who must work together, and misunderstanding occurs here.



Example: Inter-Disciplinary Involvement in System engineering



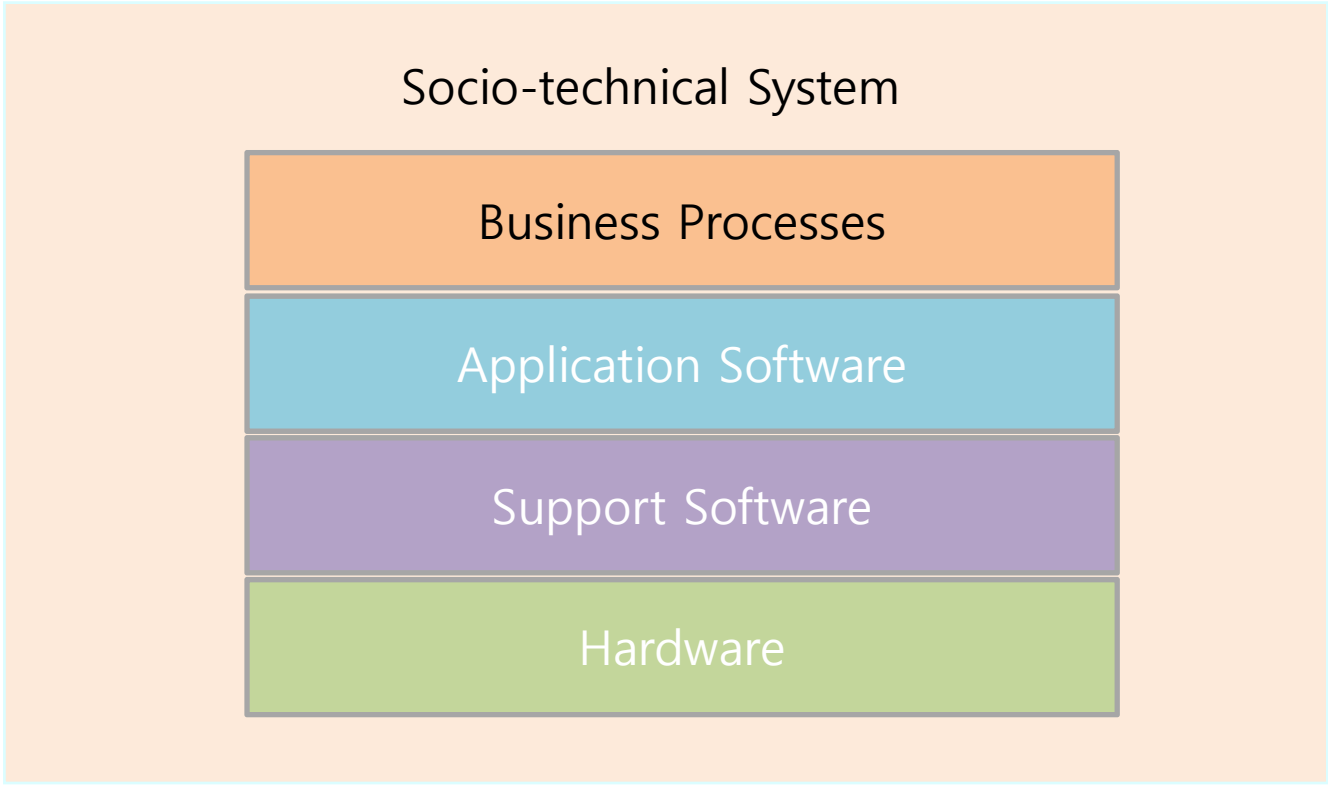
Legacy Systems

- Socio-technical systems that
  - Developed 10~20 years ago.
  - Have been in a stable manner up to now.
  - However, new business needs require a new efficient system.

- Crucial to the operation of a business
- Often too risky to change it with new ones
  - Bank customer accounting system
  - Aircraft maintenance system

- Legacy systems constrain new business processes and consume a high proportion of company budgets to maintain it..

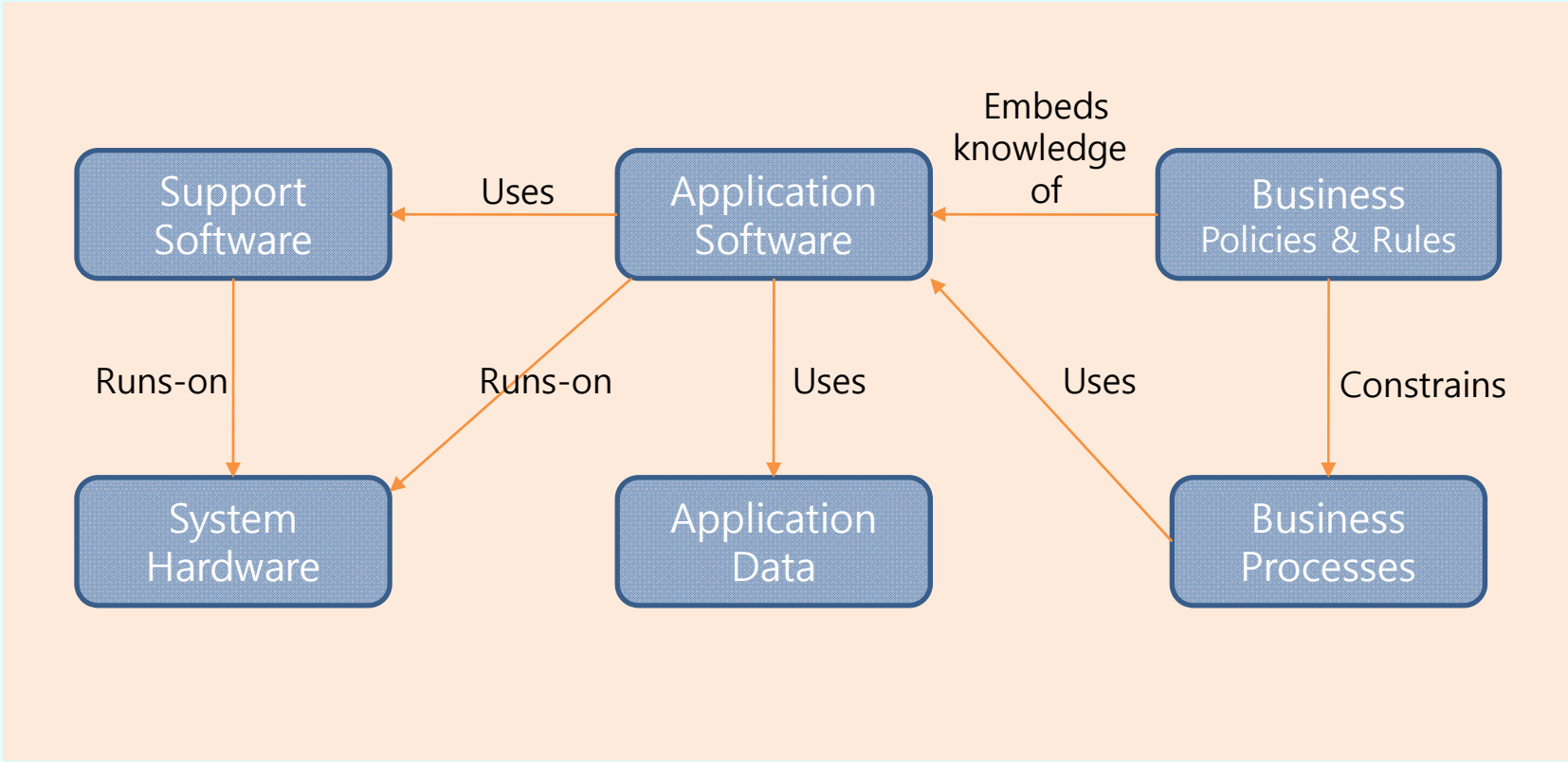
Socio-technical Legacy System



Legacy System Components

- Hardware
  - may be obsolete mainframe hardware.
- Support software
  - may rely on support software from suppliers who are no longer in business.
- Application software
  - may be written in obsolete programming languages.
- Application data
  - often incomplete and inconsistent.
- Business processes
  - may be constrained by software structure and functionality.
- Business policies and rules
  - may be implicit and embedded in the system software.

Relationship between Legacy System Components



Summary

- Socio-technical systems include computer hardware, software and people, and are designed to meet some business goal.
- Emergent properties are properties that are characteristic of the system as a whole.
- The systems engineering process includes specification, design, development, integration and testing. System integration is particularly critical.
- Human and organisational factors have a significant effect on the operation of socio-technical systems.
- A legacy system is an old system that continues to provide essential services.``

Chapter 3.

Critical Systems

Objectives

- To explain what a critical system is
- To explain four dimensions of dependability - availability, reliability, safety and security
- To explain why, for achieving dependability, you need to avoid mistakes, detect and remove errors and limit damage caused by failure (a mid-term problem)

Critical Systems

- Safety-critical systems
  - Failure results in loss of life, injury or damage to environment
  - Ex) Chemical plant protection system

- Mission-critical systems
  - Failure results in failure of some goal-directed activities
  - Ex) Spacecraft navigation system

- Business-critical systems
  - Failure results in high economic losses
  - Ex) Customer accounting system in bank

Development Methods for Critical Systems

- The costs of critical system failure are so high.
- Therefore, development methods for critical systems are not cost-effective for other types of system.

- Examples of development methods
  - Formal methods (specification and verification)
  - Static analysis
  - External quality assurance

System Dependability

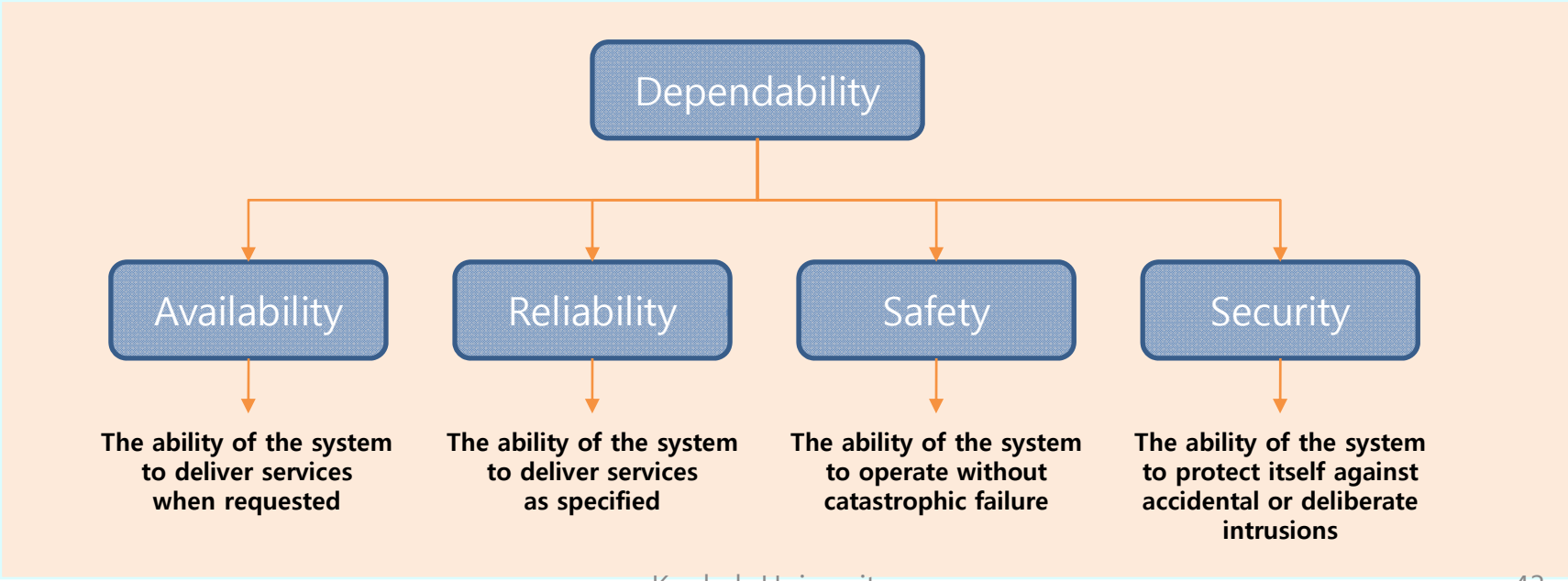
- For critical systems, it is usually the case that the most important system property is the dependability of the system.

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

Dependability

- Dependability of system equates to its trustworthiness.
- Dependable system is a system that is trusted by its users.

- Principal dimensions of dependability
  - Availability, Reliability, Safety, Security

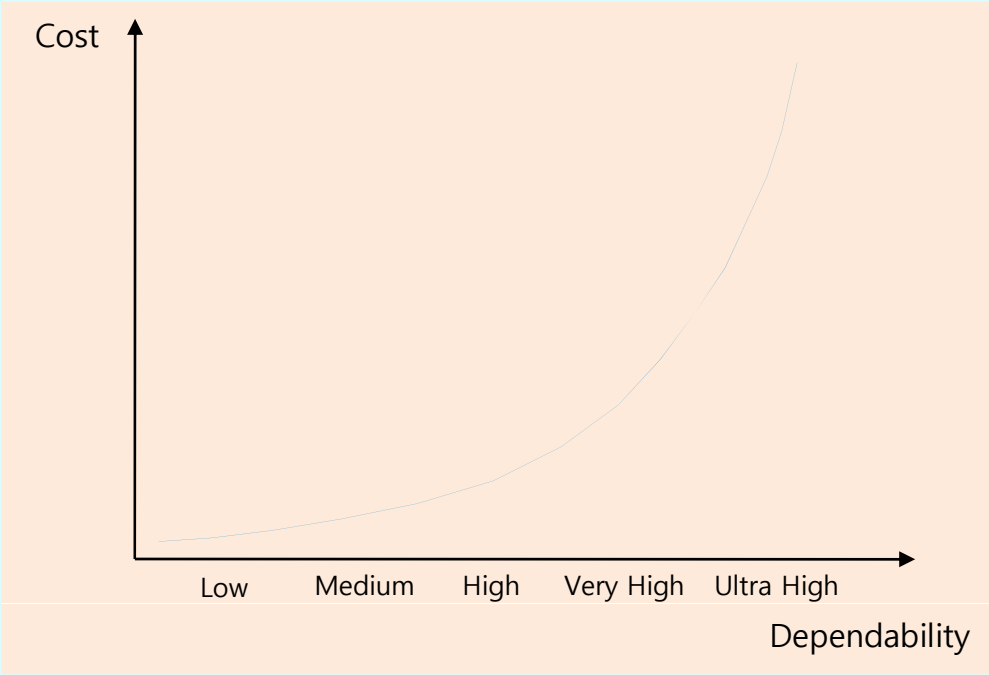


Other Dependability Properties

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

Dependability Costs

- Dependability costs tend to increase exponentially as required levels of dependability increase.
  - More expensive development techniques and hardware are required.
  - Increased testing and system validation are also required.



Dependability Economics

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

- However, it depends on
  - Social and political factors
    - Poor reputation for products may lose future business.
  - System type
    - For business systems, modest levels of dependability may be adequate.

Availability and Reliability

- Availability
  - The probability that a system will be operational and able to deliver the requested services, at a point in time
- Reliability
  - The probability of failure-free system operation over a specified time in a given environment for a given purpose
- Both of these attributes can be expressed quantitatively.
- This class considers them as the same.

Reliability Terminology

Term	Description
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users
System error	An erroneous system state that can lead to system behavior that is unexpected by system users.
System fault	A characteristic of a software system that can lead to a system error. For example, failure to initialize a variable could lead to that variable having the wrong value when it is used.
Human error or mistake	Human behavior that results in the introduction of faults into a system.

Reliability Achievement

- Fault avoidance
  - Use development technique which either minimize the possibility of mistakes or trap mistakes before they result in the introduction of system faults
- Fault detection and removal
  - Use verification and validation techniques which increase probability of detecting and correcting errors before system goes into service
- Fault tolerance
  - Use run-time techniques to ensure that system faults do not result in system errors and/or to ensure that system errors do not lead to system failures

Safety

- Safety is a system property that reflects the system's ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system's environment.

- Exclusive requirements
  - Exclude undesirable situations rather than specify required system services.
  - "Should not" property

Safety Terminology

Term	Description
Accident (mishap)	An unplanned event or sequence of events which results in human death or injury, damage to property or to the environment. A computer-controlled machine injuring its operator is an example of an accident.
Risk	This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity and the probability that a hazard will result in an accident.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people killed as a result of an accident to minor injury or property damage.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that detects an obstacle in front of a machine is an example of a hazard.
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic where many people are killed to minor where only minor damage results.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from probable (say 1/100 chance of a hazard occurring) to implausible (no conceivable situations are likely where the hazard could occur).

Safety Achievement

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

Security

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

Security Terminology

Term	Description
Exposure	Possible loss or harm in a computing system. This can be loss or damage to data or can be a loss of time and effort if recovery is necessary after a security breach.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.
Attack	An exploitation of a system vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
Threats	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack.
Control	A protective measure to reduce a system vulnerability. Encryption would be an example of a control that reduced a vulnerability of a weak access control system.

Security Assurance

- Vulnerability avoidance
  - Design the system so that vulnerabilities do not occur
  - For example, if there is no external network connection, any external attack is impossible.

- Attack detection and elimination
  - Design the system so that attacks on vulnerabilities are detected and neutralised before they result in an exposure
  - For example, virus checkers find and remove viruses before they infect a system.

- Exposure limitation
  - Design the system so that the adverse consequences of a successful attack are minimized
  - For example, a backup policy allows damaged information to be restored.

Summary

- A critical system is a system where failure can lead to high economic loss, physical damage or threats to life.
- Dependability of a system reflects user's trust in that system.
- Availability is the probability that it will be available to deliver services when requested.
- Reliability is the probability that system services will be delivered as specified.
- Safety is a system attribute that reflects the system's ability to operate without threatening people or the environment.
- Security is a system attribute that reflects the system's ability to protect itself from external attack.

Chapter 4.

Software Processes

Objectives

- To introduce software process models
- To describe three generic process models
- To describe common process activities
- To explain the Rational Unified Process(RUP) model

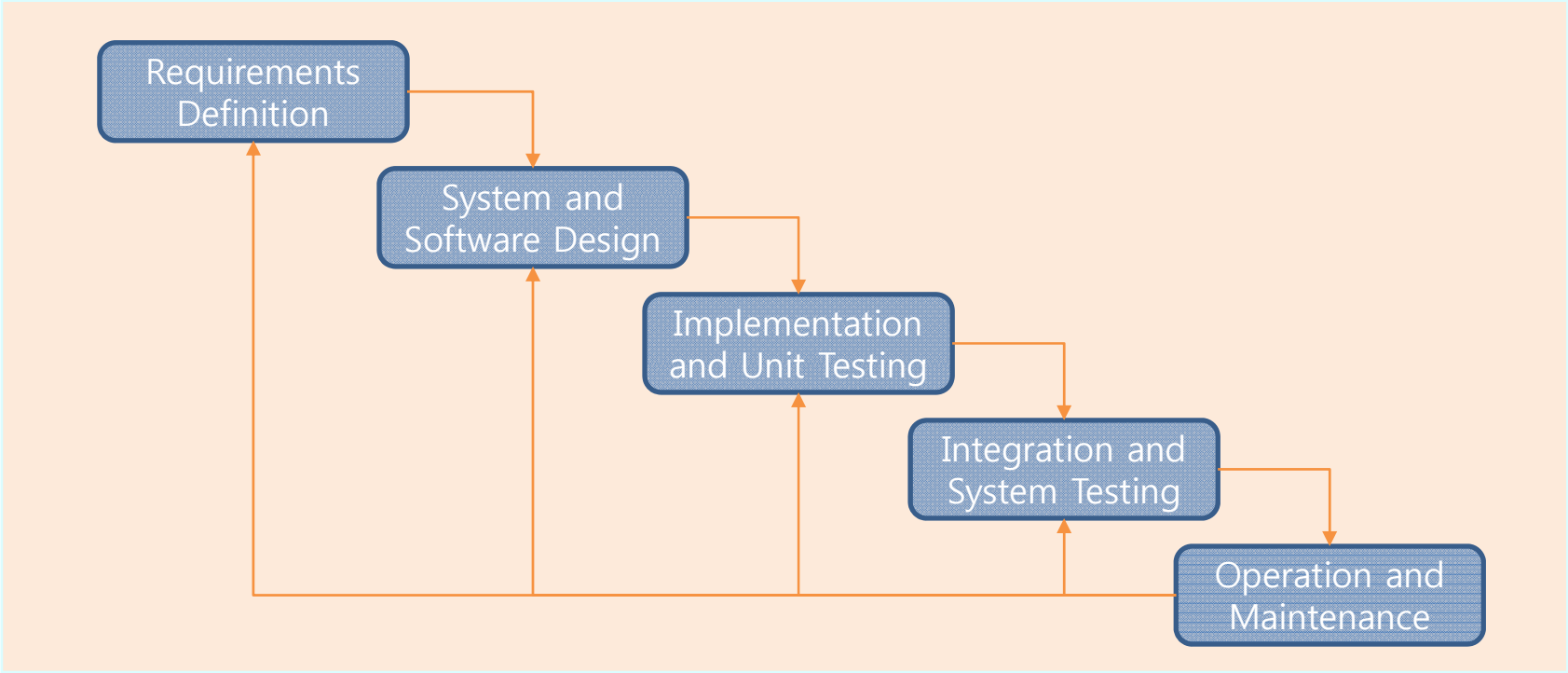
Software Process

- A structured set of activities required to develop a software system
  - Specification
  - Design
  - Validation
  - Evolution

- A software process model is an abstract representation of a process.
  - Waterfall model
  - Evolutionary development
  - Component-based software engineering
  - Many variants

Waterfall Model

- A classic life cycle model
  - Suggests a systematic, sequential approach to software development
  - The oldest paradigm
  - Separate and distinct phases of specification and development
  - Useful in situations where
    - requirements are fixed and work is to proceed to completion in a linear manner



Waterfall Model

- Inflexible partitioning of project into distinct stages makes it difficult to respond to changing customer requirements.

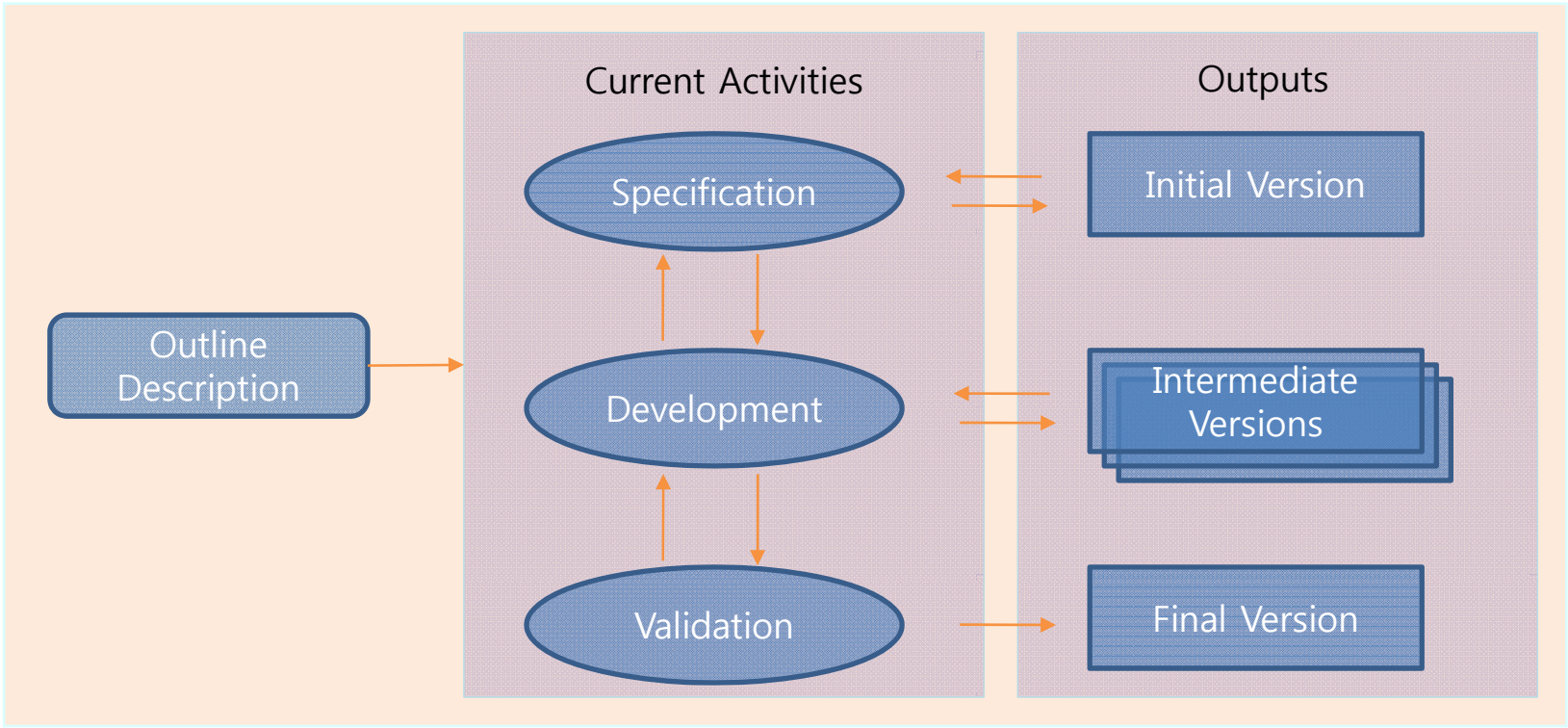
- Therefore, it is only appropriate when
  - Requirements are well-understood.
  - Changes will be fairly limited during design process.

- Waterfall model is mostly used for large system engineering projects where a system is developed at several sites.

- However, few business systems have stable requirements.

Evolutionary Development

- Exploratory development
  - Evolve a final system from an initial outline specification to work with customers.
  - Start with well-understood requirements and add new features as proposed by the customer.



Evolutionary Development

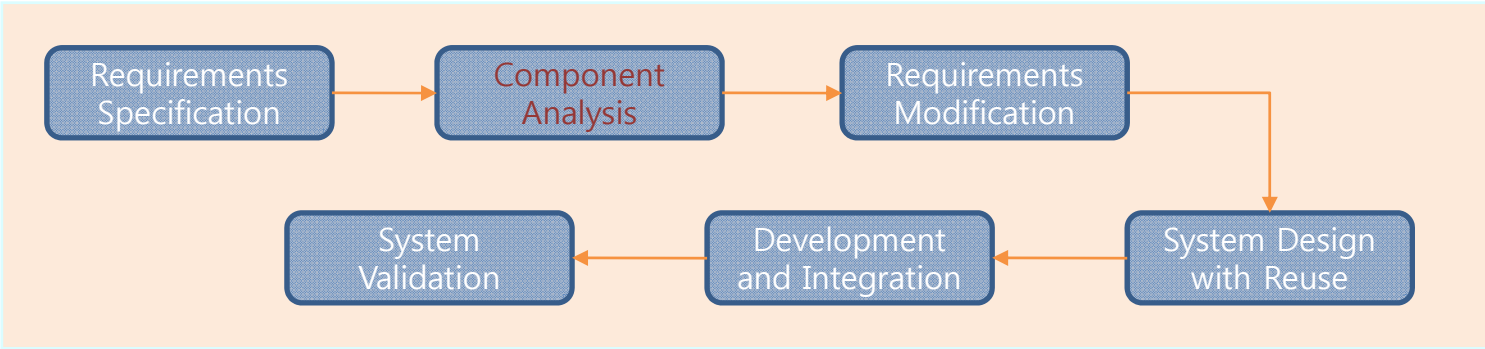
- Problems
  - Lack of process visibility
  - Systems are often poorly structured.
  - Special skills (e.g. in languages for rapid prototyping) may be required.

- Applicability
  - For small or medium-size interactive systems
  - For parts of large systems (e.g. the user interface)
  - For short-lifetime systems

Component-Based Software Engineering

- Systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- Based on systematic reuse

- Process stages



Process Iteration

- System requirements always evolve in the course of a project.
- Process iteration itself is often a part of the process for large systems.

- Iteration can be applied to any of the generic process models.

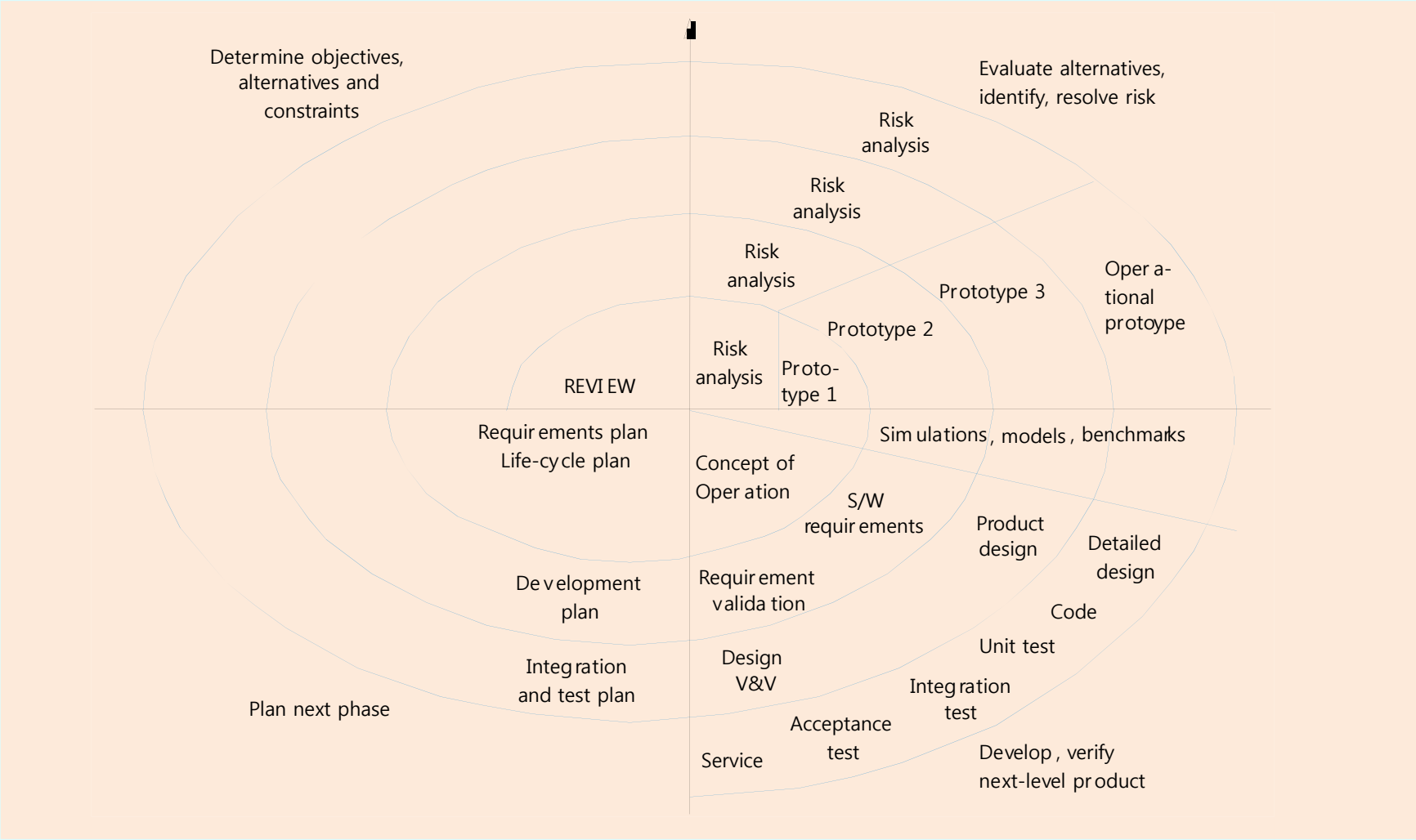
- Two (related) approaches
  - Incremental delivery (→ evolutionary development)
  - Spiral development

Spiral Development

- Represented as a spiral.
- No fixed phases - loops in the spiral are chosen depending on what is required.
- Risks are explicitly assessed and resolved throughout the process.

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

Spiral Development Model



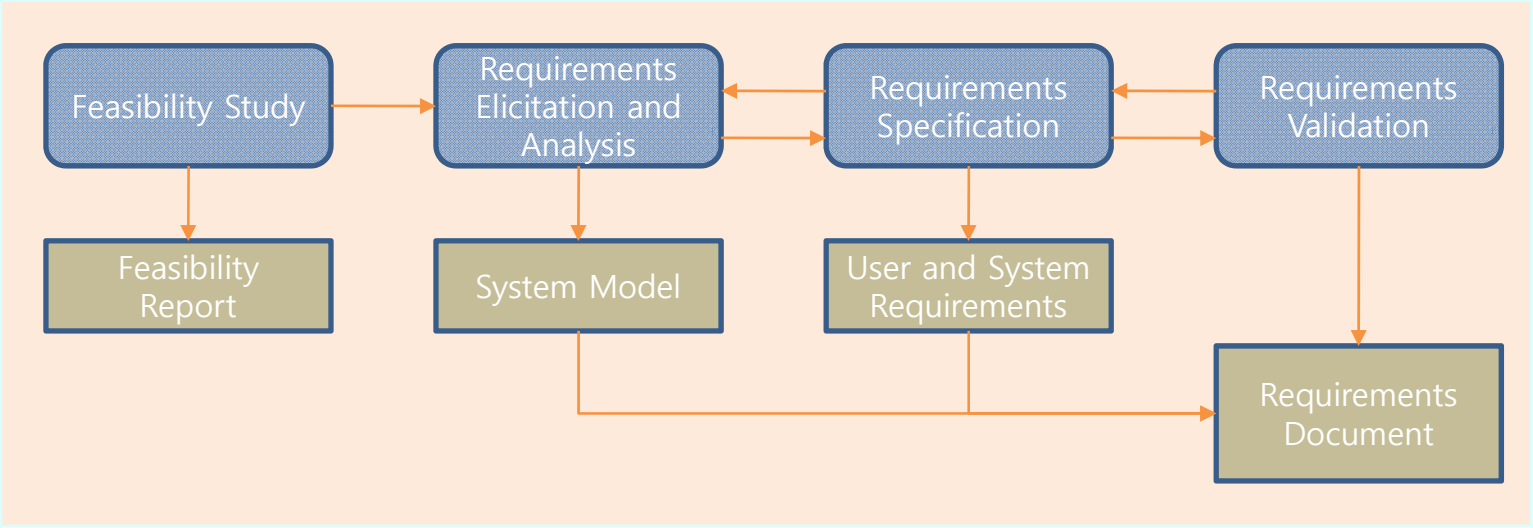
4 Common Process Activities

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

1. Software Specification

- Process of establishing
  - What services are required and
  - Constraints on the system's operation and development
  - Called "Requirements Engineering"

- Requirements engineering process

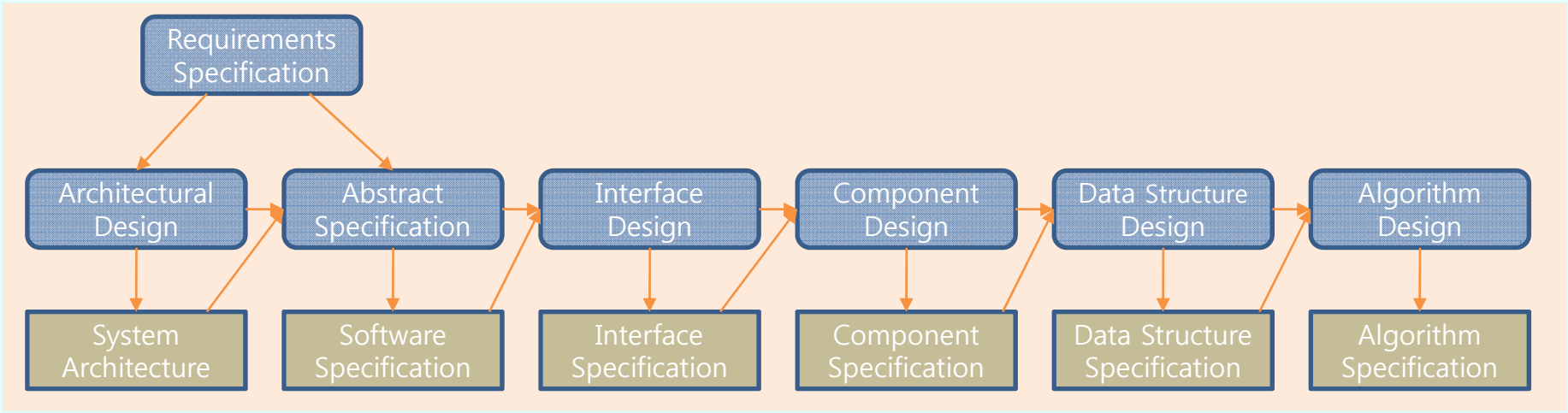


2. Software Design and Implementation

- Process of converting system specification into executable system.

- Software design
  - Design a software structure to realize the specification
- Implementation
  - Translate the design structure into an executable program.
  - Programming is a personal activity. No generic programming process.

- Software design process



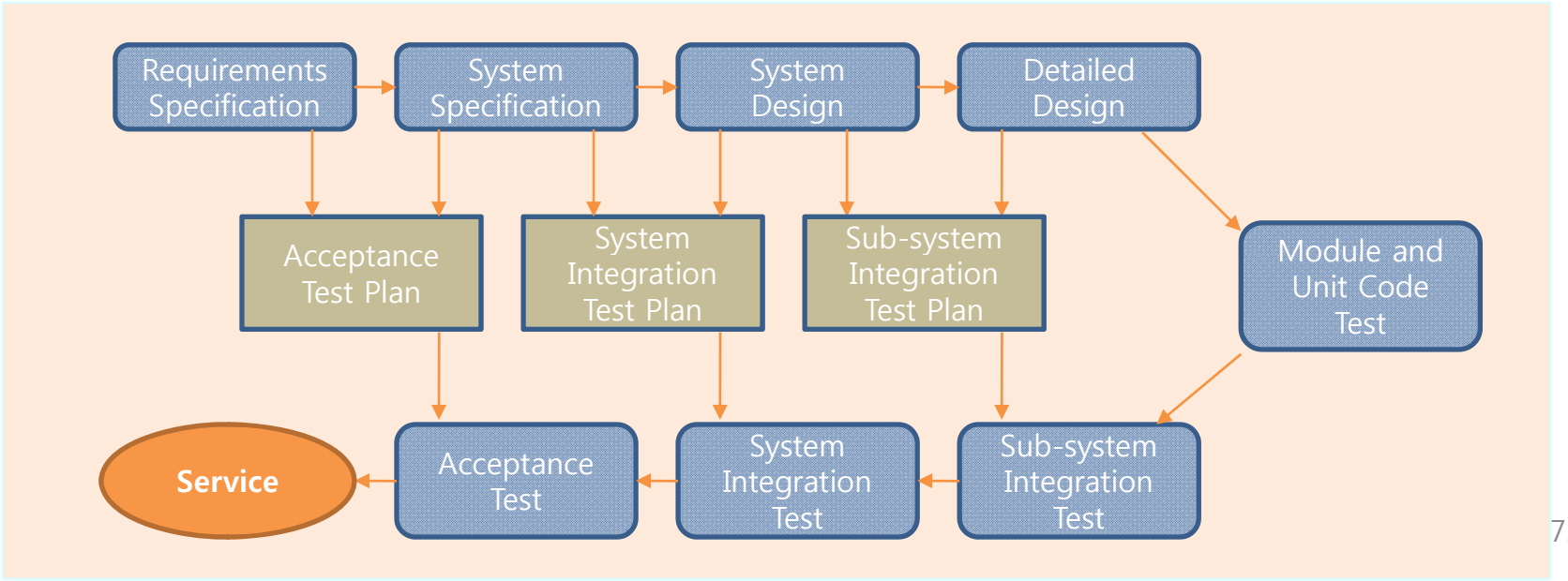
3. Software Validation

- Verification and validation (V & V) is intended to show that
  - System conforms to its specification.
  - System meets requirements of the system customer.
  - Involves
    - Checking (Formal/Informal)
    - Review processes
    - System testing

- System testing involves executing the system with test cases derived from its specification.

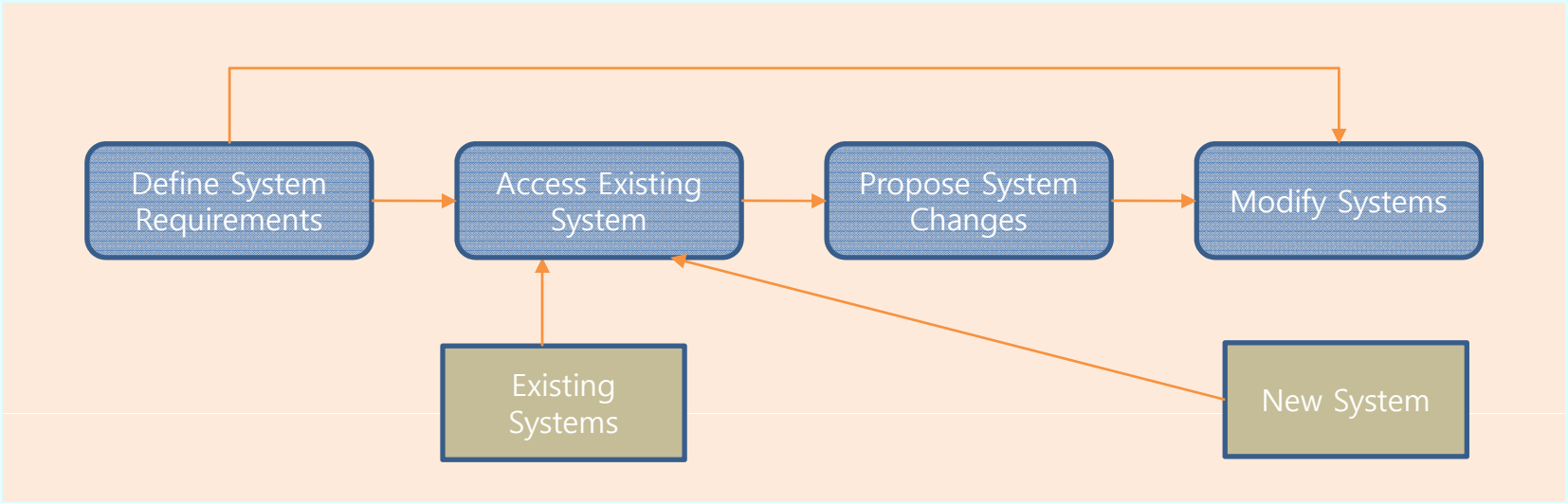
Testing Stages and Phases

- Unit or Component testing
  - Individual components are tested independently.
- System testing
  - Testing of the system as a whole.
- Acceptance testing
  - Testing with customer data to check whether the system meets the customer's needs.



4. Software Evolution

- Software is inherently flexible and can change.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.

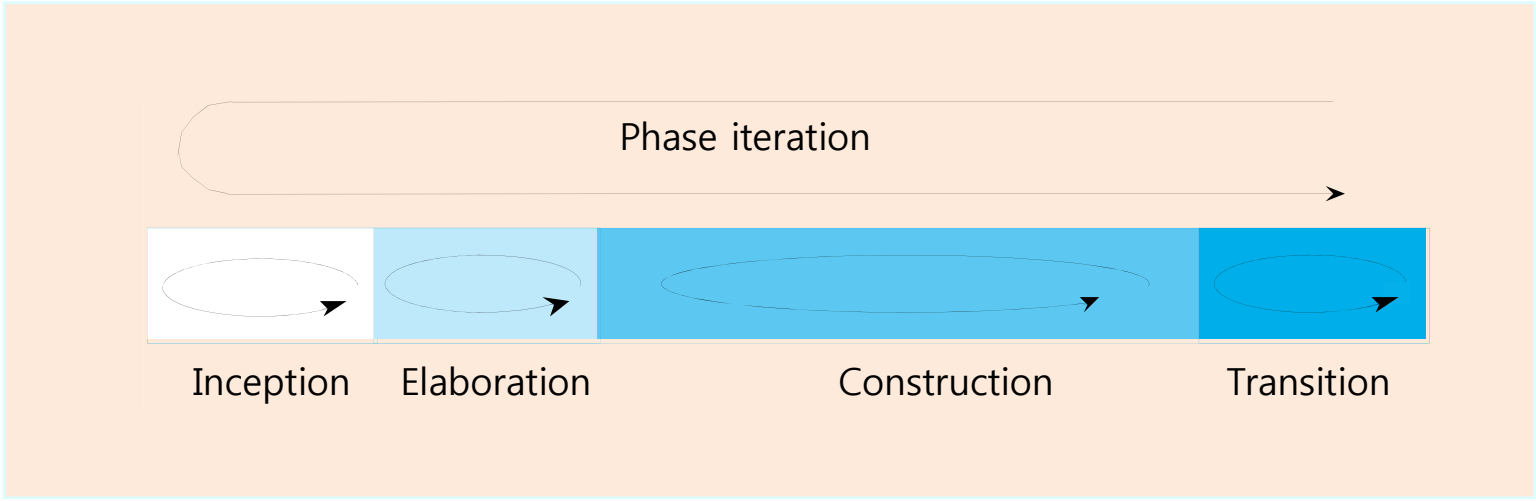


The Rational Unified Process

- A modern process model
  - Derived from working groups on the UML

- Normally described from 3 perspectives
  - Dynamic perspective : shows phases over time
  - Static perspective : shows process activities
  - Practice perspective : suggests good practice.

4 Phases of RUP (Dynamic Perspective)



Workflows(Activities) of the RUP (Static Perspective)

Workflow	Description
Business Modeling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and Design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Test	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and Change Management	This supporting workflow managed changes to the system (see Chapter 29).
Project Management	This supporting workflow manages the system development (see Chapter 5).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

RUP Good Practice (Practice Perspective)

- Suggestions:
  - Develop software iteratively
  - Manage requirements
  - Use component-based architectures
  - Model software Visually
  - Verify software quality
  - Control changes to software

- More than 1,00 best practices.

Summary

- Software processes are the activities involved in producing and evolving a software system.
- Software process models are abstract representations of these processes.
- Generic process models describe organization of software processes. Examples include the waterfall model, evolutionary development and component-based software engineering.
- General software process activities are specification, design and implementation, validation and evolution.
- The Rational Unified Process is a generic process model based on UML.

Chapter 5.

Project Management

Objectives

- To explain main tasks undertaken by project managers
- To introduce software project management and to describe its distinctive characteristics
- To discuss project planning and planning process
- To discuss the notion of risks and risk management process

Software Project Management

- Concerned with activities involved in ensuring that software is delivered
  - on time and
  - on schedule and
  - in accordance with the requirements of the organizations developing and procuring the software.

- Needed because software development is always subject to budget and schedule constraints that are set by the organization developing the software.

Project Management Activities

- Proposal writing
- Project staffing
- Project planning and scheduling
- Project costing
- Project monitoring and reviews
- Personnel selection and evaluation
- Report writing and presentations

Project Staffing

- May not be possible to appoint ideal people to work on a project
  - Project budget may not allow for the use of highly-paid staff.
  - Staff with appropriate experience may not be available.
  - Organization may wish to develop employee skills through performing software projects.

- Managers have to work within these constraints especially when there are shortages of trained staff.

Project Planning

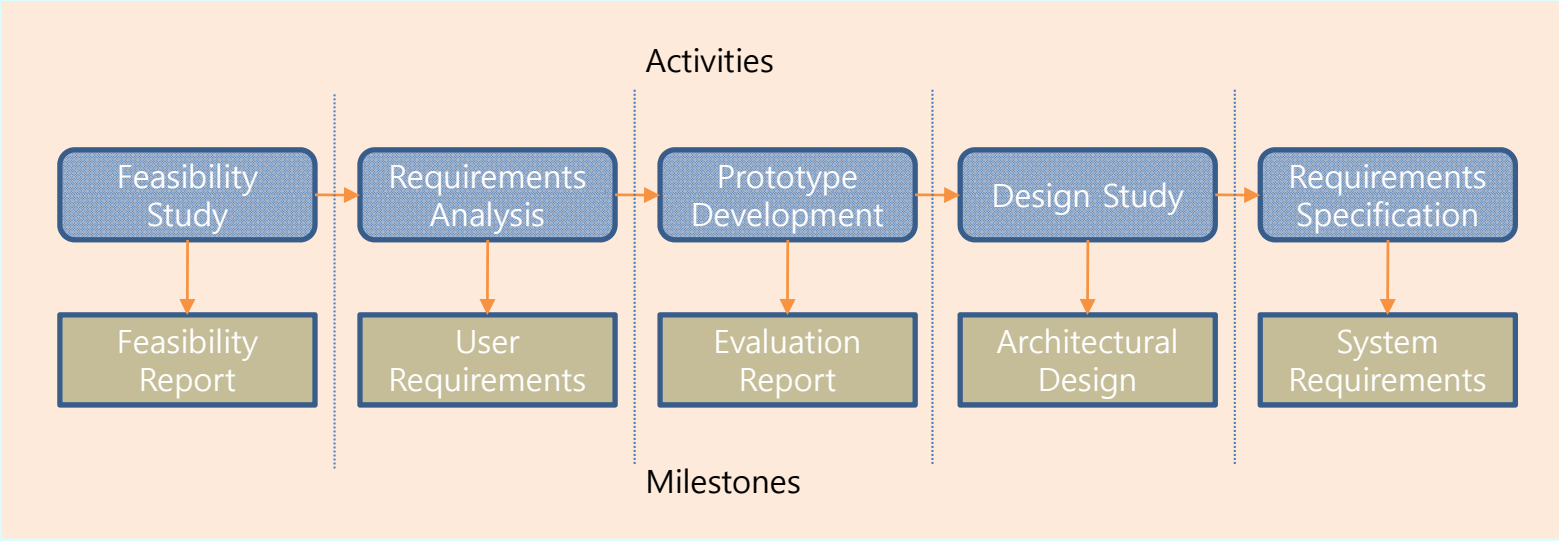
- Probably the most time-consuming project management activity
  - Continuous activity from initial concept through to system delivery
  - Plans must be regularly revised as new information becomes available.
- Various different types of plan may be developed to support main software project plan.

Plan	Description
Quality Plan	Describes the quality procedures and standards that will be used in a project. See Chapter 27.
Validation Plan	Describes the approach, resources and schedule used for system validation. See Chapter 22.
Configuration Management Plan	Describes the configuration management procedures and structures to be used. See Chapter 29.
Maintenance Plan	Predicts the maintenance requirements of the system, maintenance costs and effort required. See Chapter 21.
Staff Development Plan	Describes how the skills and experience of the project team members will be developed. See Chapter 25.

Project Planning Process

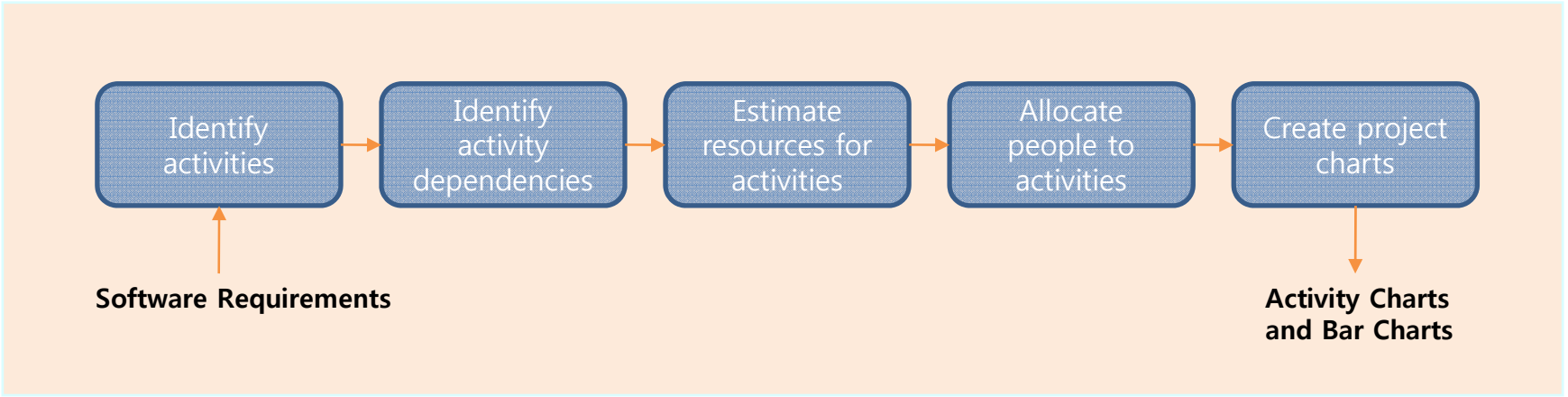
- Activities: produce tangible outputs for management to judge progress
- Milestones : end-point of a process activity
- Deliverables : project results delivered to customers

- Waterfall process allows straightforward definition of progress milestones.

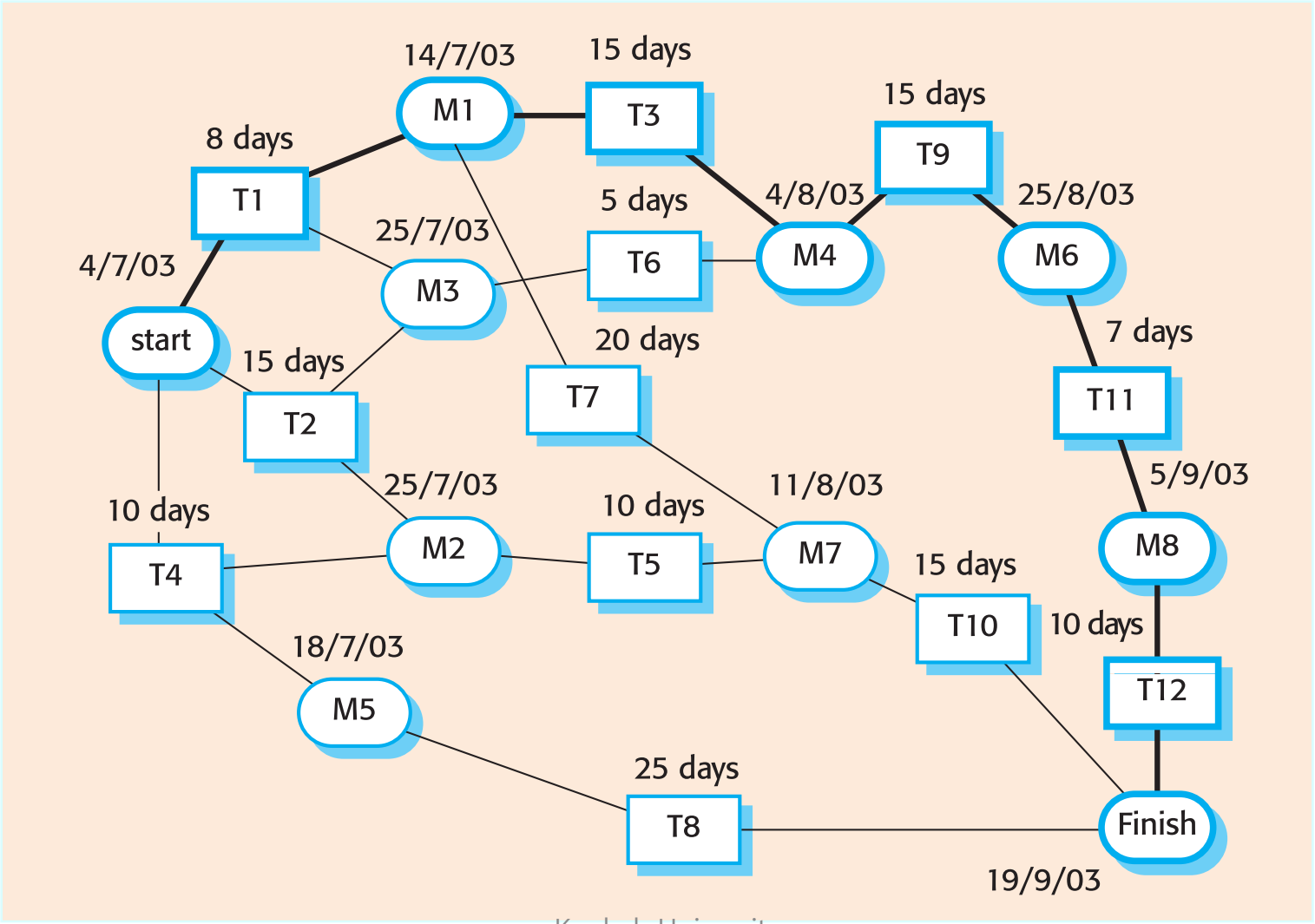


Project Scheduling Process

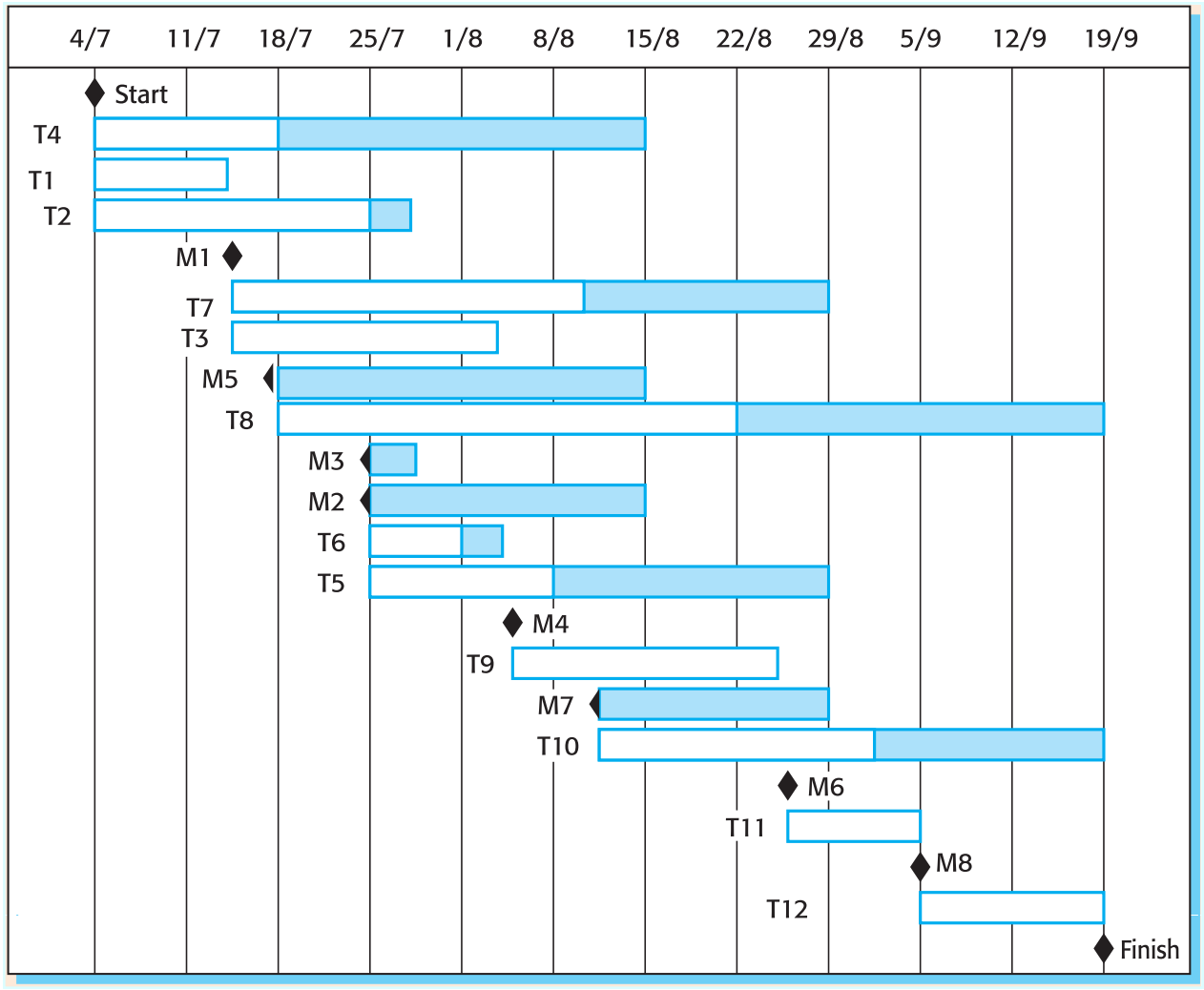
- Split project into tasks and estimate time and resources required to complete each task.
  - Organize tasks concurrently to make optimal use of workforce.
  - Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- Depend on project manager's intuition and experience.



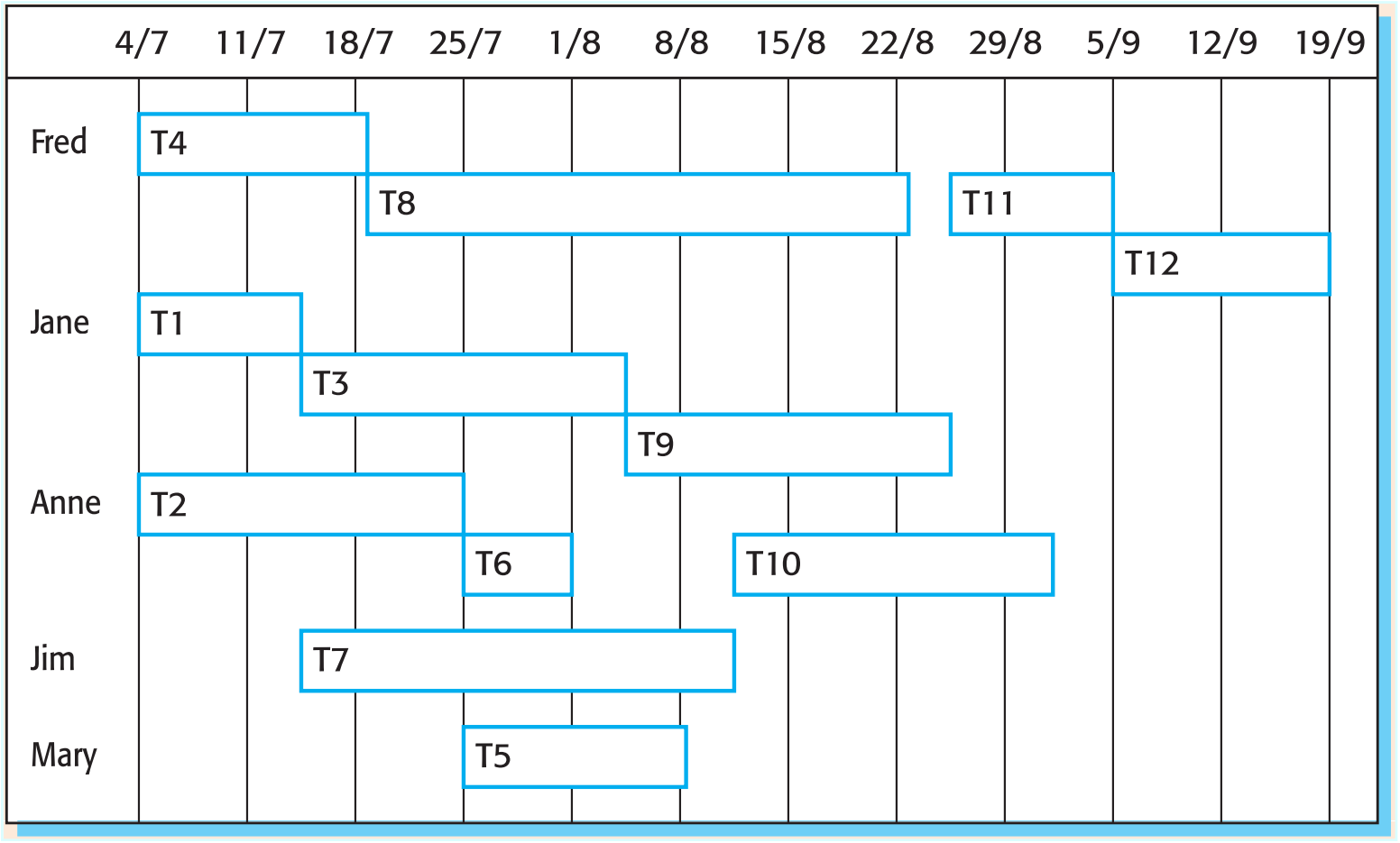
Activity Network



Activity Timeline



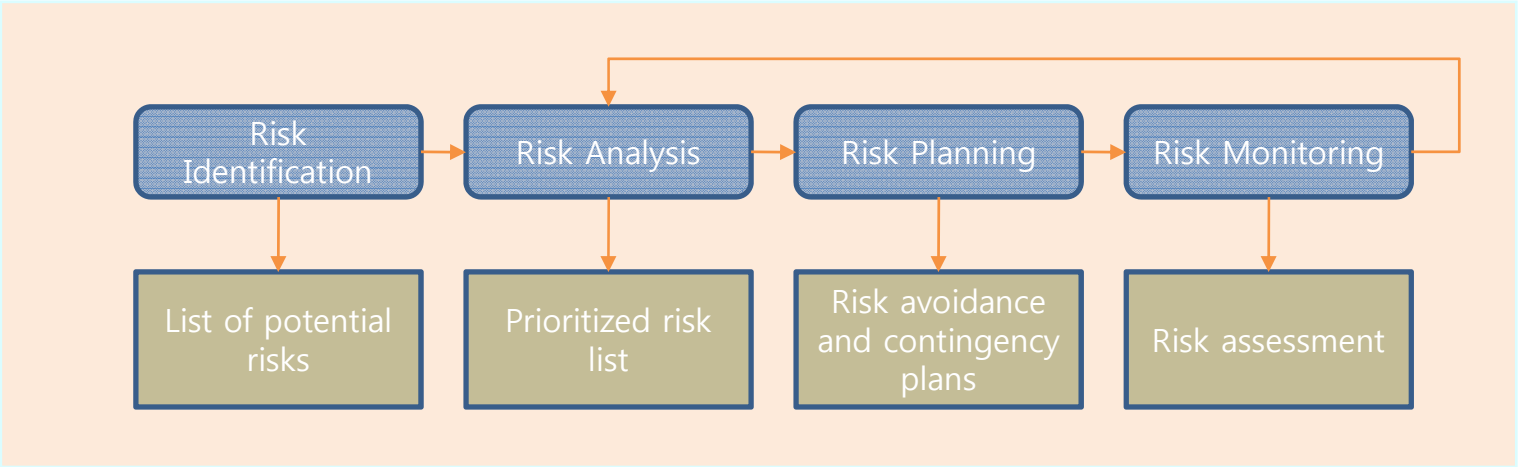
Staff Allocation



Risk Management

- Concerned with identifying risks and drawing up plans to minimize their effect on a project.
- A risk is a probability that some adverse circumstance will occur
  - Project risk affects schedule or resources.
  - Product risk affects quality or performance of the software being developed.
  - Business risk affects the organization developing or procuring the software.

- Risk management process



Summary

- Good project management is essential for project success.
- Managers have diverse roles but their most significant activities are planning, estimating and scheduling.
- Project scheduling involves preparing various graphical representations showing project activities, their durations and staffing.
- Risk management is concerned with identifying risks which may affect the project, and planning to ensure that these risks do not develop into major threats.

Part II. Requirements

Chapter 6.

Software Requirements

Objectives

- To introduce concepts of user and system requirements
- To describe functional and non-functional requirements
- To explain how software requirements may be organized in a requirements document

Requirements Engineering

- Requirements engineering is the process of establishing
  - the services that the customer requires from a system
  - the constraints under which it operates and is developed

- The requirements are the descriptions of the system services and constraints that are generated during the requirements engineering process.

Requirements

- Range from a high-level abstract statement of service or system constraint to detailed mathematical functional specification.

- Types of requirements
  - User requirements
    - Statements in natural language, diagrams of the services the system provides and its operational constraints
    - Written for customers
    - Defined.
  - System requirements
    - Structured document setting out detailed descriptions of the system's functions, services and operational constraints.
    - Define what should be implemented
    - May be part of a contract between clients and contractors
    - Specified.

Requirements Definitions and Specifications

User Requirement Definition

1. The software must provide a means of representing and accessing external files created by other tools.

System Requirement Specification

1. The user should be provided with facilities to define the type of external files.
2. Each external file type may have an associated tool which may be applied to the file.
3. Each external file type may be represented as a specific icon on the user's display.
4. Facilities should be provided for the icon representing an external file type to be defined by the user.
5. When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

Functional vs. Non-Functional Requirements

- Functional requirements
  - Statements of services which the system should provide
  - How the system should react to particular inputs
  - How the system should behave in particular situations

- Non-functional requirements
  - Constraints on the services or functions offered by the system
    - timing constraints
    - constraints on the development process
    - Standards

- Domain requirements
  - Requirements that come from the application domain of the system
  - Reflect characteristics of the target domain
  - May be functional or non-functional or the both

Example: LIBSYS System

- System description: A LIBSYS library system
  - Provides a single interface to a number of databases of articles in different libraries
  - Users can search for, download, and print these articles for personal study.

- Function requirements
  - The user shall be able to search either all of the initial set of databases or select a subset from it.
  - The system shall provide appropriate viewers for the user to read documents in the document store.
  - Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.

Requirements Completeness and Consistency

- Problems arise when requirements are not precisely stated.
  - Ambiguous requirements may be interpreted in different ways.

- 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, it is impossible to produce a complete and consistent requirements document with natural languages.

Non-Functional Requirements

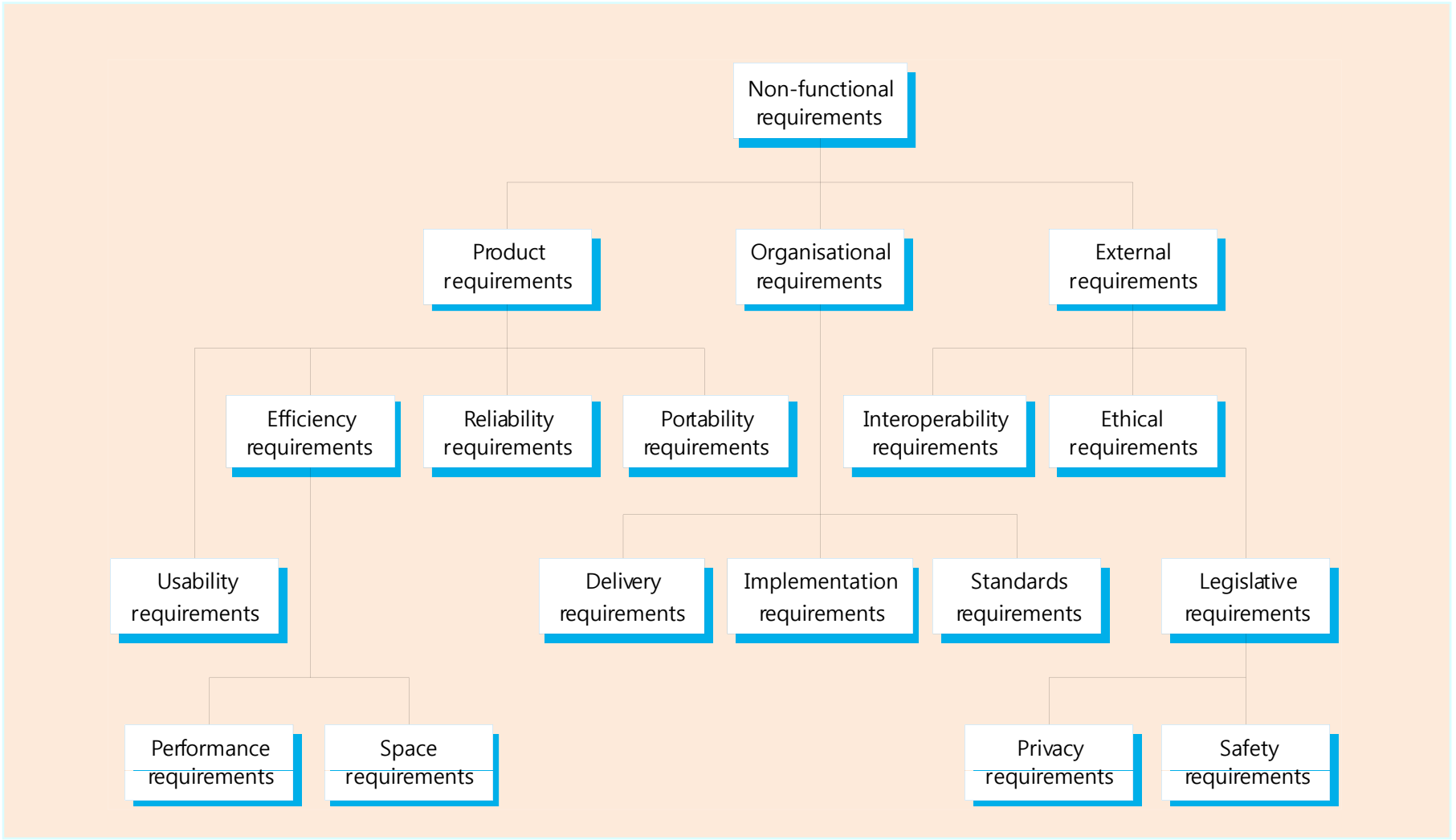
- Define system properties and constraints
  - Reliability
  - Response time
  - Storage requirements
  - Constraints on I/O device capability
  - System representations
  - Etc.

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

Classification of Non-Functional Requirements

- Three types of non-functional requirements
  - Product requirements
    - 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 organizational policies and procedures
    - e.g. process standards, implementation requirements, etc.
  - External requirements
    - Requirements which arise from the factors external to the development process
    - e.g. interoperability requirements, legislative requirements, etc.

Non-Functional Requirement Types



Examples of Non-Functional Requirements

- Product requirement
  - 8.1 The user interface for LIBSYS shall be implemented as simple HTML without frames or Java applets.
- Organisational requirement
  - 9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.
- External requirement
  - 7.6.5 The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Goals and Requirements

- Non-functional requirements may be very difficult to state precisely.
  - Imprecise requirements may be also difficult to verify.
  - Write a "goal" first → transform into " verifiable non-functional requirements"

- Goal
  - A general intention of the user, (e.g. ease of use)
  - "The system should be easy to use by experienced controllers and should be organized in such a way that user errors are minimized."

- Verifiable non-functional requirement
  - A statement using some measure that can be tested objectively
  - "Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day."

Domain Requirements

- Describe system characteristics and features of the target domain
  - Derived from the application domain

- Domain requirements may be
  - new functional requirements
  - constraints on existing requirements
  - definition of specific computations

- If domain requirements are not satisfied, the system may be unworkable.

Domain Requirements Example : LIBSYS

- There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.
- Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer.

Problems with Natural Language Specification

- Ambiguity
  - Readers and writers of the requirement must interpret the same words in the same way.
  - Natural language is naturally ambiguous.
- Over-flexibility
  - The same thing may be said in a number of different ways in the specification.
- Lack of modularisation
  - NL structures are inadequate to structure system requirements.
- Alternatives to natural language specifications
  - Structural language specification
  - Graphical notations
  - Design description language
  - Mathematical specifications

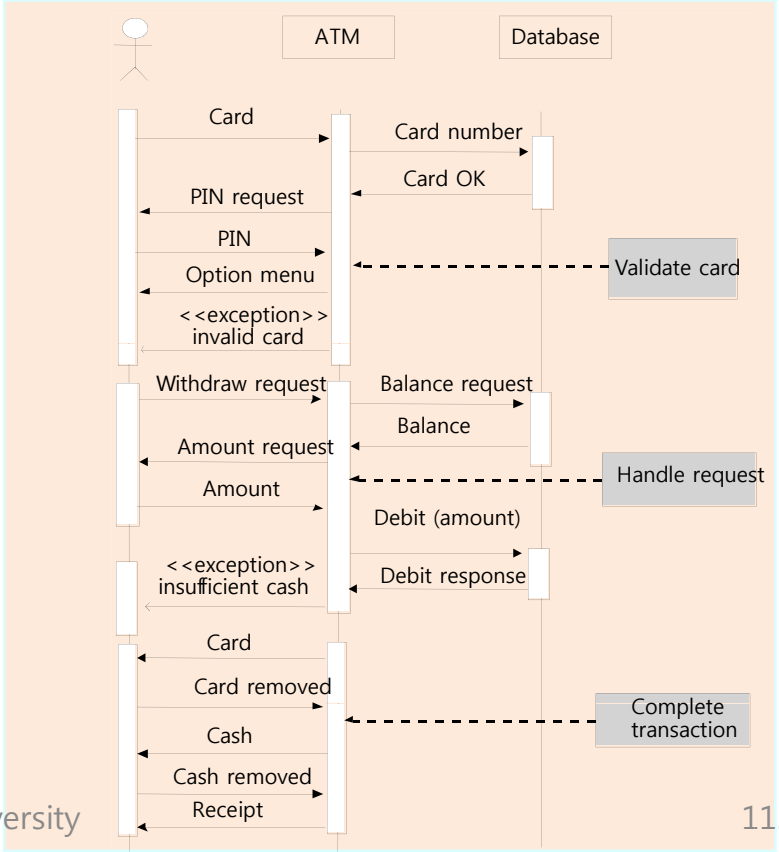
Structured Language Specifications

- The freedom of writing requirements is limited by a predefined template.
- Form-based specifications

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 Š 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.
Requires	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

Graphical Notations

- Graphical notation is useful
  - when you need to show how state changes
  - where you need to describe a sequence of actions.
- Different graphical models are explained in Chapter 8.
- Sequence diagram (ATM example) :



Interface Specification

- Most systems must operate with other systems.
- Operating interfaces must be specified as part of the requirements.
  - Procedural interfaces
  - Data structures that are exchanged
  - Data representations

- Formal notations are an effective technique for interface specification.

```
interface PrintServer {

// defines an abstract printer server
// requires:            interface Printer, interface PrintDoc
// provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter

            void initialize ( Printer p ) ;
            void print ( Printer p, PrintDoc d ) ;
            void displayPrintQueue ( Printer p ) ;
            void cancelPrintJob (Printer p, PrintDoc d) ;
            void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
```

Requirements Document

- Requirements document is an official statement of what is required of the system developers.
  - Should include both user requirements and system requirements
  - Should be a set of WHAT the system should do rather than HOW it should do it

- IEEE standard on requirements document
  - Introduction
  - General description
  - Specific requirements
  - Appendices
  - Index

- Preface
- Introduction
- Glossary
- User requirements definition
- System architecture
- System requirements specification
- System models
- System evolution
- Appendices
- Index

Summary

- Requirements set out what the system should do and define constraints on its operation and implementation.
- Functional requirements set out services the system should provide.
- Non-functional requirements constrain the system being developed or the development process.
- User requirements are high-level statements of what the system should do.
- System requirements are intended to communicate the functions that the system should provide.
- A software requirements document is an agreed statement of the system requirements.
- The IEEE standard is a useful starting point for defining more detailed specific requirements standards.

Chapter 7.

Requirements Engineering Processes

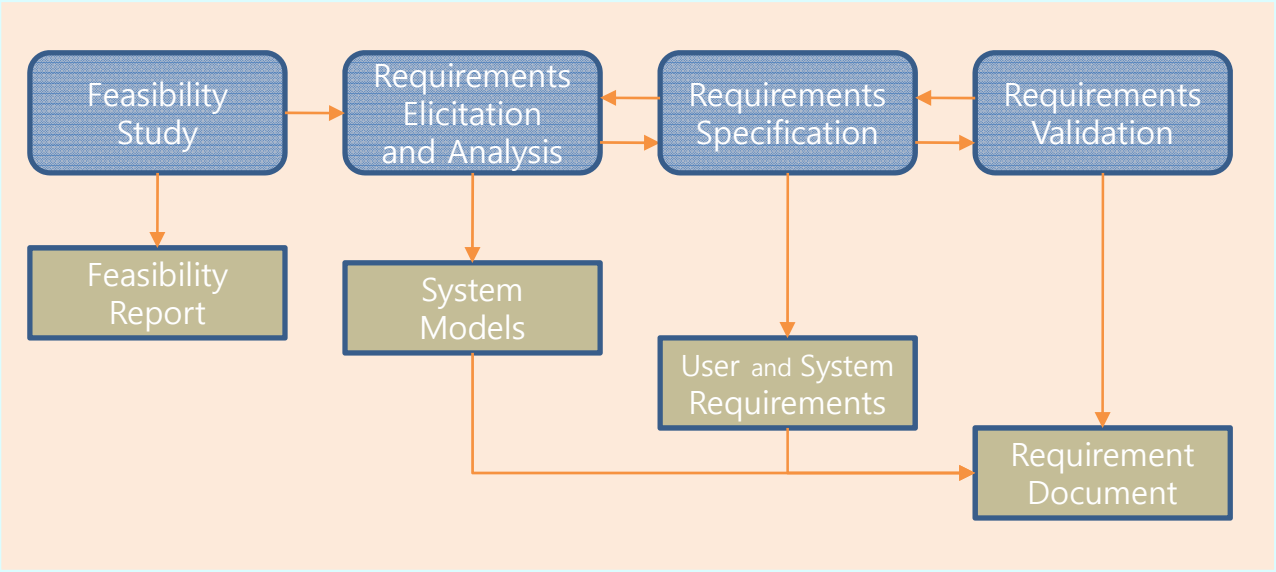
Objectives

- To describe principal requirements engineering activities and their relationships
- To introduce techniques for requirements elicitation and analysis
- To describe requirements validation and the role of requirements reviews
- To discuss the role of requirements management

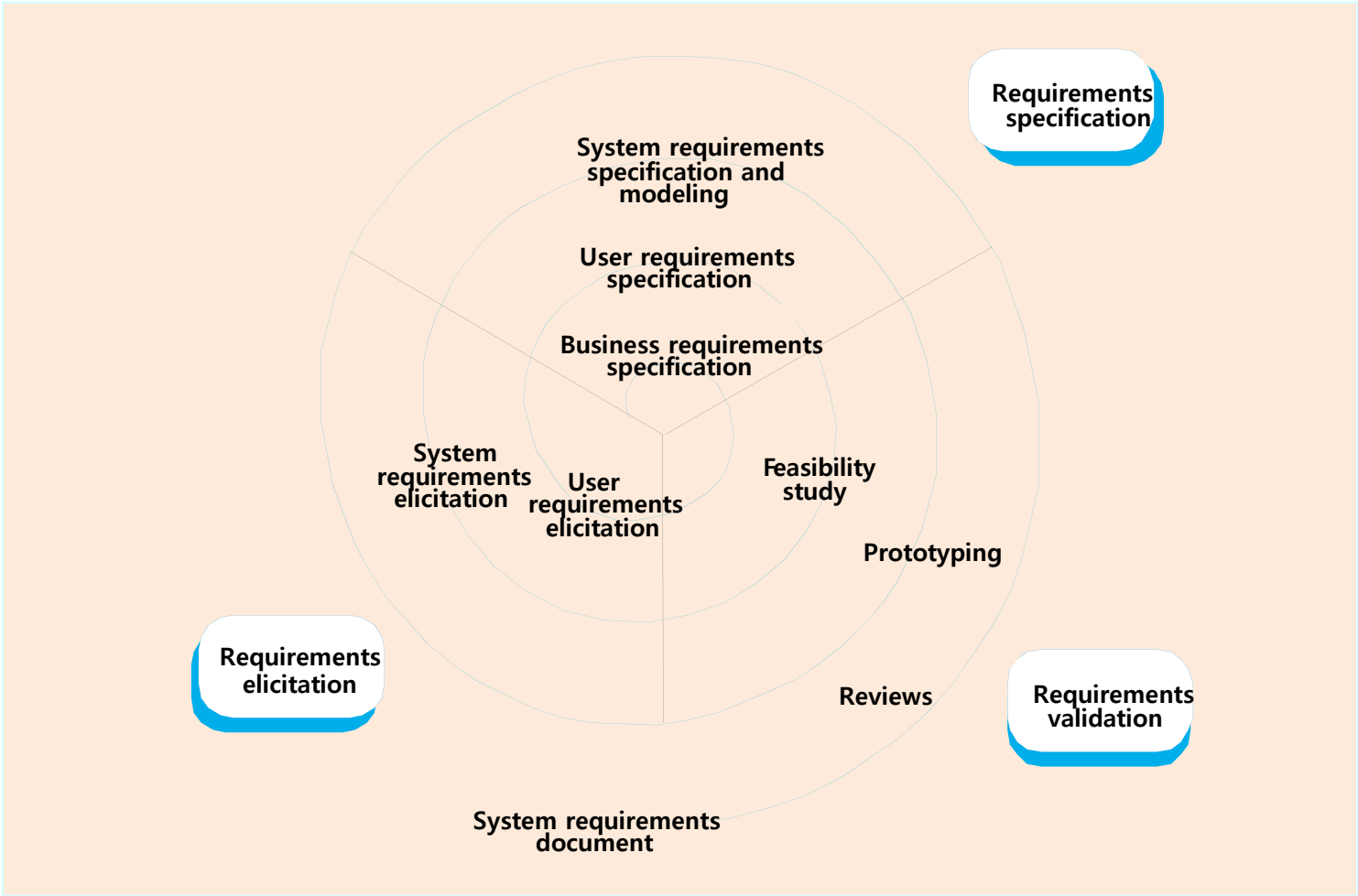
Requirements Engineering Processes

- Requirement engineering processes vary widely depending on
  - Application (target) domain
  - people involved
  - organization developing the requirements

- Generic activities common to all requirements engineering processes



Requirements Engineering Processes



1. Feasibility Study

- Decides whether or not the proposed system is worth to develop

- A short focused study to check
  - If the system contributes to organizational objectives
  - If the system can be engineered using current technology and within budget
  - If the system can be integrated with other systems that are used

- Questions for feasibility:
  - What if the system was not implemented?
  - What are the problems in the current process ?
  - How will the proposed system help to satisfy customer's requirements?
  - What will be the integration problems?
  - Is new technology needed? What skills?
  - What facilities must be supported by the proposed system?

2. Requirements Elicitation and Analysis

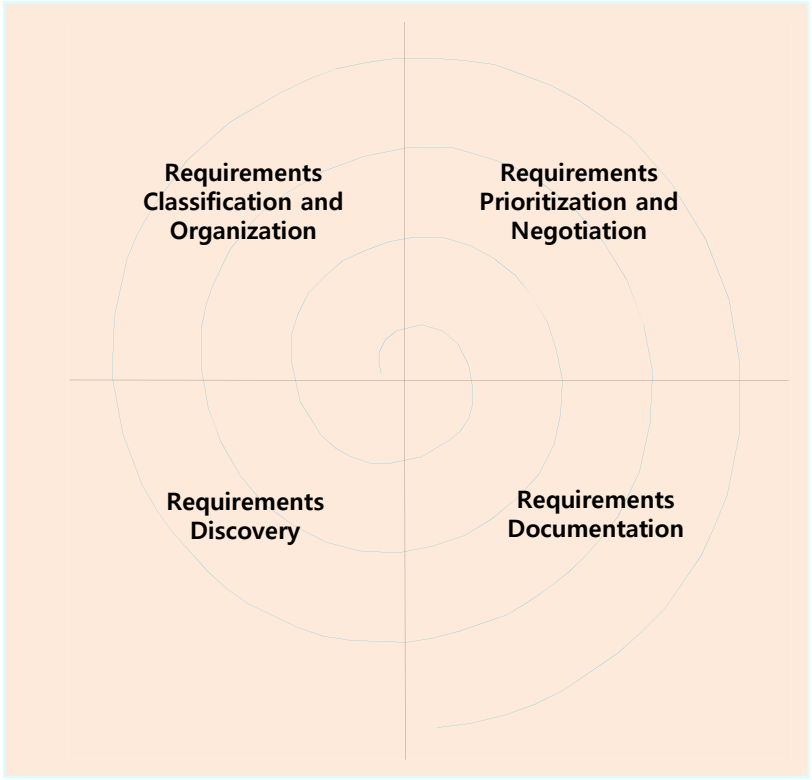
- Called also requirements discovery
- To find out about
  - application domain, services that the system should provide
  - system's operational constraints

- May involve various stakeholders
  - end-users, managers, engineers
  - domain experts, trade unions, etc.

- Problems:
  - 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.

Activities in Requirements Elicitation and Analysis

- Requirements discovery
  - Interact with stakeholders to discover their requirements
  - Discovery domain requirements also
- Requirements classification and organization
  - Group related requirements and organize them into coherent clusters
- Prioritization and negotiation
  - Prioritize the requirements and resolve conflicts among requirements
- Requirements documentation
  - Document requirements
  - Input it into the next round of the spiral



Requirements Discovery

- Requirements discovery is the process of
  - gathering information about the proposed and existing systems
  - distilling the user and system requirements from this information

- Sources of information
  - documentation
  - system stakeholders
  - specifications of similar systems

Interviewing

- The requirements engineering team puts questions to stakeholders about the system to develop.
  - An efficient way of requirements discovery
- Two types of interview
  - Closed interviews : pre-defined set of questions are answered
  - Open interviews : no pre-defined agenda and a range of issues are explored with stakeholders
- A mix of closed and open-ended interviews is normally used.

Scenarios

- Real-life examples of how the system can be used
  - An efficient way of requirements discovery
- Scenarios should include descriptions of
  - Starting situation, normal flow of events and finishing situation
  - Exception cases
  - Information about other concurrent activities
- Example: LIBSYS Scenario

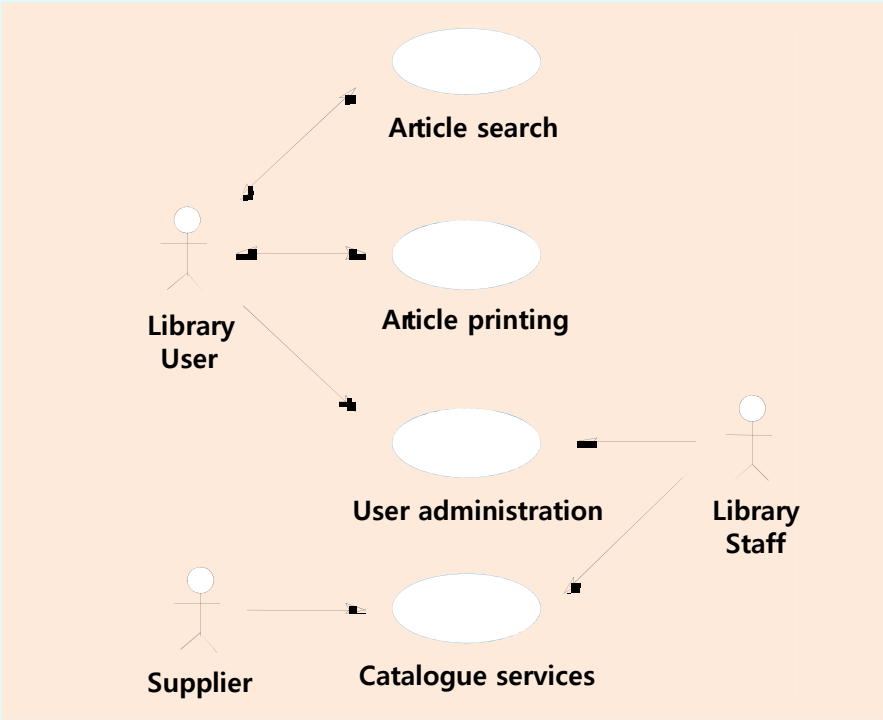
Initial assumption: The user has logged on to the LIBSYS system and has located the journal containing the copy of the article.

Normal: The user selects the article to be copied. He or she is then prompted by the system to either provide subscriber information for the journal or to indicate how they will pay for the article. Alternative payment methods are by credit card or by quoting an organisational account number.
The user is then asked to fill in a copyright form that maintains details of the transaction and they then submit this to the LIBSYS system.
The copyright form is checked and, if OK, the PDF version of the article is downloaded to the LIBSYS working area on the user's computer and the user is informed that it is available. The user is asked to select a printer and a copy of the article is printed. If the article has been flagged as 'print-only' it is deleted from the user's system once the user has confirmed that printing is complete.

Use Cases

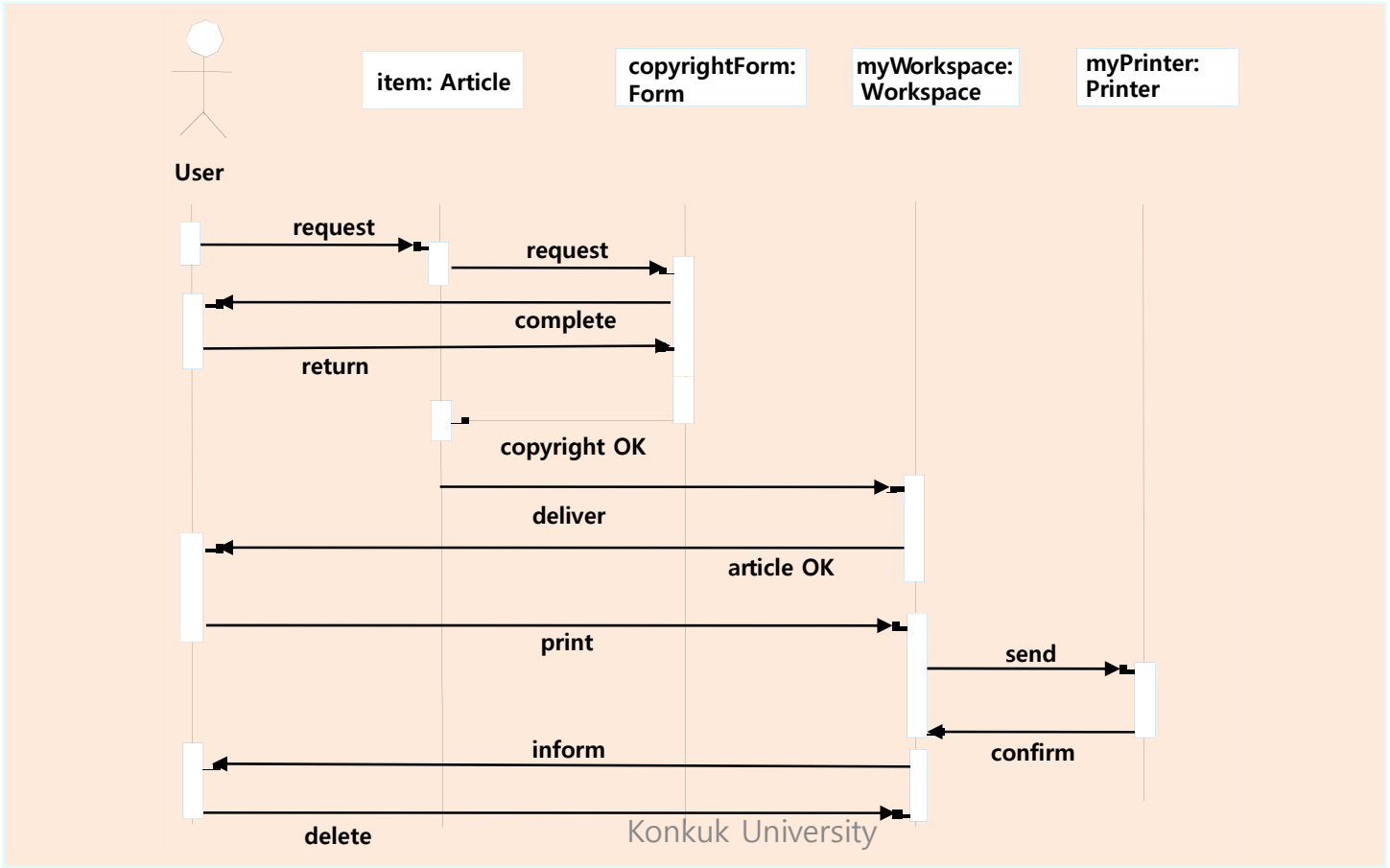
- A scenario based technique in the UML
  - Identify actors in an interaction
  - Describe interactions between actors and the system

LIBSYS use cases



Sequence Diagram

- Add detail to use-cases by showing the sequence of event processing in the system



3. Requirements Validation

- Demonstrate whether the requirements we defined are what the customer really wants.
  - Requirements error costs are high, so validation is very important

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

Requirements Validation Techniques

- Requirements reviews
  - Systematic analysis of requirements
  - Manual analysis
  - Focusing on
    - Verifiability (Testability), Comprehensibility
    - Traceability, Adaptability

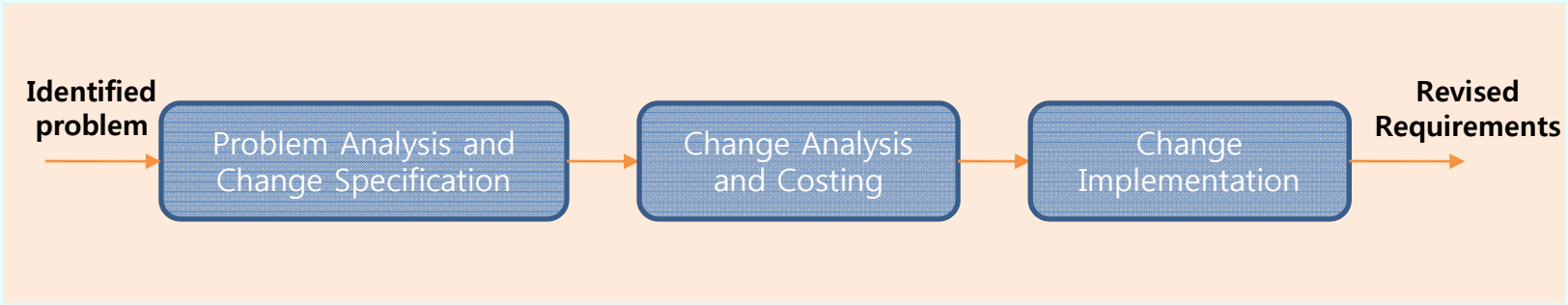
- Prototyping
  - Develop an executable model of the system to check the requirements

- Test-case generation
  - Develop test cases for the requirements to check testability

4. Requirements Management

- The process of managing requirements change during the RE process and system development

- Requirements are inevitably incomplete and inconsistent.
  - New requirements emerge during the process, as business needs change and a better understanding of the system is developed.
  - Different viewpoints have different requirements and these are often contradictory.



Traceability

- Concerned with the relationships between requirements, their sources and the system design
  - Source traceability
    - Links from requirements to stakeholders who proposed these requirements
  - Requirements traceability
    - Links between dependent requirements
  - Design traceability
    - Links from the requirements to the design

Traceability Matrix

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			D		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

CASE Tool Support

- Requirements storage
  - Requirements should be managed in a secure and managed data store.

- Change management
  - A workflow process whose stages should be clearly defined
  - Information flow between stages are partially automated.

- Traceability management
  - Automated retrieval of the links between requirements/sources/designs

Summary

- The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification and requirements management.
- Requirements elicitation and analysis involves domain understanding, requirements collection, classification, structuring, prioritization and validation.
- Systems have multiple stakeholders with different requirements.
- Social and organization factors influence system requirements.
- Requirements validation is concerned with checks for validity, consistency, completeness, realism and verifiability.
- Business changes inevitably lead to changing requirements.
- Requirements management includes planning and change management.

Chapter 8.

System Models

Objectives

- To explain why the context of a system should be modelled as a part of requirements engineering process
- To describe behavioural modelling, data modelling and object modelling
- To show how CASE workbenches support system modelling

System Modelling

- Helps analysts to understand the functionality of the system.
  - System models are used to communicate with customers.

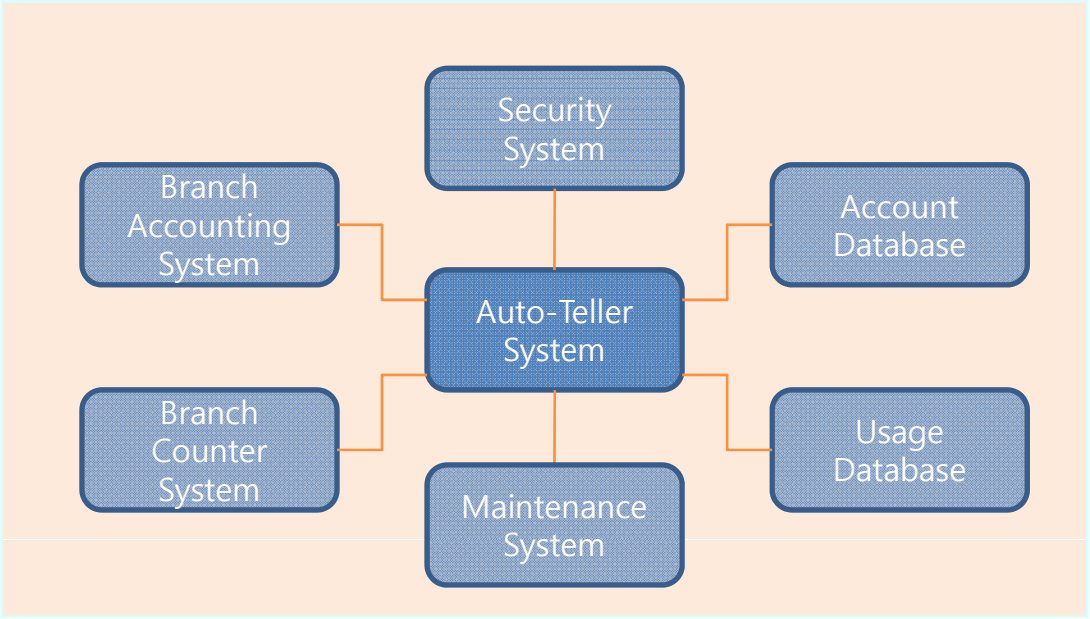
- Different models present the system from different perspectives
  - External perspective : showing the system's context or environment
  - Behavioural perspective : showing the behaviour of the system
  - Structural perspective : showing the system or data architecture

- System model types
  - Data processing model: showing how the data is processed at different stages
  - Composition model: showing how entities are composed of other entities
  - Architectural model: showing principal sub-systems
  - Classification model: showing how entities have common characteristics
  - Stimulus/response model: showing the system's reaction to events
  - Many ones

System Context Model

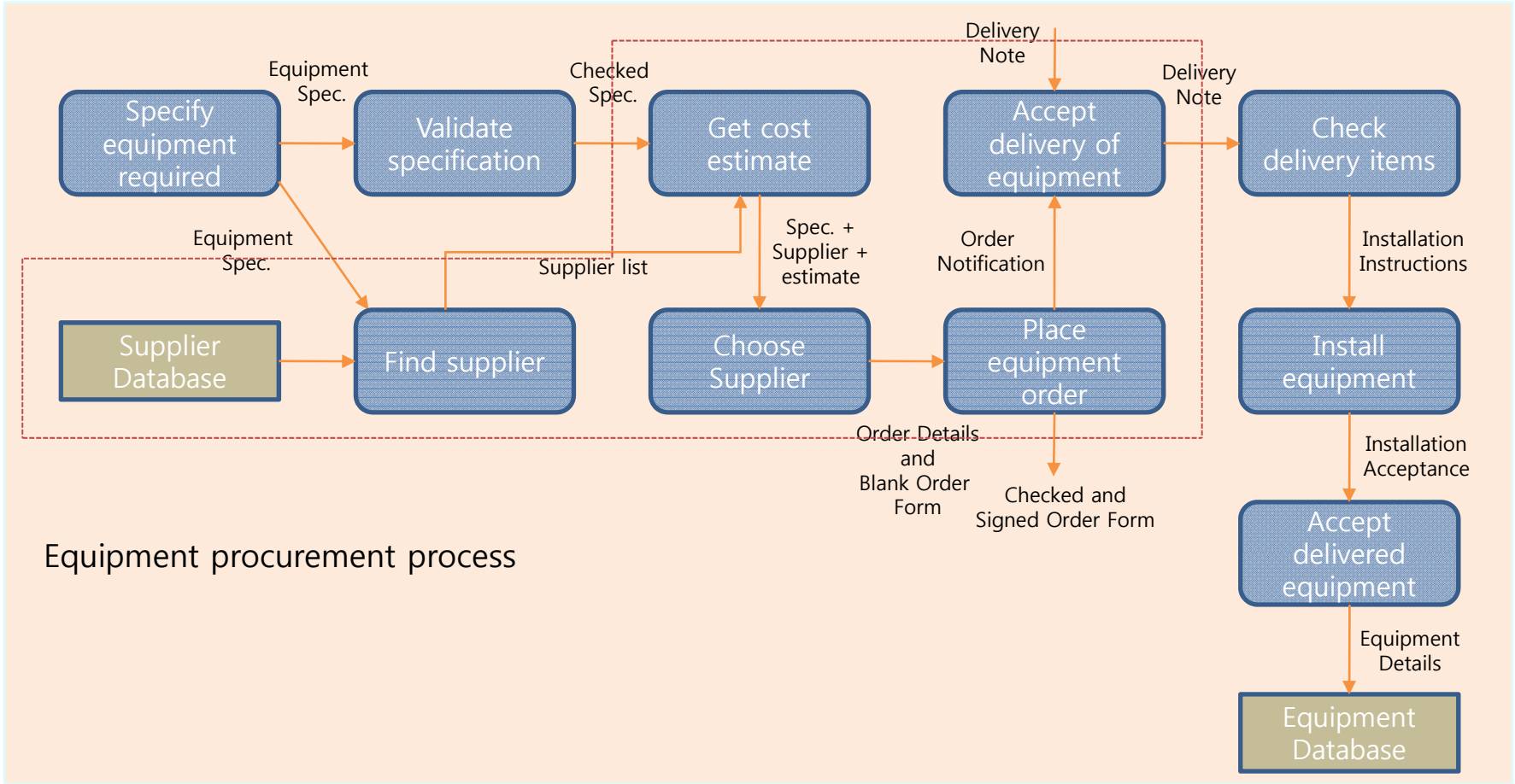
- System Context (models) are used to illustrate the operational context of a system
  - Showing what lies outside the system boundaries
  - Showing the system and its relationship with other systems
  - Social and organizational concerns may affect the decision of system boundaries.

System Context Model for ATM



Process Model

- Process models show the overall process supported by the system.



Equipment procurement process

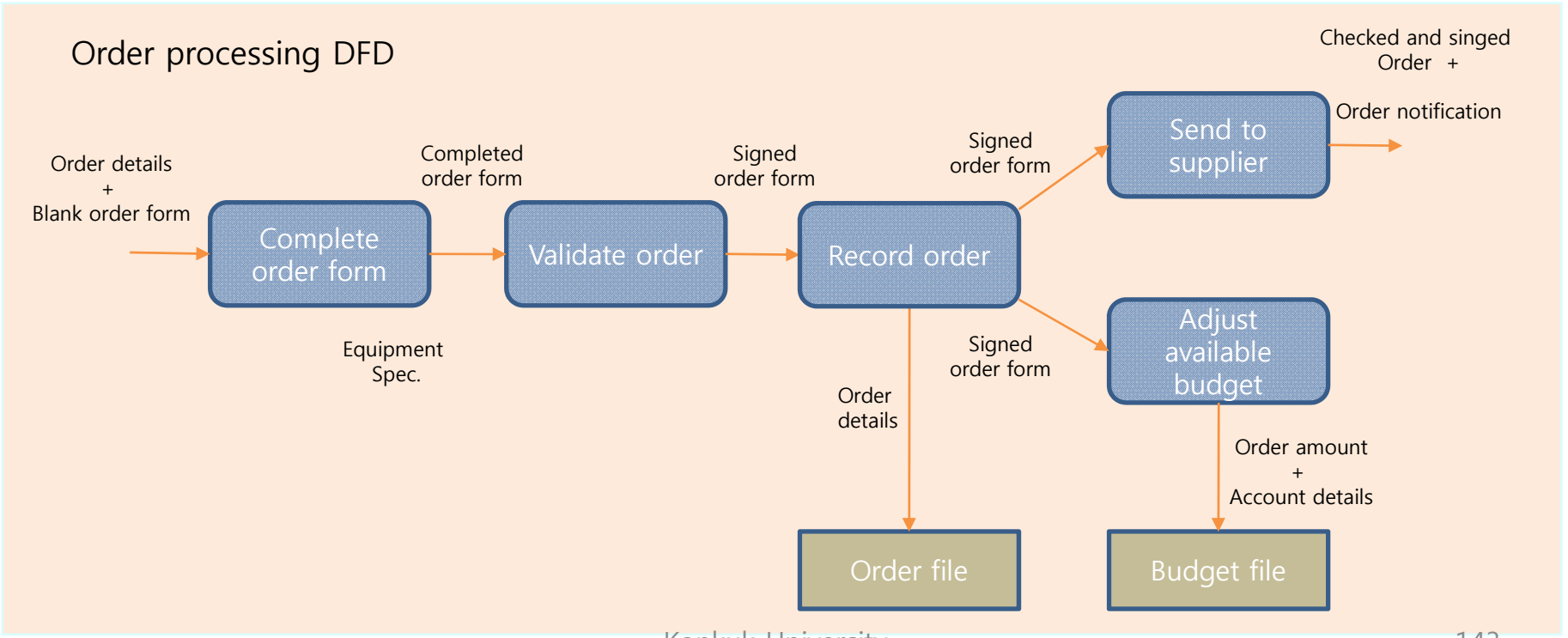
Behavioural Model

- Behavioural models are used to describe the overall behaviour of the system.
  - Data processing models : showing how data is processed as it moves through the system
  - State machine models : showing how the system responses to events

  - Two models show different perspectives.
  - Both of them are required to describe the system's behaviour.

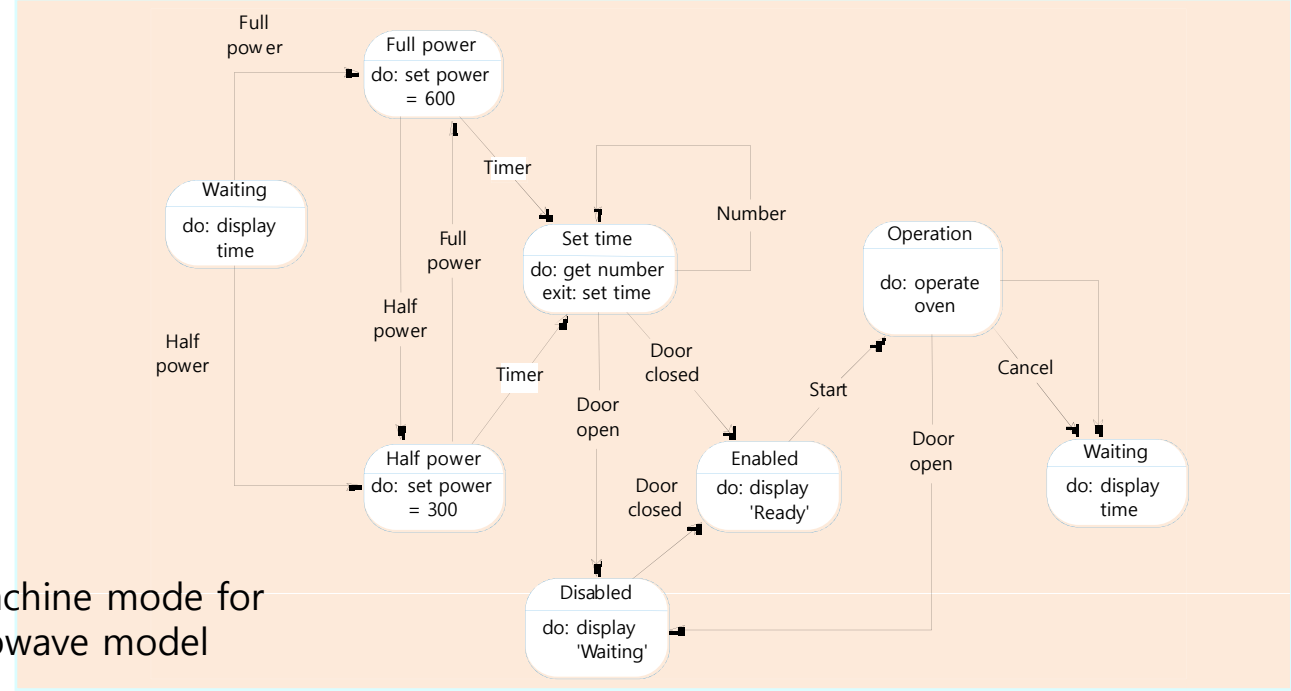
Data Processing Model

- Data flow diagrams(DFDs) are used to model the system's data processing.
  - Show the processing steps as data flows through a system
  - Use simple and intuitive notation that customers can understand
  - Show end-to-end processing of data



State Machine Model

- State machine models model the behaviour of the system in response to external and internal events.
  - Show the system's responses to stimuli
  - Often used for modelling real-time systems
  - Show system states as nodes and events as arcs between these nodes

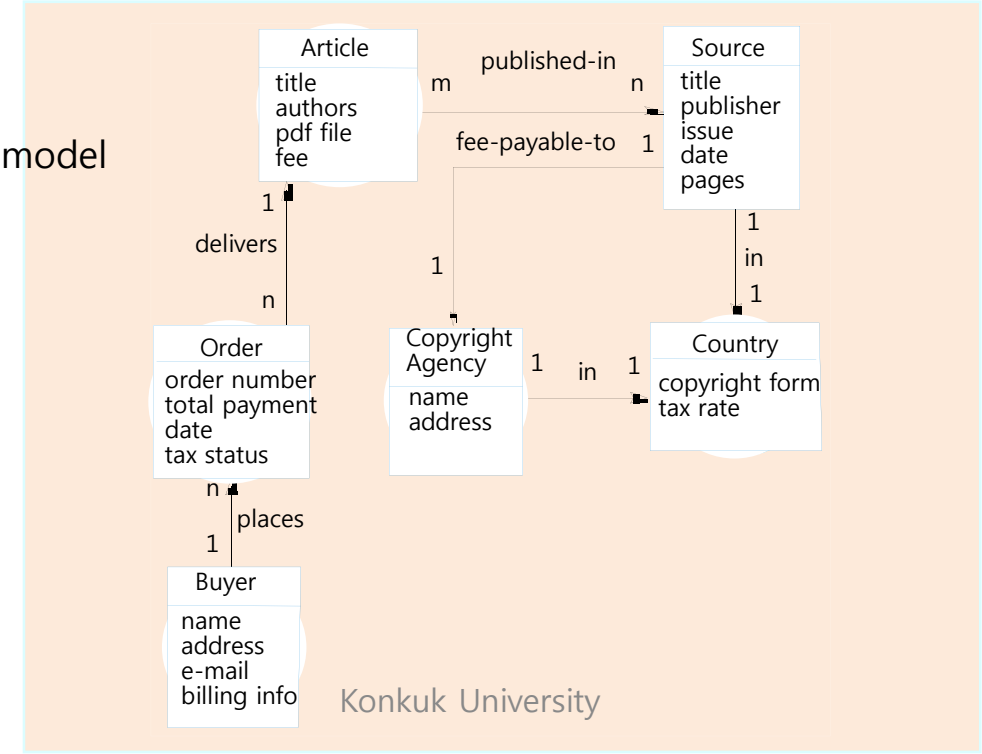


State machine mode for Microwave model

Semantic Data Model

- Semantic data models are used to describe the logical structure of data processed by the system.
  - Entity-relation-attribute model : setting out the entities in the system, relationships between these entities, and the entity attributes
  - Widely used in rational database design

Library semantic model



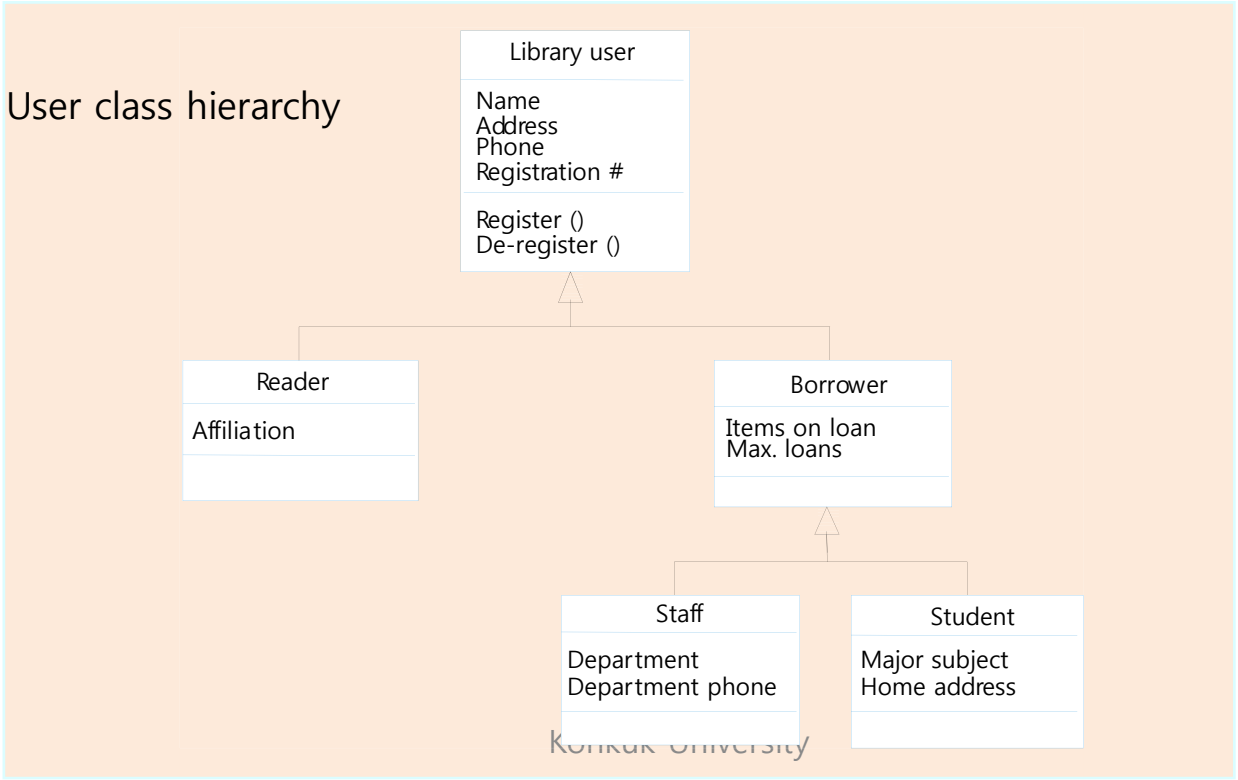
Object Model

- Object models describe the system in terms of object classes and their associations.
  - An object class is an abstraction over a set of objects with common attributes and the services (operations).
  - Object classes are reusable across systems.

- Various object models
  - Inheritance model
  - Aggregation model
  - Interaction model

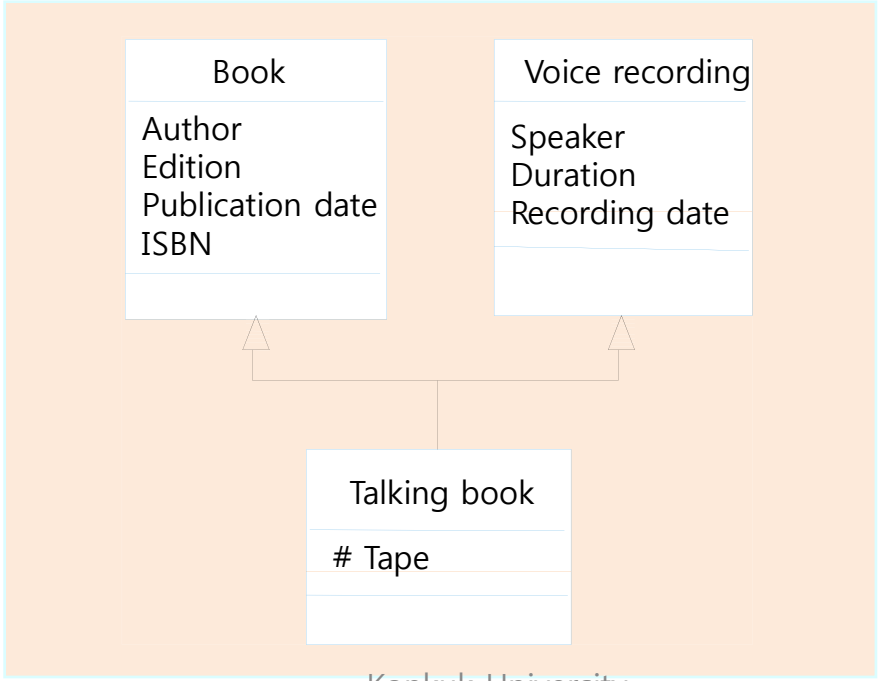
Inheritance Model

- Inheritance models organize domain object classes into a hierarchy.
  - Classes at the top of the hierarchy reflect common features of all classes.
  - Object classes inherit their attributes and services from one or more super-classes.



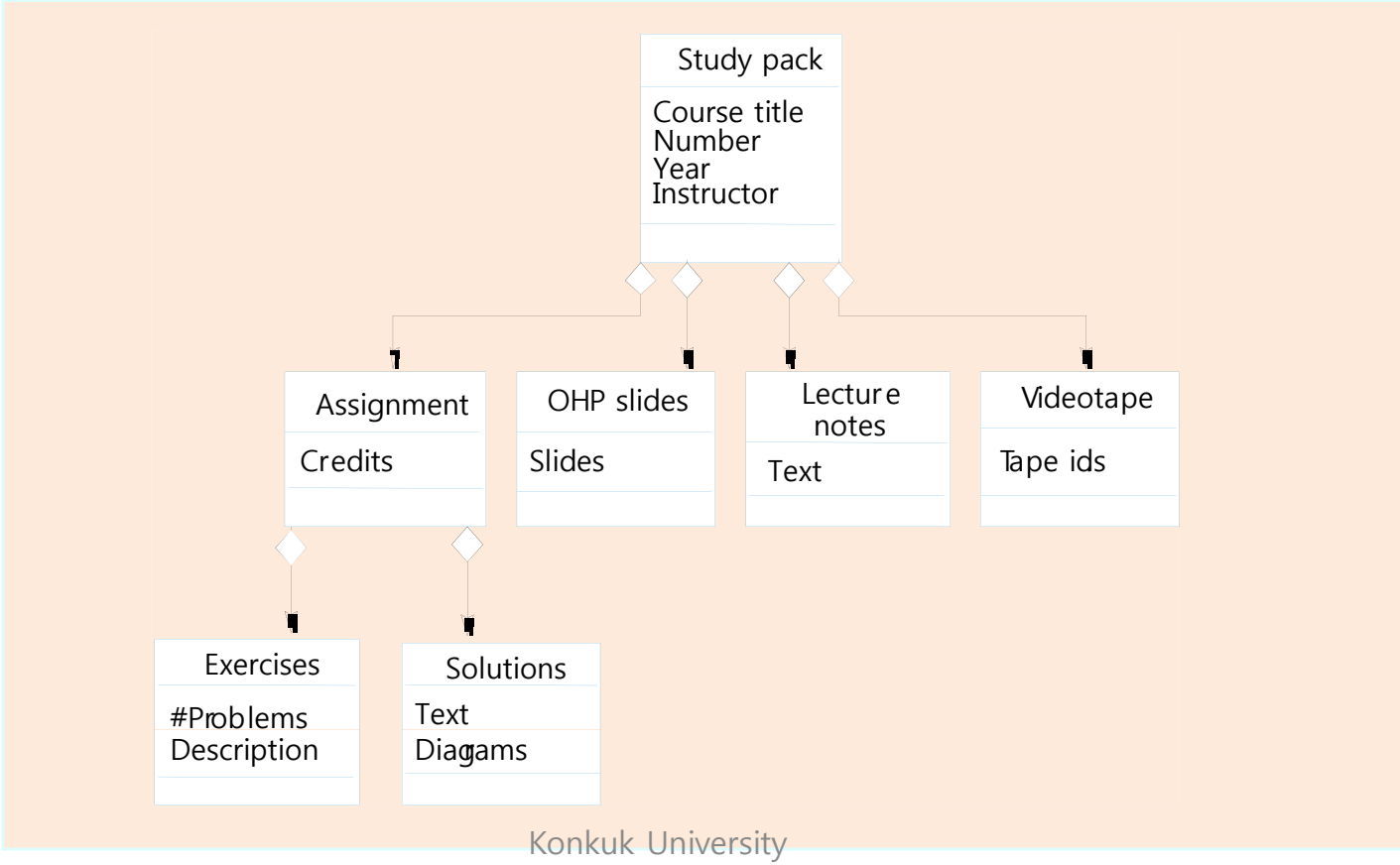
Multiple Inheritance

- Multiple inheritance allows object classes to inherit from several super-classes.
  - May lead to semantic conflicts where attributes/services with the same name in different super-classes have different semantics
  - Make class hierarchy reorganisation more complex



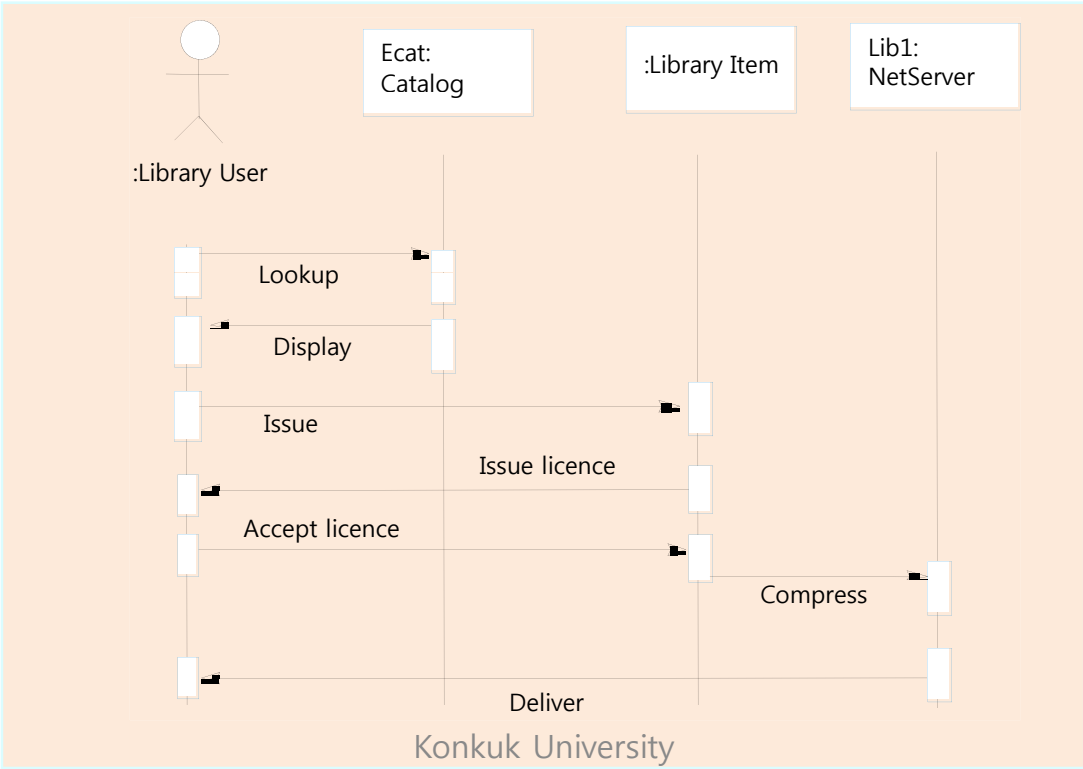
Object Aggregation Model

- Aggregation models show how classes are composed of other classes.
  - Similar to the part-of relationship in semantic data models



Object Behaviour Model

- Object behavioural models show the interactions between objects
  - To produce some particular system behaviour specified as in use-cases
  - Called interaction model
  - Sequence diagrams (or collaboration diagrams) in the UML



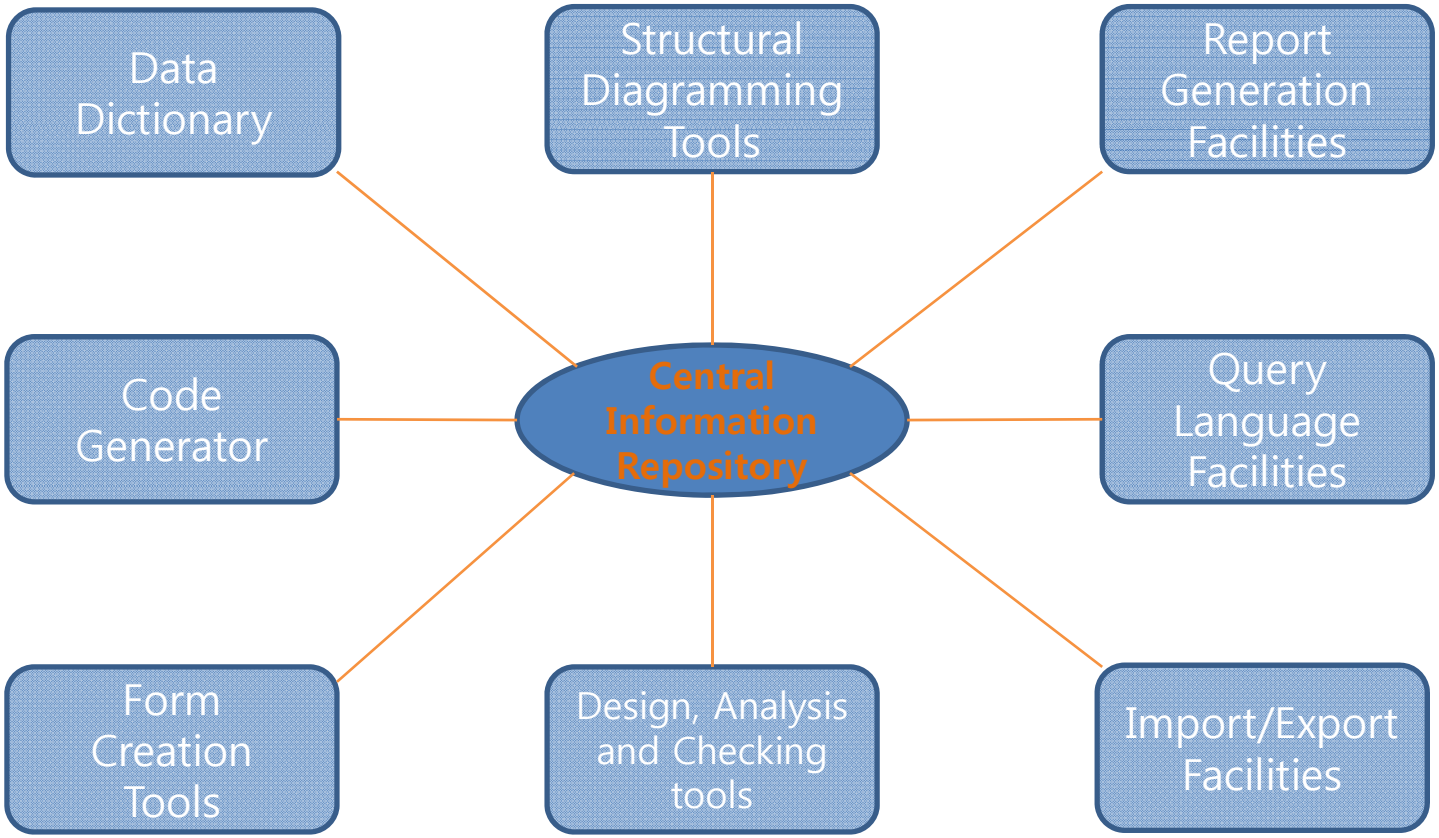
Structured Method

- Structured methods incorporate system modelling as an inherent part of the method.

- Structured methods define
  - a set of models
  - a process for deriving these models
  - rules and guidelines that should apply to the models
  - CASE tools to support system modelling

- CASE Workbench:
  - A coherent set of tools that is designed to support related software process activities such as analysis, design or testing.
  - Analysis and design workbenches support system modelling during both requirements engineering and system design.
  - May support a specific design method
  - May support to create several different types of system model

Analysis and Design Workbench: An example



Summary

- A model is an abstract system view. Complementary types of model provide different system information.
- Context models show the position of a system in its environment with other systems and processes.
- Data flow models are used to model the data processing in a system.
- State machine models model the system's behaviour in response to internal or external events.
- Semantic data models describe the logical structure of data which is imported to or exported by the systems.
- Object models describe logical system entities, their classification and aggregation.
- Sequence models show the interactions between actors and the system objects that they use.
- Structured methods provide a framework for developing system models.

Part III. Design

Chapter 13.

Application Architectures

Objectives

- To explain two fundamental models of business systems - batch processing system and transaction processing system
- To describe abstract architecture of resource management systems
- To explain how generic editors are event processing systems
- To describe the structure of language processing systems

Generic Application Architectures

- As businesses have much in common,
  - their application systems also tend to have a common architecture that reflects the application requirements.

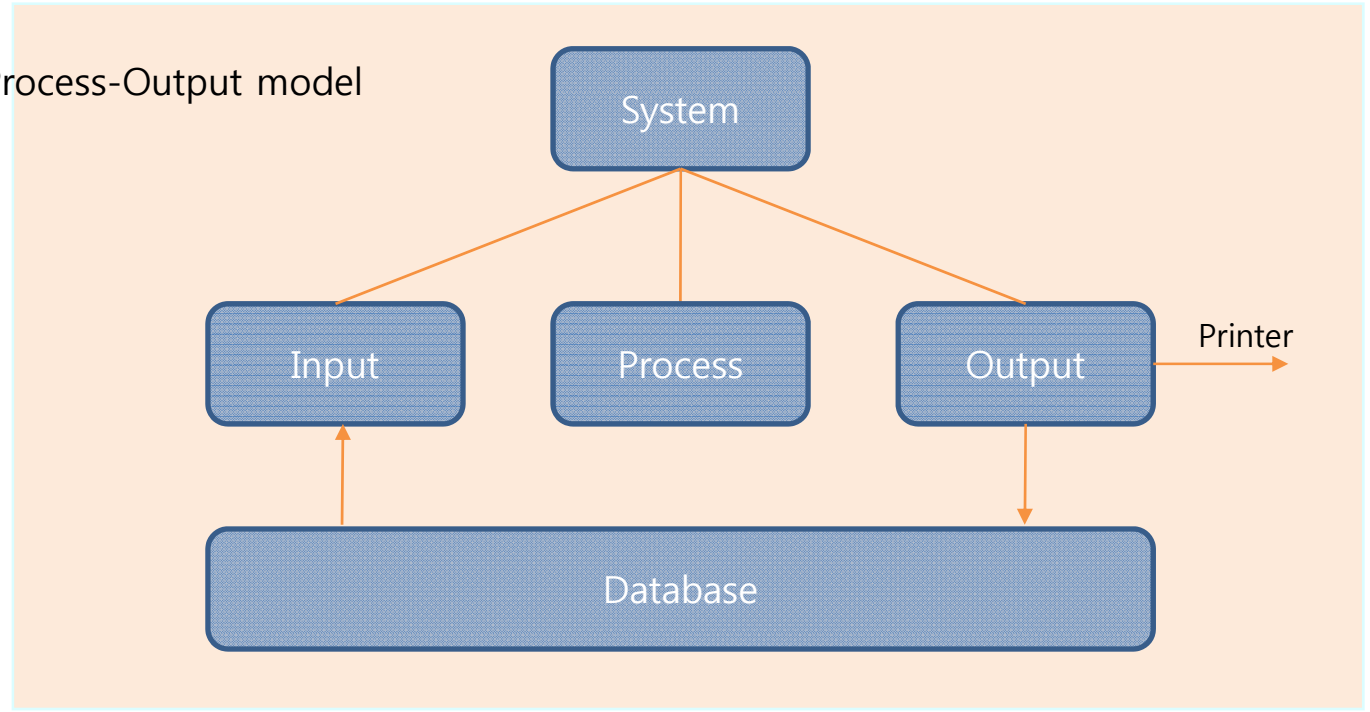
Application Types

- Application types
  1. Data processing application
     - Data driven applications
     - Process data in batches without user intervention during the processing.
     - Ex) Billing system, Payroll system
  2. Transaction processing application
     - Data-centered applications
     - Process user requests and update information in a system database.
     - Ex) E-commerce system, Reservation system
  3. Event processing system
     - System actions depend on interpreting events from the system's environment.
     - Ex) Word processor, Real-time system
  4. Language processing system
     - Users' intentions are specified in a formal language.
     - Processed and interpreted by the system.
     - Ex) Compiler, Command interpreter

1. Data Processing System

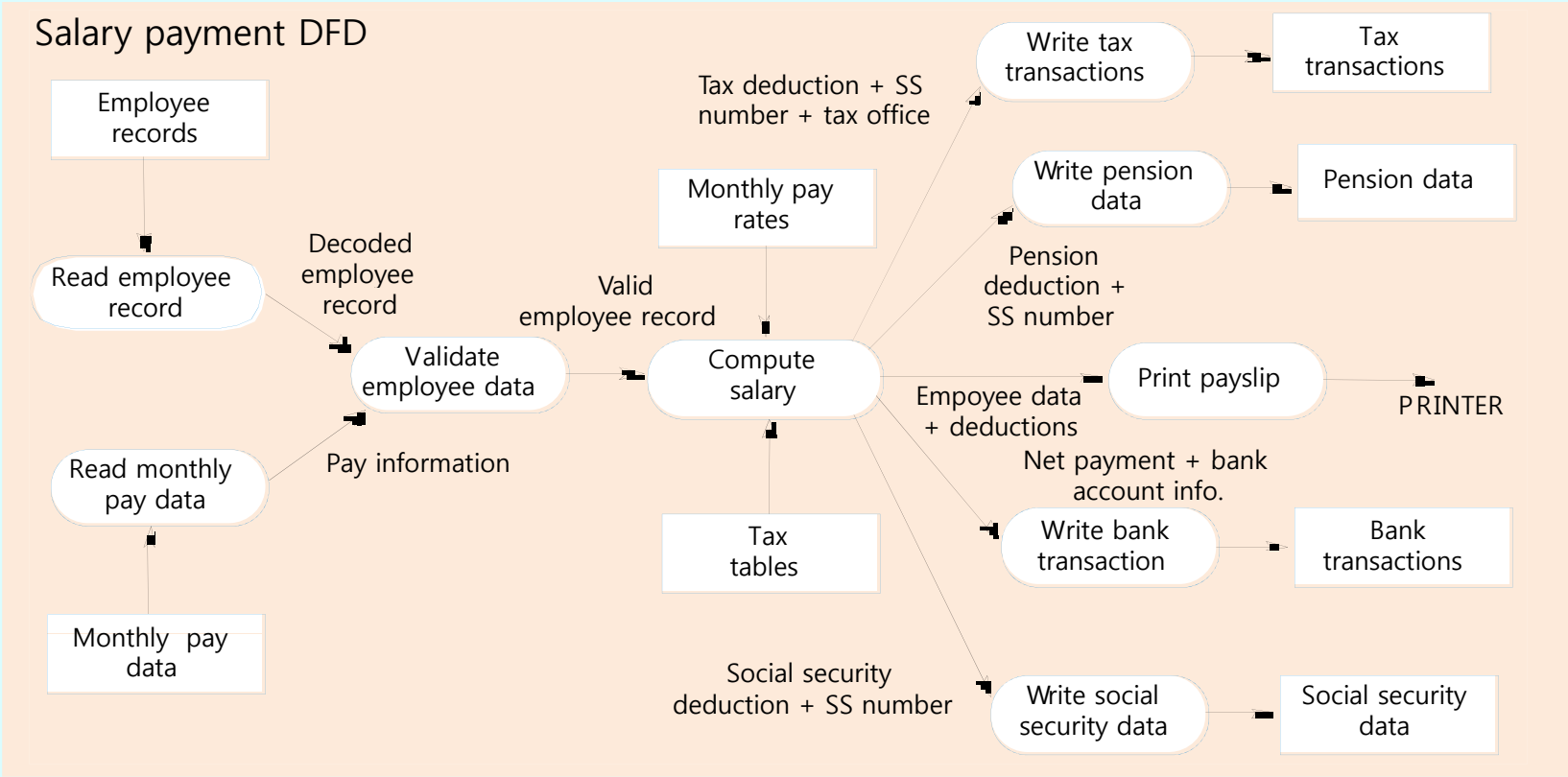
- Data-centered system, where databases used are usually orders of magnitude larger than the software itself.
  - Data is input and output in batches.
  - Have an input-process-output structure

Input-Process-Output model



Data-Flow Diagram

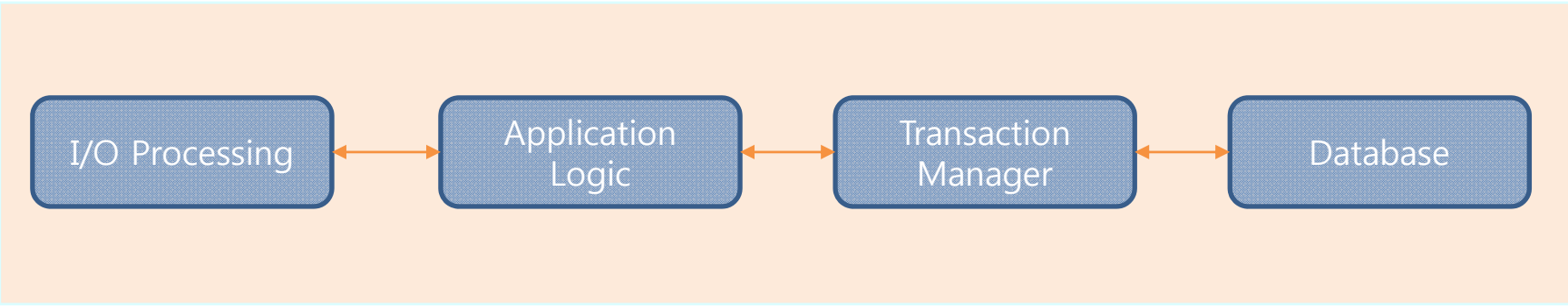
- DFD shows how data is processed as it moves through a system.
  - Round-edged rectangles : transformations
  - Arrows : data-flows
  - Rectangles : data (input/output)



2. Transaction Processing System

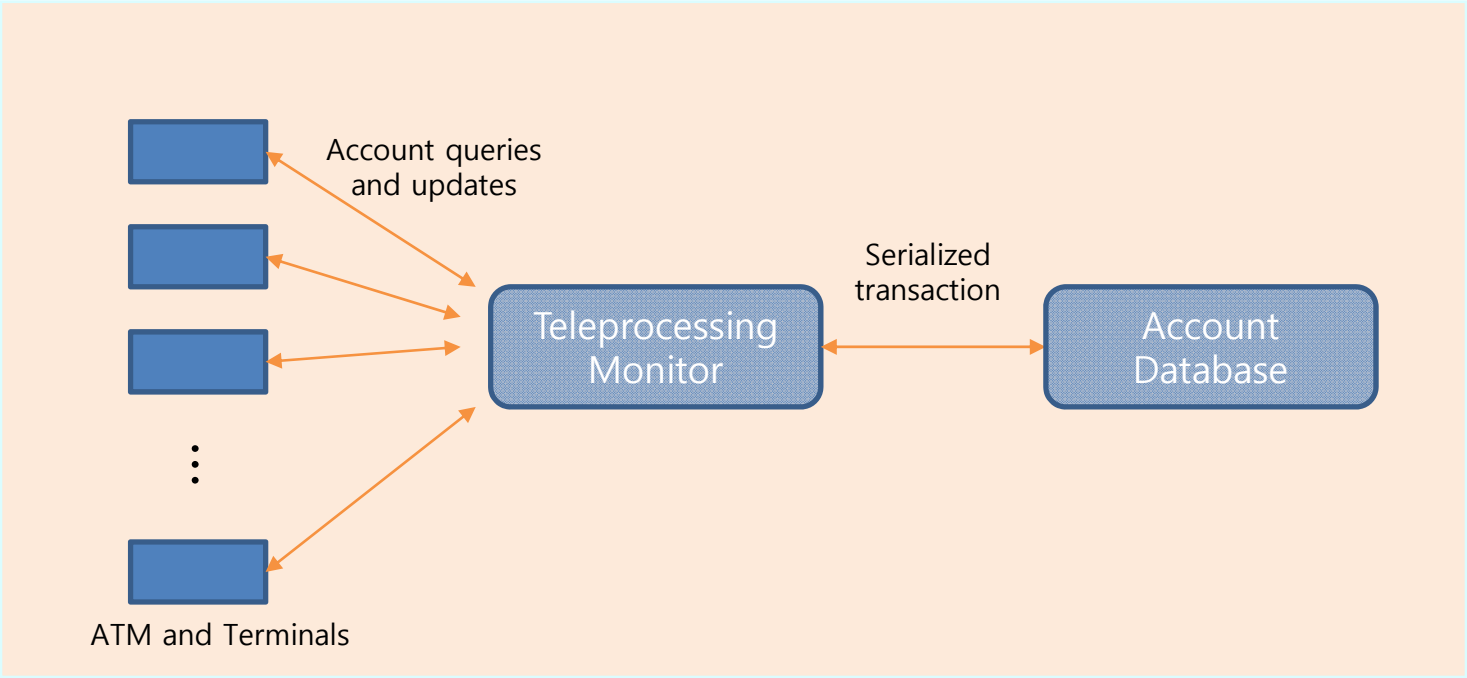
- Transaction processing systems process
  - User requests for information from a database or
  - User requests to update the database.

  - Users make asynchronous requests for service which are then processed by a transaction manager.
  - Many examples
    - Transaction processing middleware
    - Information system architecture
    - Resource allocation system
    - E-commerce system architecture



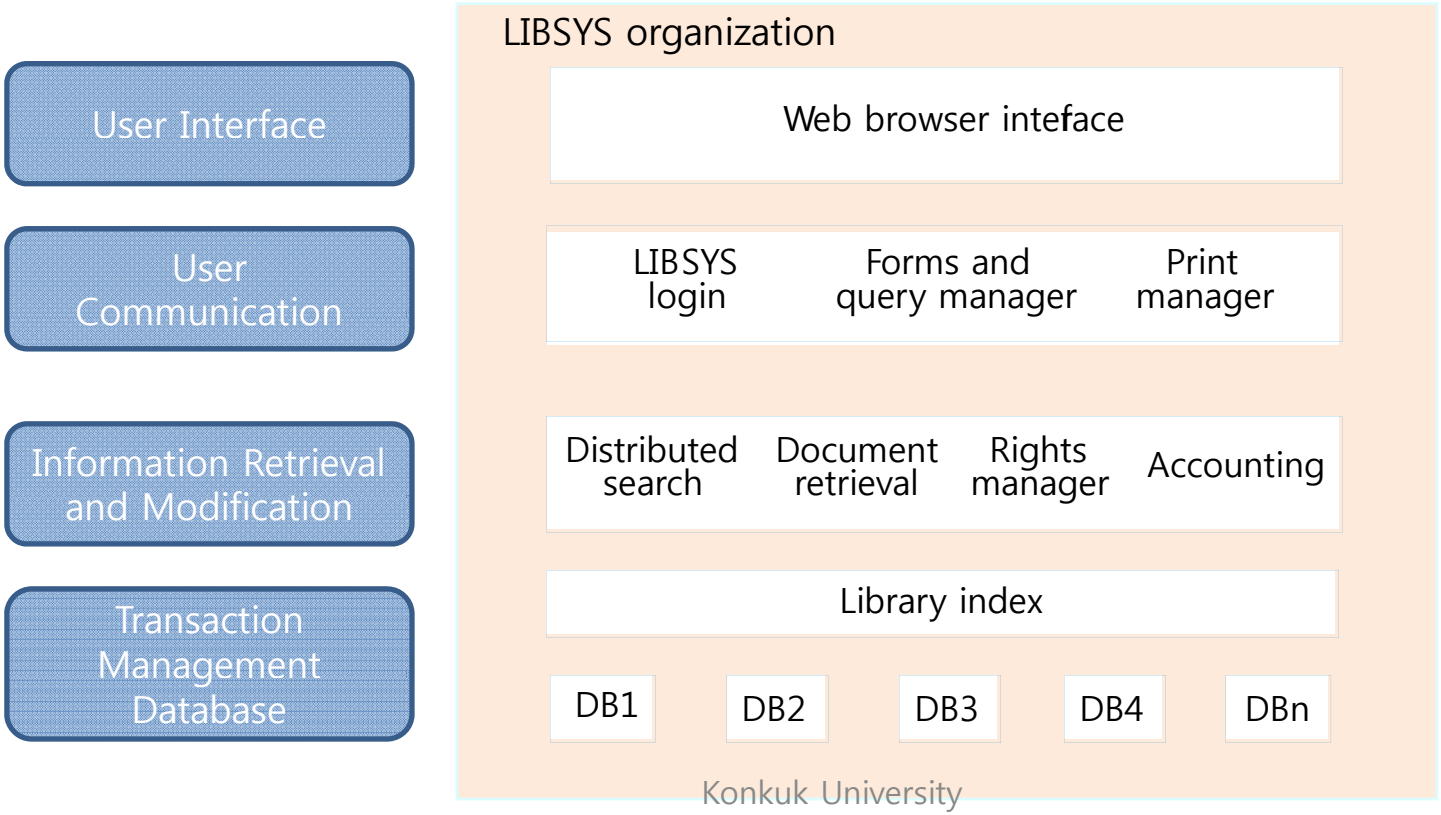
Transaction Processing Middleware

- Transaction management middleware or teleprocessing monitors
  - Handle communications with different terminal types, serializes data and sends it for processing
  - Query processing takes place in the system database and results are sent back through the transaction manager to the user's terminal.



Information System Architecture

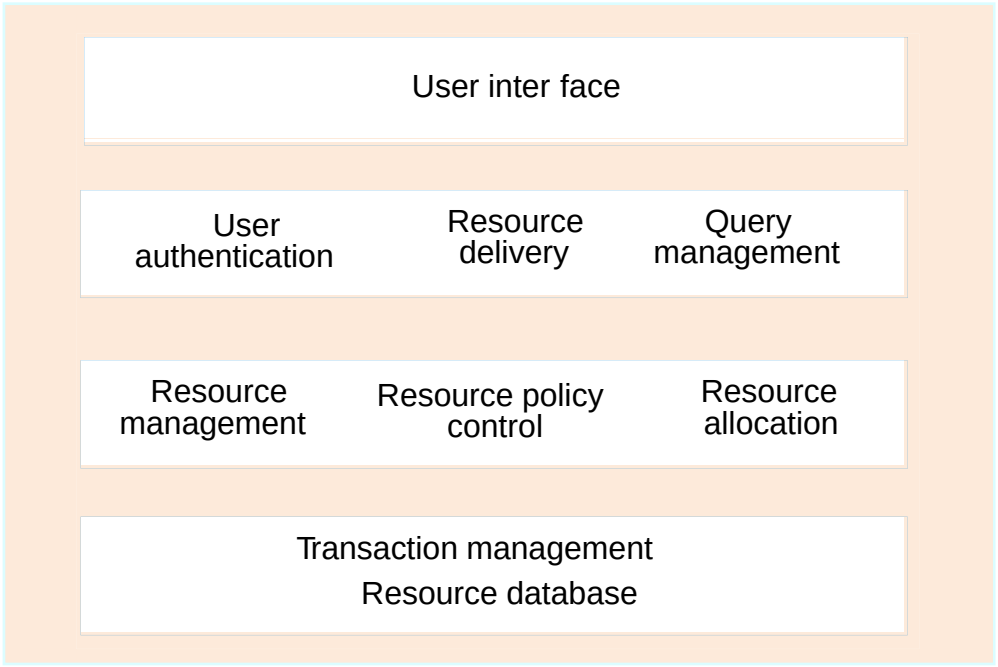
- Information systems can be organized as a layered architecture.
- LIBSYS example :



Resource Allocation System

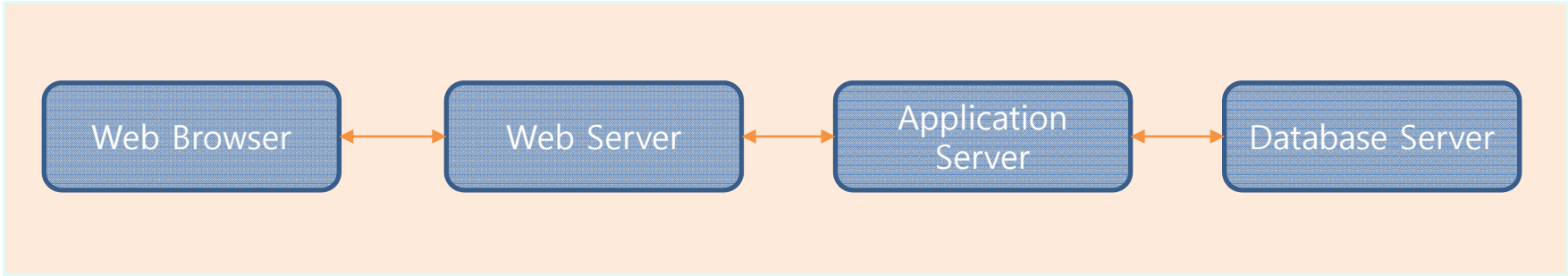
- Resource allocation systems manage fixed amount of resource and allocate them to users.
  - Timetabling system : the resource being allocated is a time period
  - Library system : the resource being managed is books for loan
  - Air traffic control system : the resource being managed is the airspace

- Layer resource allocation architecture



E-commerce System Architecture

- E-commerce systems are internet-based resource management systems
  - Accept electronic orders for goods or services
  - Organized using a multi-tier architecture with application layers associated with each tier

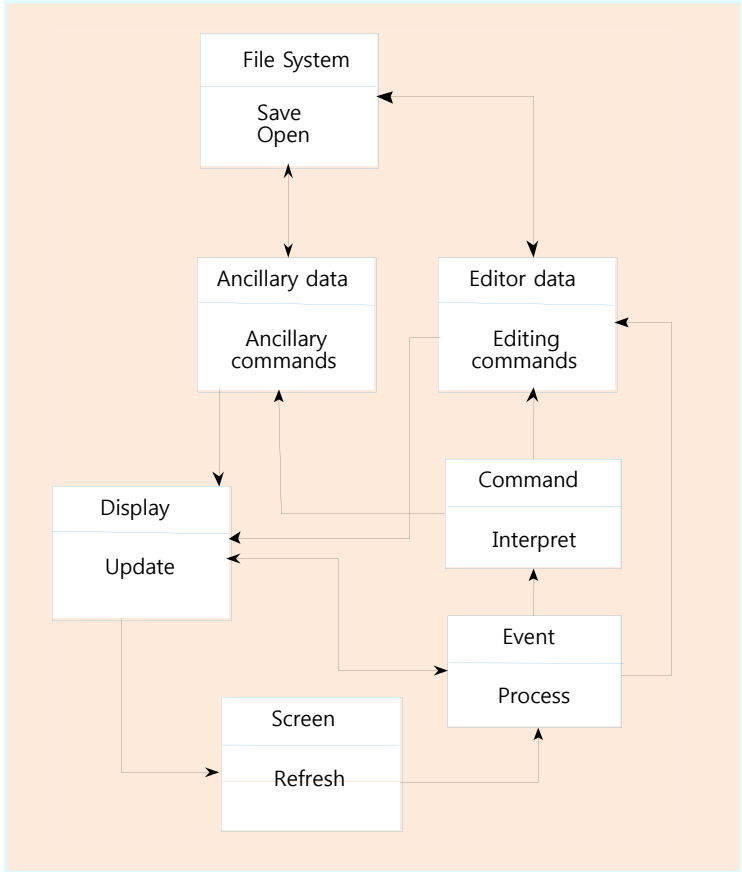


3. Event Processing Systems

- Event processing systems respond to events in the system's environment.
  - Event timing is unpredictable, so the architecture has to be organized to handle this.
  - Many common systems:
    - Word processors
    - Games
    - Real-time systems
    - Etc.

Editing System

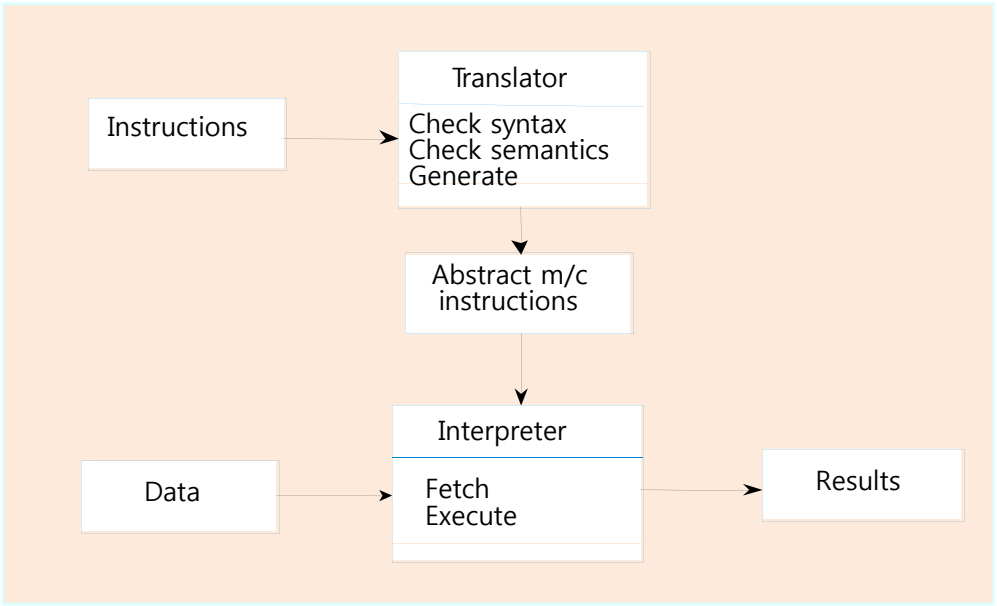
- Editing systems are the most common types of event processing system.
  - Single user system
  - Must provide rapid feedback to user actions
  - Organized around long transactions
  - May include recovery facilities



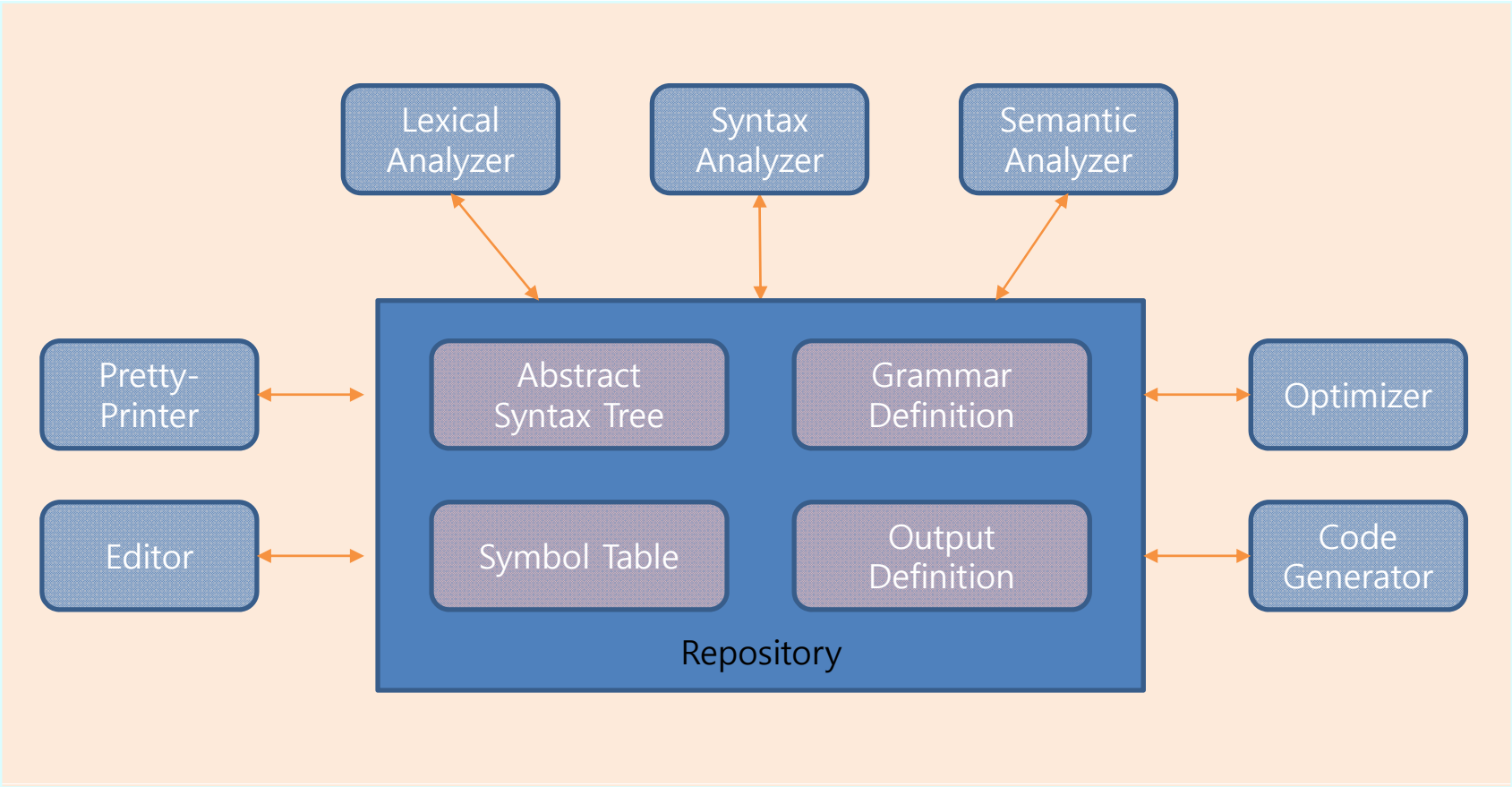
4. Language Processing System

- Language processing systems accept a natural or artificial language as input and generate some other representation of that language.
  - May include an interpreter

- Components of language processing systems
  - Lexical analyzer
  - Symbol table
  - Syntax analyzer
  - Syntax tree
  - Semantic analyzer
  - Code generator



Repository Model of Compiler



Summary

- Generic models of application architectures help us understand and compare applications.
- Important classes of application are data processing systems, transaction processing systems, event processing systems and language processing system.
- Data processing systems operate in batch mode and have an input-process-output structure.
- Transaction processing systems allow information in a database to be remotely accessed and modified by multiple users.
- Event processing systems include editors and real-time systems.
- In an editor, user interface events are detected and an in-store data structure is modified.
- Language processing systems translate texts from one language to another and may interpret the specified instructions.

Chapter 14.

Object-Oriented Design

Objectives

- To explain how a software design may be represented as a set of interacting objects that manage their own states and operations
- To describe the activities in object-oriented design process
- To introduce various models that can be used to describe an object-oriented design
- To show how the UML may be used to represent these models

Object-Oriented Development

- Object-oriented analysis, design and programming are related but distinct.
  - OOA : concerned with developing an object model of the application domain
  - OOD : concerned with developing an object-oriented system model to implement requirements
  - OOP : concerned with realizing an OOD using an OO programming language such as Java or C++

- Characteristics of OOD
  - Objects are abstractions of real-world or system entities.
  - Objects encapsulate state and representation information.
  - System functionality is expressed in terms of object services.
  - Shared data areas are eliminated.
  - Objects communicate by message passing.
  - Objects may be distributed and may execute sequentially or in parallel.

Advantages of OOD

- Easier maintenance
  - Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- Easy to implement for some systems
  - There may be an obvious mapping from real world entities to system objects.

Objects and Object Classes

- Objects are entities in software system
  - Represent instances of real-world and system entities
- Object classes are templates for objects
  - Used to create objects
  - May inherit attributes and services from other object classes

An object is an entity that has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.

Objects are created according to some object class definition. An object class definition serves as a template for objects. It includes declarations of all the attributes and services which should be associated with an object of that class.

Unified Modelling Language

- Several different notations for describing object-oriented designs were proposed in the 1980s and 1990s.

- Unified Modelling Language(UML) is an integration of these.
  - Describes notations for a number of different models that may be produced during OO analysis and design
  - A de facto standard for OO modelling

Class Example: Employee Object

Object Communication

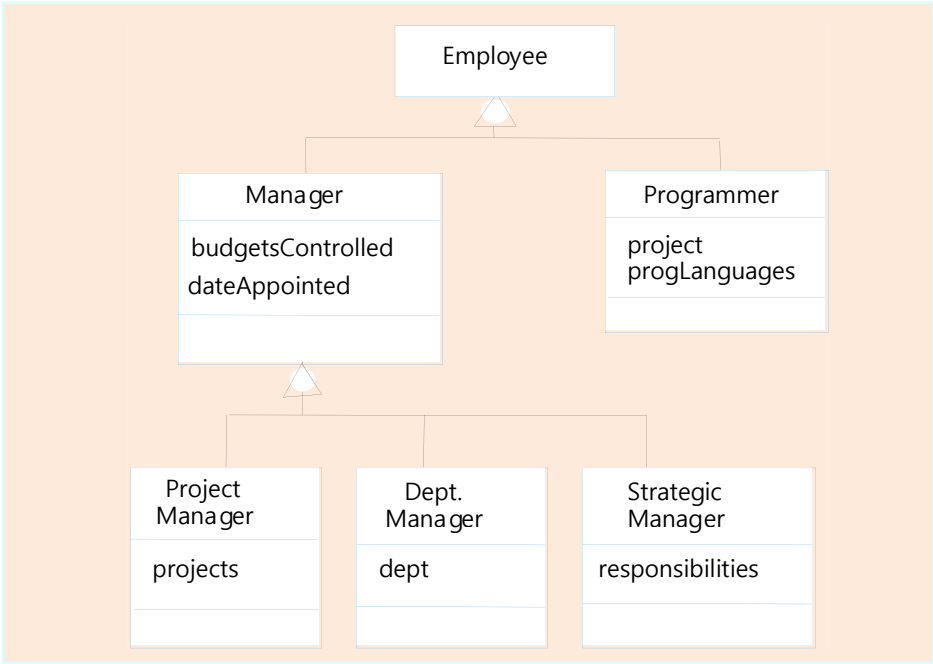
- Conceptually, objects communicate by message passing.
- Messages
  - Name of service requested by calling object
  - Copies of information required to execute the service

- In practice, messages are often implemented by procedure calls.
  - Name = procedure name
  - Information = parameter list

```
// Call a method associated with a buffer object that returns the next value in the buffer
        v = circularBuffer.Get ( ) ;

// Call the method associated with a thermostat object that sets the temperature
// to be maintained
        thermostat.setTemp (20) ;
```

Generalization and Inheritance

- Classes may be arranged in a class hierarchy, where one class (a super-class) is a generalization of one or more other classes (sub-classes).
  - A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own.
  - Generalization in the UML is implemented as an inheritance in OO programming languages.



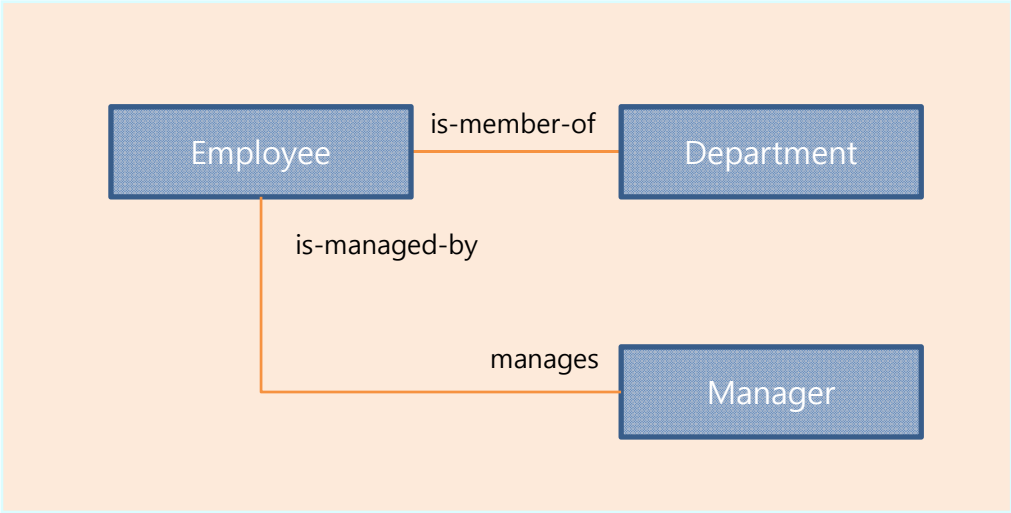
Features of Inheritance

- Advantages:
  - Abstraction mechanism : may be used to classify entities.
  - Reuse mechanism at both the design and the programming level.
  - Inheritance graph is a source of organizational knowledge about domains and systems.
- Problems:
  - Object classes are not self-contained. They cannot be understood without reference to their super-classes.
  - Designers have a tendency to reuse the inheritance graph created during analysis. It may lead to significant inefficiency.
  - Inheritance graphs of analysis, design and implementation have different functions and should be separately maintained.

UML Association

- Objects and object classes participate in relationships with other objects and object classes.

- In the UML, a generalized relationship is indicated by an association.
  - May be annotated with information that describes the association
    - May indicate that an attribute of an object is an associated object
    - May indicate that a method relies on an associated object



Concurrent Object

- The nature of objects :
  - Self-contained entities are suitable for concurrent implementation.
  - Message-passing model of object communication can be implemented directly if objects are running on separate processors in a distributed system.

- Servers
  - The object is implemented as a parallel process (server) with entry points corresponding to object operations.
  - If no calls are made to it, the object suspends itself and waits for further requests for service.

- Active objects
  - Objects are implemented as parallel processes and the internal object state may be changed by the object itself and not simply by external calls.
  - Thread in Java is a simple construct for implementing concurrent objects.

Java Thread

- Thread in Java is a simple construct for implementing concurrent objects.
  - Threads must include a method called run( ) and this is started up by the Java run-time system.
  - Active objects typically include an infinite loop so that they are always carrying out the computation.

Object-Oriented Design Process

- Structured design processes involve developing a number of different system models.
  - Require a lot of effort for development and maintenance of these models
  - For small systems, it may not be cost-effective.
  - However, for large systems developed by different groups, design models are an essential communication mechanism.

- Common key activities for OOD processes
  1. Define the context and modes of use of the system
  2. Design the system architecture (Architectural design)
  3. Identify the principal system objects (Object identification)
  4. Develop design models
  5. Specify object interfaces (Object interface specification)

Example: Weather Mapping System Description

A weather mapping system is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations and other data sources such as weather observers, balloons and satellites. Weather stations transmit their data to the area computer in response to a request from that machine.

The area computer system validates the collected data and integrates it with the data from different sources. The integrated data is archived and, using data from this archive and a digitised map database a set of local weather maps is created. Maps may be printed for distribution on a special-purpose map printer or may be displayed in a number of different formats.

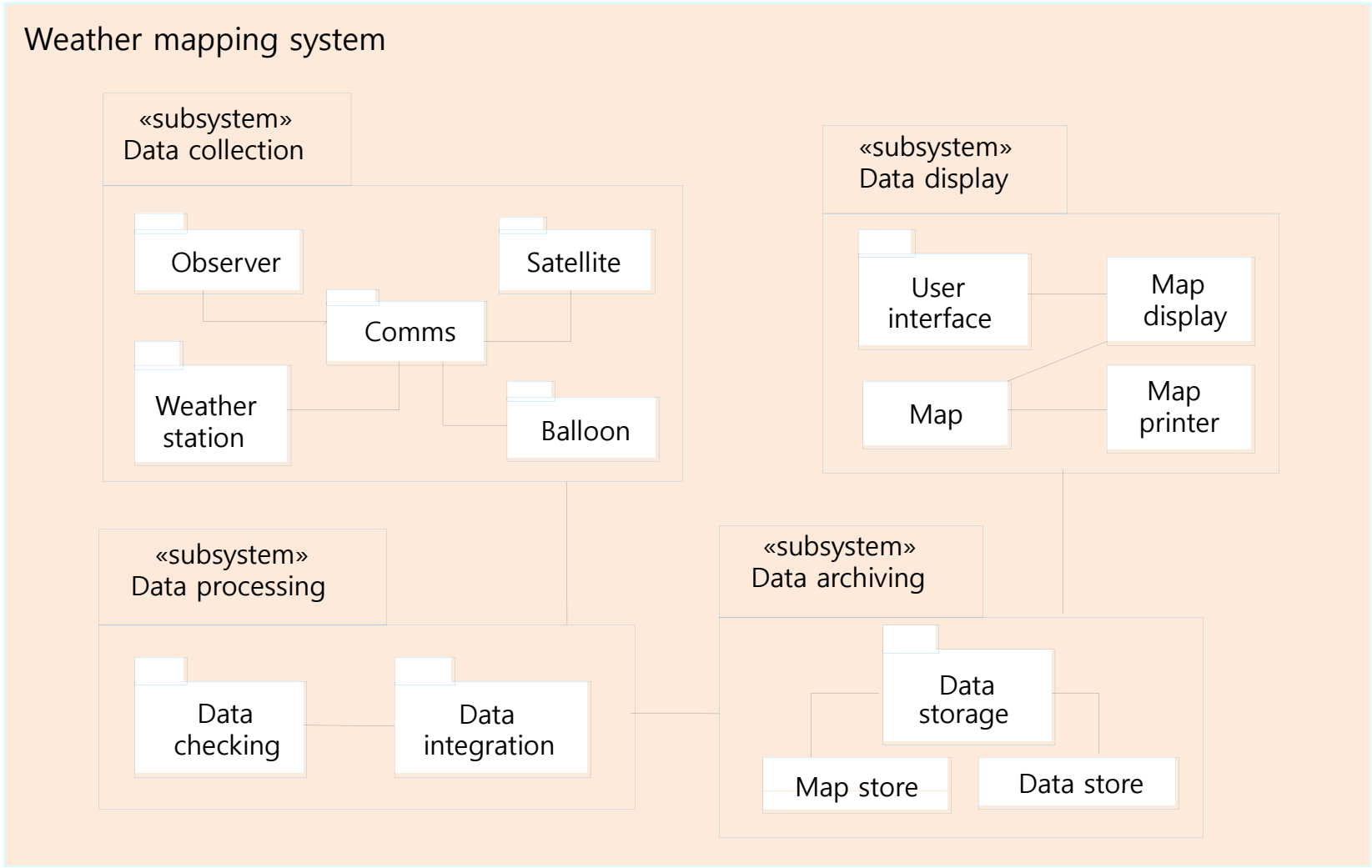
1. System Context and Models of System Use

- Develop an understanding of the relationships between the software being designed and its external environment

- System context
  - A static model that describes other systems in the environment
  - Use a subsystem model to show other systems

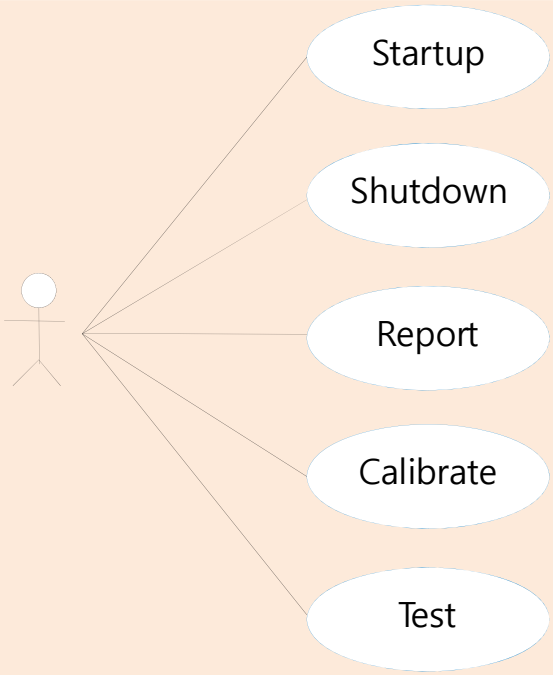
- Model of system use
  - A dynamic model that describes how the system interacts with its environment
  - Use use-cases to show interactions

Subsystem Model



Use-Case Model

Weather station use-case



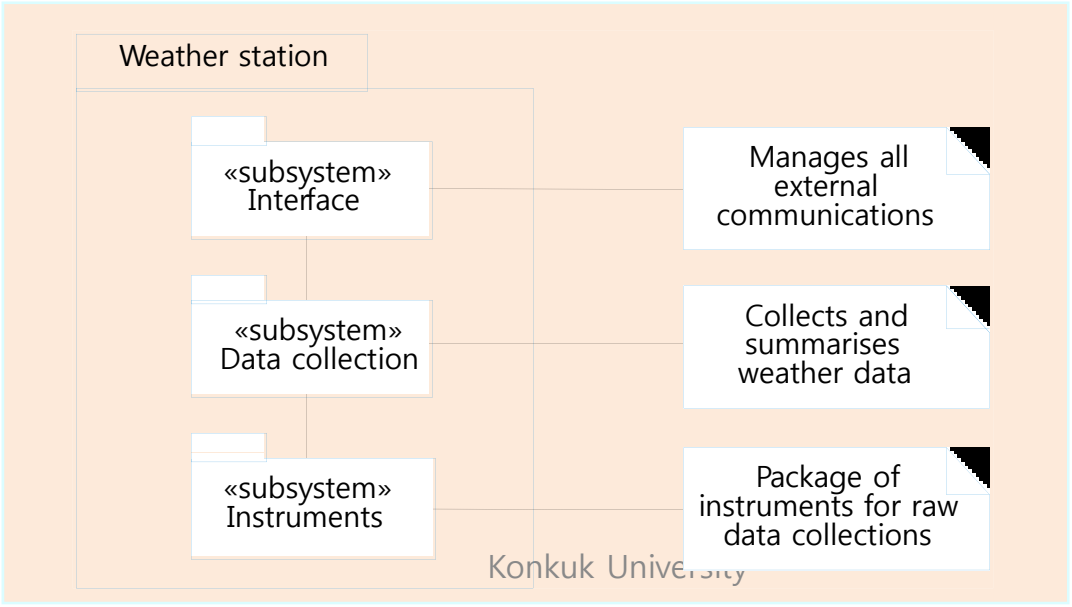
Use-case description

System	Weather station
Use-case	Report
Actors	Weather data collection system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather data collection 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 5 minute intervals.
Stimulus	The weather data collection system establishes a modem link with the weather station and requests transmission of the data.
Response	The summarised data is sent to the weather data collection system
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to the other and may be modified in future.

2. Architectural Design

- Design the system architecture using the understanding about the interactions between the system and its environment.

- A layered architecture is appropriate for the weather station
  - Interface layer for handling communications
  - Data collection layer for managing instruments
  - Instruments layer for collecting data

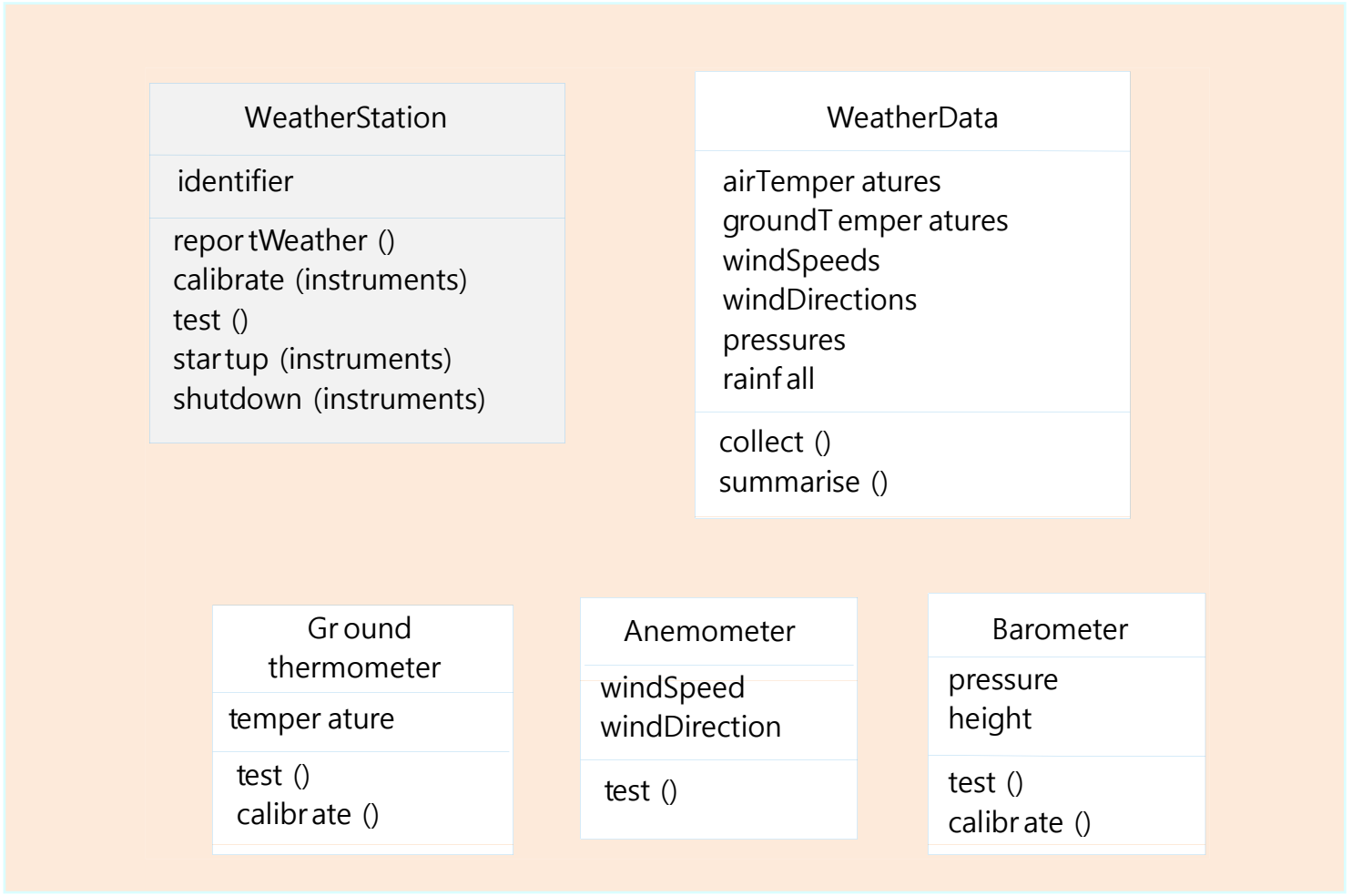


3. Object Identification

- Identifying objects (or object classes) is the most difficult part of object oriented design.
  - No 'magic formula' for object identification.
  - Relies on the skill, experience and domain knowledge of system designers
  - An iterative process

- Approaches to object identification:
  - Use a grammatical approach based on a natural language description of the system (used in Hood OOD method)
  - Based on the identification on tangible things in the application domain
  - Use a behavioural approach and identify objects based on what participates in what behaviour
  - Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

Weather Station Object Classes



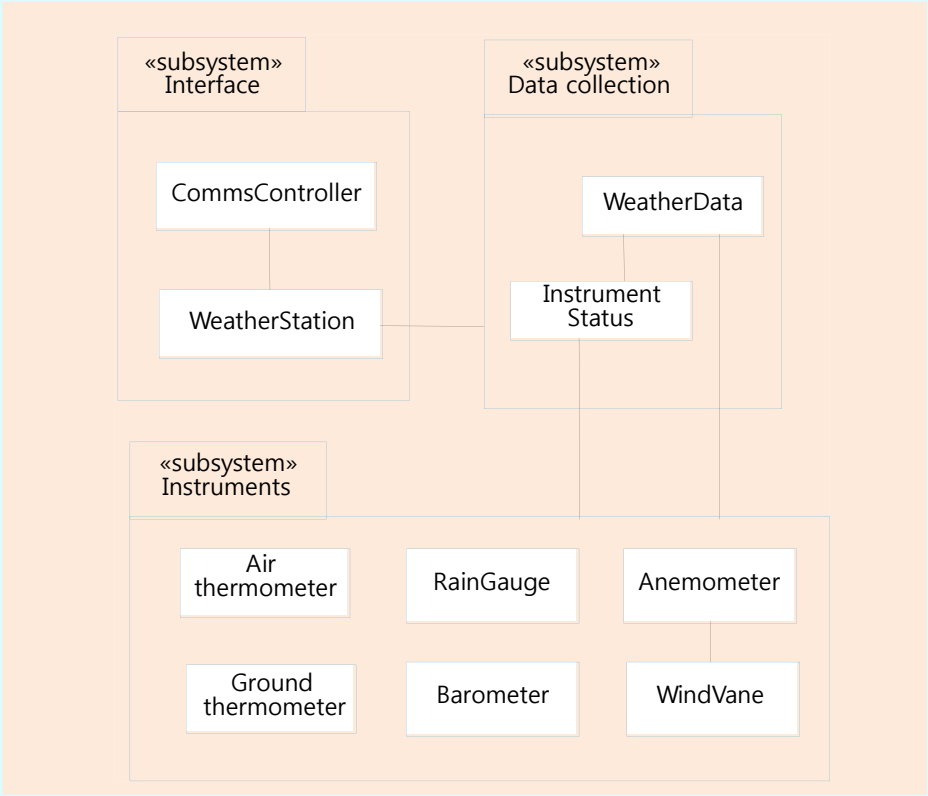
4. Developing Design Model

- Design models show the objects, object classes and relationships between these entities.
  - Static models describe the static structure of the system in terms of object classes and relationships.
  - Dynamic models describe the dynamic interactions between objects.

- Examples of design models:
  - Sub-system model : shows logical groupings of objects into coherent subsystems
  - Sequence model : shows the sequence of object interactions
  - State machine model : show how individual objects change their state in response to events.
  - Other models include use-case models, aggregation models, generalisation models, etc.

Subsystem Model

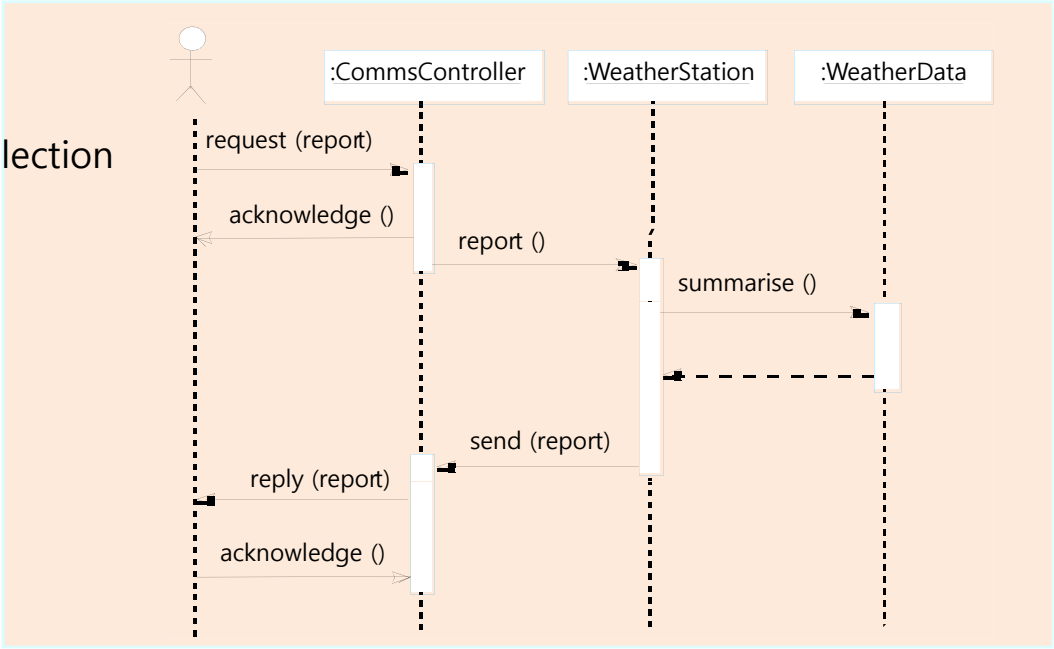
- Show how the design is organized into logically related groups of objects.
  - A logical model
  - The actual organization of objects may be different.
  - In the UML, these are shown using packages



Sequence Model

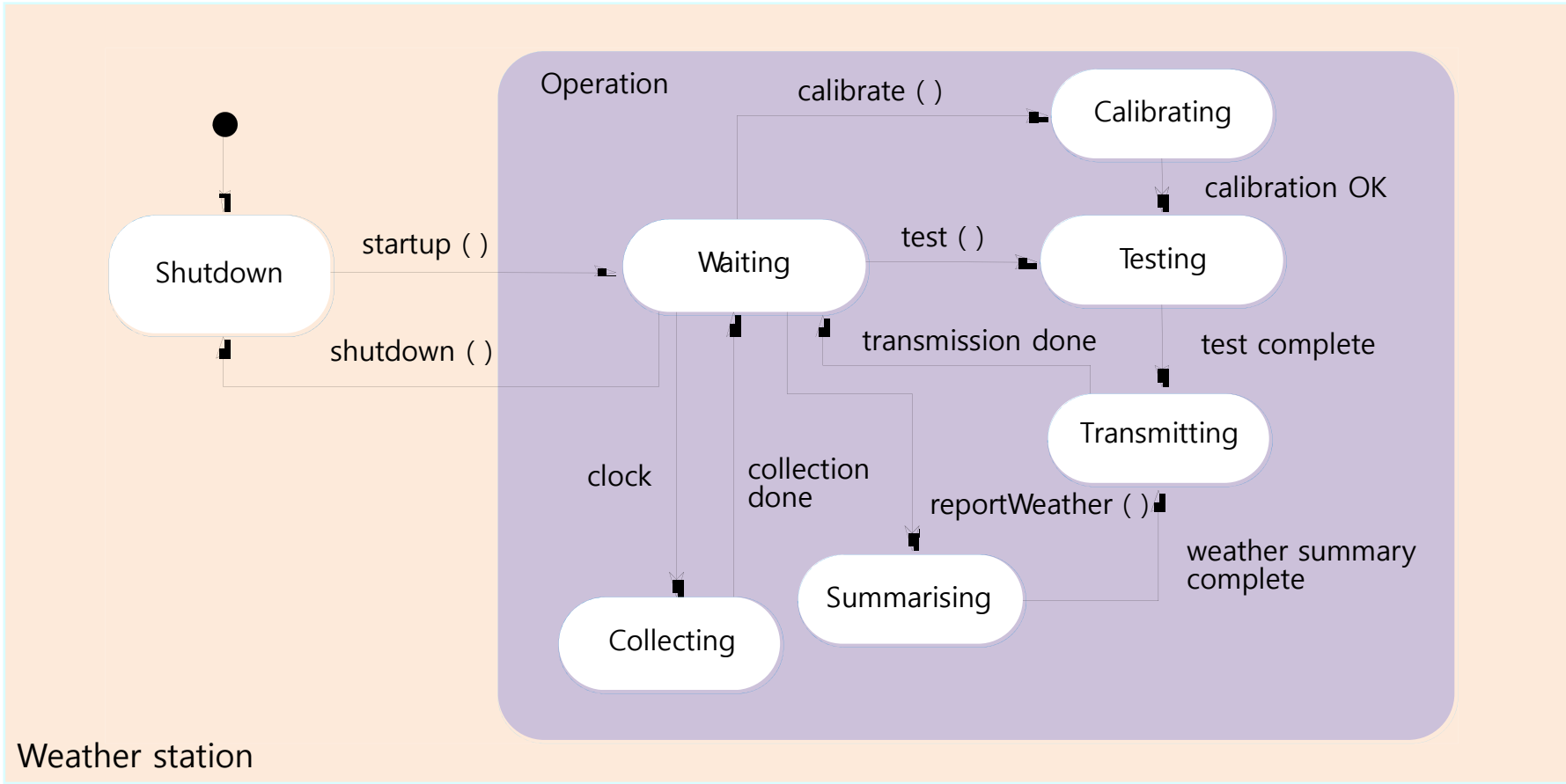
- Show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top.
  - Time is represented vertically, so models are read top to bottom.
  - Interactions are represented by labelled arrows.
  - Different styles of arrow represent different types of interaction.
  - Thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

Data collection



State Machine Model: Statecharts

- Show how objects respond to different service requests and the state transitions triggered by these requests



5. Object Interface Specification

- Object interfaces specification make the design of objects and other components performed in parallel.
  - Objects may have several interfaces (viewpoints).
  - The UML uses class diagram for interface specification

```
interface WeatherStation {

    public void WeatherStation () ;

    public void startup () ;
    public void startup (Instrument i) ;

    public void shutdown () ;
    public void shutdown (Instrument i) ;

    public void reportWeather ( ) ;

    public void test () ;
    public void test ( Instrument i ) ;

    public void calibrate ( Instrument i) ;

    public int getID () ;

} //WeatherStation
```

Summary

- OOD is an approach to design so that design components have their own private state and operations.
- Objects should have constructor and inspection operations. They provide services to other objects.
- Objects may be implemented sequentially or concurrently.
- The Unified Modelling Language provides different notations for defining different object models.
- A range of different models may be produced during an object-oriented design process. These include static and dynamic system models.
- Object interfaces should be defined precisely using a programming language like Java.

Chapter 15.

Real-Time Software Design

Objectives

- To explain the concept of a real-time system and why these systems are usually implemented as concurrent processes
- To describe a design process for real-time systems
- To explain the role of real-time operating systems
- To introduce generic process architectures for monitoring and control and data acquisition systems

Real-Time systems

- Systems which monitor and control their environment
- Inevitably associated with hardware devices
  - Sensors : collect data from the system environment
  - Actuators : change the system's environment (in some way)
- Time is critical.
  - Real-time systems MUST respond within specified times.

Definition

- Real-time system is a software system where the correct functioning of the system depends on
  - the results produced by the system and
  - the time at which these results are produced

- Soft real-time system
  - Operation is degraded if results are not produced according to the specified timing requirements.

- Hard real-time system
  - Operation is incorrect if results are not produced according to the timing specification.

Stimulus/Response Systems

- Given a stimulus, the system must produce a response within a specified time.

- Periodic stimuli
  - Stimuli which occur at predictable time intervals
  - Example: a temperature sensor may be polled 10 times per second.

- Aperiodic stimuli
  - Stimuli which occur at unpredictable times
  - Example: a system power failure may trigger an interrupt which must be processed by the system.

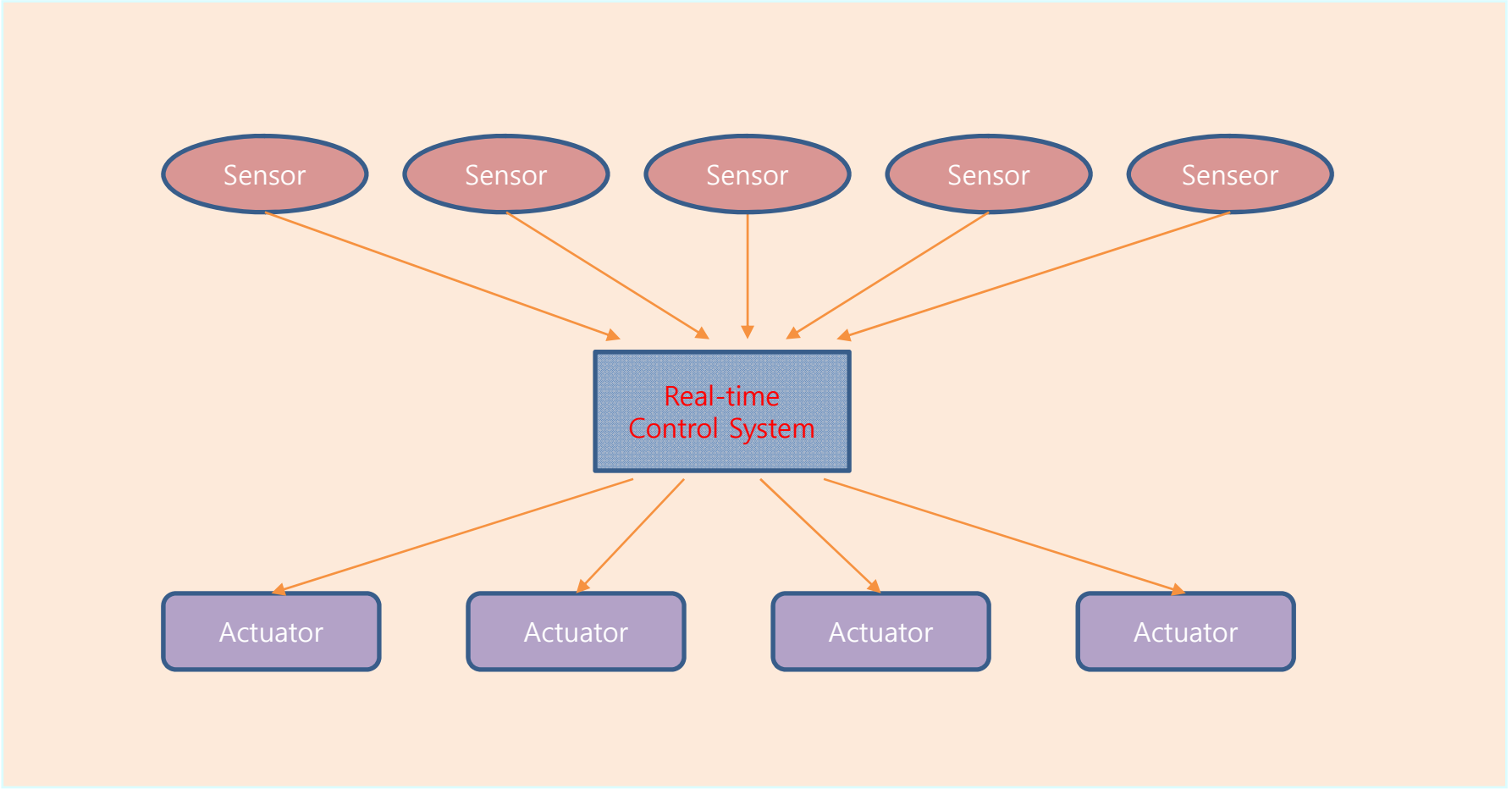
Architectural Considerations

- Because of the need to respond to timing demands made by different stimuli/responses, the system architecture must allow for fast switching between stimulus handlers.

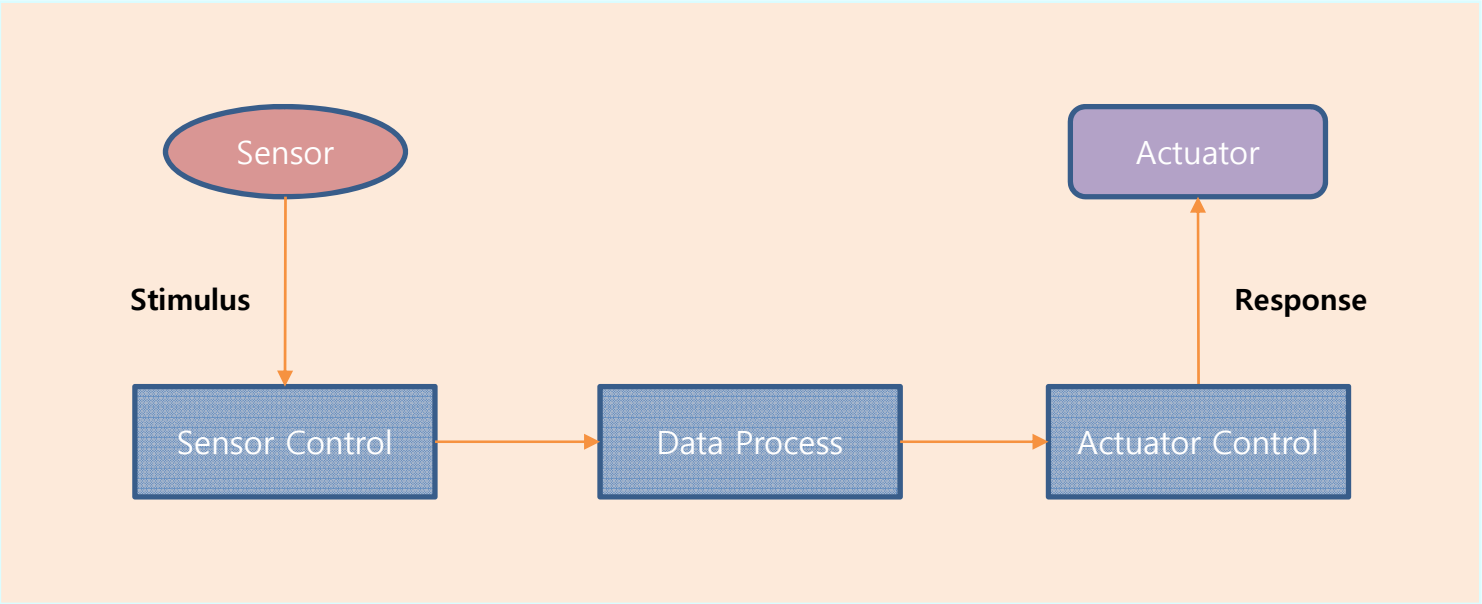
- Timing demands of different stimuli are different so a simple sequential loop is not usually adequate.

- Real-time systems are therefore usually designed as cooperating processes with a real-time executive controlling these processes.
  - Cooperating processes
  - One real-time executive

A Real-Time System Model



Sensor/Actuator Processes



System Elements

- Sensor control processes
  - Collect information from sensors
  - May buffer collected information in response to a sensor stimulus.
- Data processor
  - Carries out processing of collected information
  - Computes the system response
- Actuator control processes
  - Generates control signals for the actuators

Real-Time Programming

- Hard-real time systems may have to programmed in assembly language to ensure that deadlines are met.
  - Languages such as C allow efficient programs to be written, but do not have constructs to support concurrency or shared resource management.
  - Java supports lightweight concurrency (threads and synchronized methods) and can be used for some soft real-time systems.

- Real-time versions of Java are now available addressing problems like
  - Not possible to specify thread execution time
  - Different timing in different virtual machines
  - Uncontrollable garbage collection
  - Not possible to discover queue sizes for shared resources
  - Not possible to access system hardware
  - Not possible to do space or timing analysis

System Design

- Design both the hardware and the software associated with system
  - Partition functions to either hardware or software
  - Design decisions should be made on the basis on non-functional system requirements.

  - Hardware delivers better performance but potentially longer development and less scope for change.

Real-Time Systems Design Process

1. Identify the stimuli to be processed and the required responses to these stimuli.
2. For each stimulus and response, identify the timing constraints.
3. Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response.
4. Design algorithms to process each class of stimulus and response. These must meet the given timing requirements.
5. Design a scheduling system which will ensure that processes are started in time to meet their deadlines.
6. Integrate using a real-time operating system.

Timing Constraints

- May require extensive simulation and experiment to ensure that these are met by the system
- May mean that certain design strategies such as object-oriented design cannot be used because of the additional overhead involved
- May mean that low-level programming language features have to be used for performance reasons

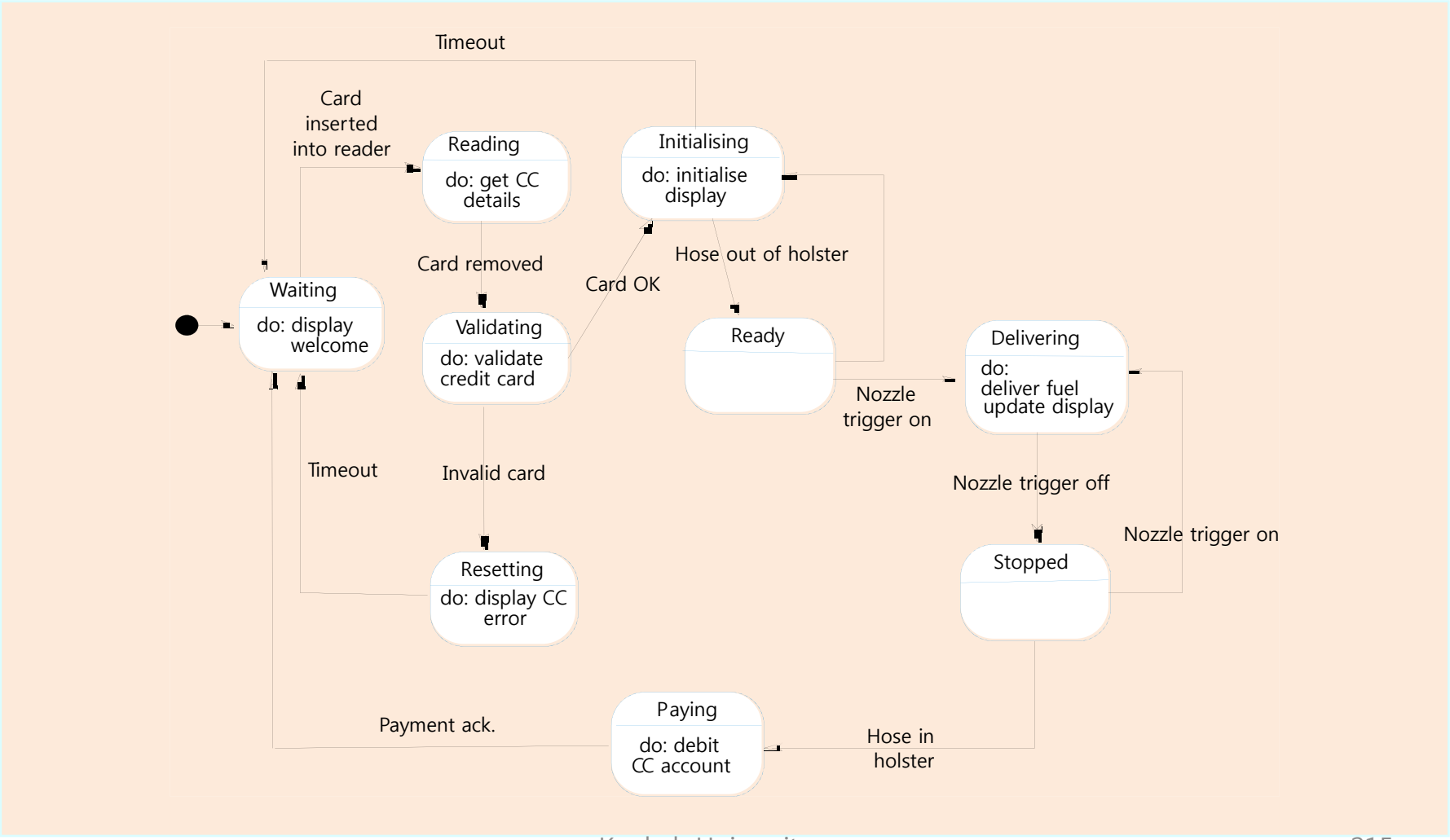
Real-Time System Modelling

- The effect of a stimulus in a real-time system may trigger a transition from one state to another.

- Finite State Machines (FSM) can be used for modelling real-time systems.
  - However, FSM models lack structure. Even simple systems can have complex models.
  - The UML includes notations for defining state machine models.

- See Chapter 8 for further examples of state machine models.

Petrol Pump State Model

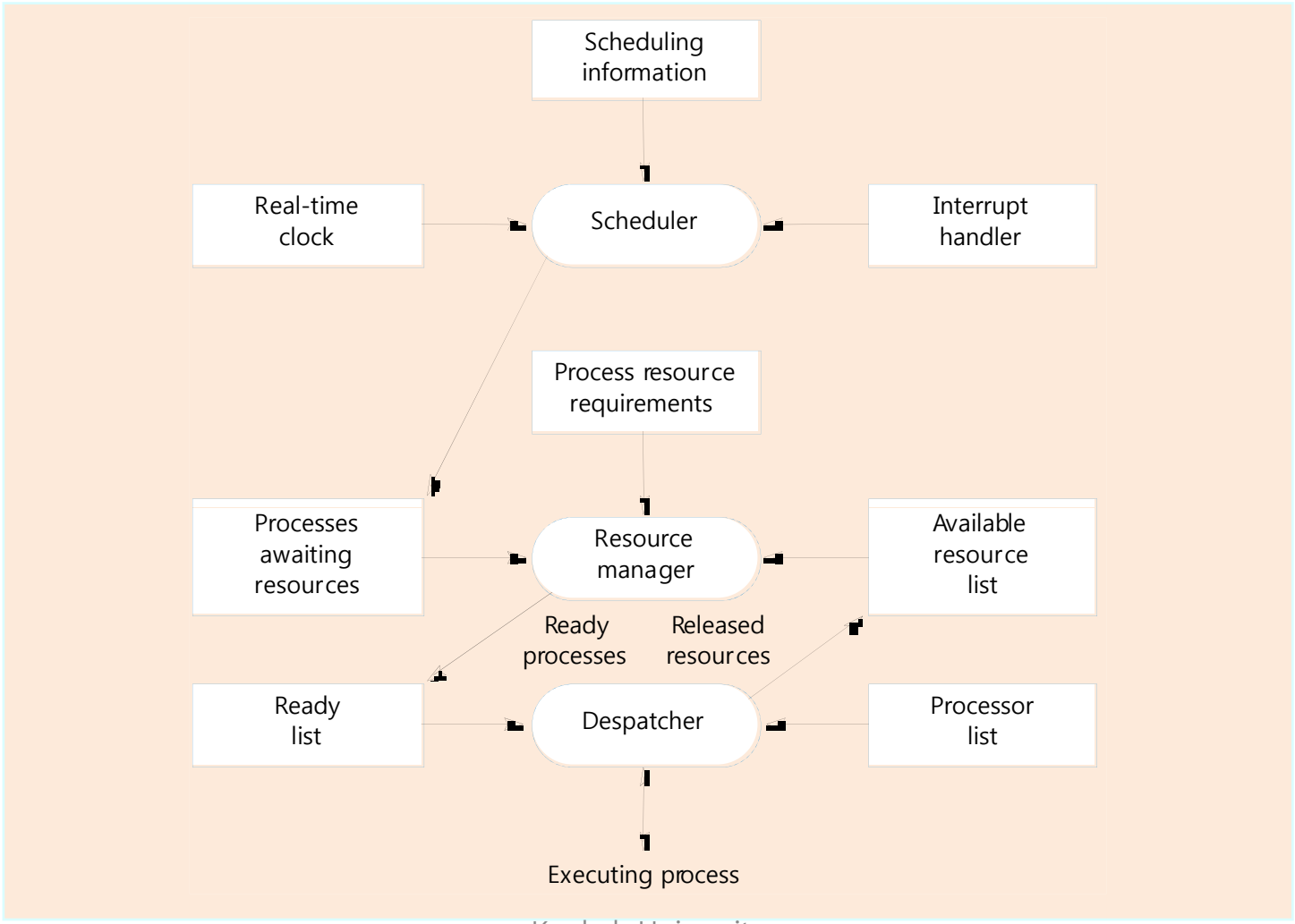


Real-Time Operating Systems

- Real-time operating systems are specialized operating systems which manage the processes in the RTS.
  - Responsible for process management and resource (processor and memory) allocation
  - May be based on a standard kernel which is used unchanged or modified for a particular application
  - Do not normally include facilities such as file management

- Real-time operating system components
  - Real-time clock : provides information for process scheduling
  - Interrupt handler : manages aperiodic requests for service
  - Scheduler : chooses the next process to be run
  - Resource manager : allocates memory and processor resources
  - Dispatcher : starts process execution

Real-Time OS Components



Non-Stop System Components

- Configuration manager
  - Responsible for the dynamic reconfiguration of the system software and hardware.
  - Hardware modules may be replaced and software upgraded without stopping the systems.

- Fault manager
  - Responsible for detecting software and hardware faults and taking appropriate actions (e.g. switching to backup disks)
  - To ensure that the system continues in operation

Process Priority

- The processing of some types of stimuli must sometimes take priority.
  - Interrupt level priority
    - Highest priority
    - Allocated to processes requiring a very fast response
  - Clock level priority
    - Allocated to periodic processes
- Within these, further levels of priority may be assigned.

Interrupt Servicing

- Control is transferred automatically to a pre-determined memory location.
  - This location contains an instruction to jump to an interrupt service routine.
  - Further interrupts are disabled, the interrupt serviced and the control returned to the interrupted process.

- Interrupt service routines MUST be short, simple and fast.

Periodic Process Servicing

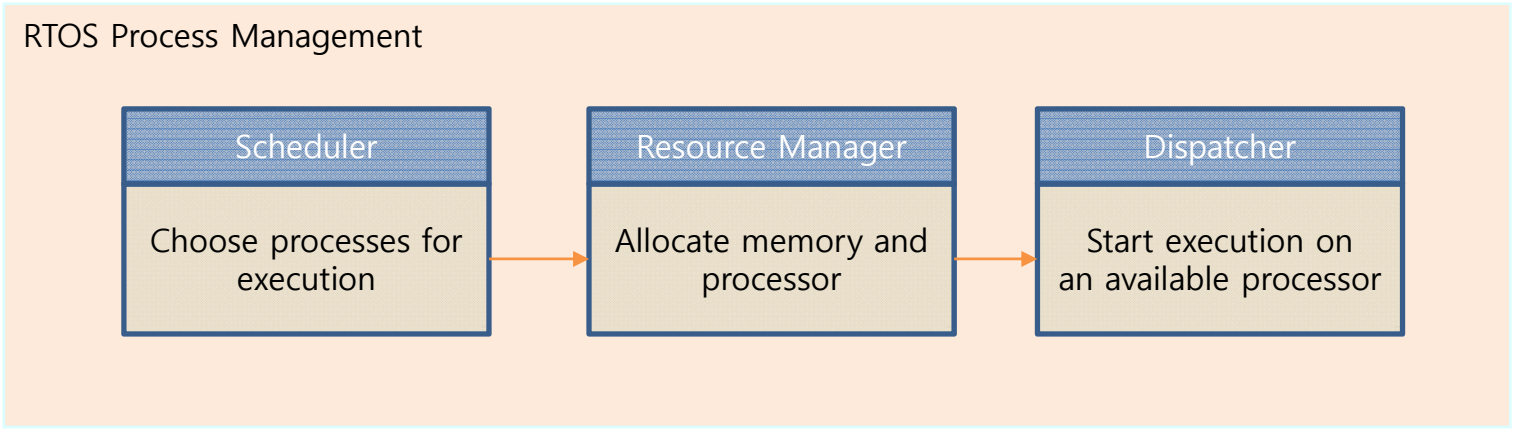
- In most real-time systems, there will be several classes of periodic process, each with different periods (the time between executions), execution times and deadlines (the time by which processing must be completed).

- The real-time clock ticks periodically and each tick causes an interrupt which schedules the process manager for periodic processes.
- The process manager selects a process which is ready for execution.

Process Management

- Concerned with managing the set of concurrent processes.
- Periodic processes are executed at pre-specified time intervals.

- The RTOS uses the real-time clock to determine when to execute a process taking into account
  - Process period : time between executions.
  - Process deadline : the time by which processing must be complete.



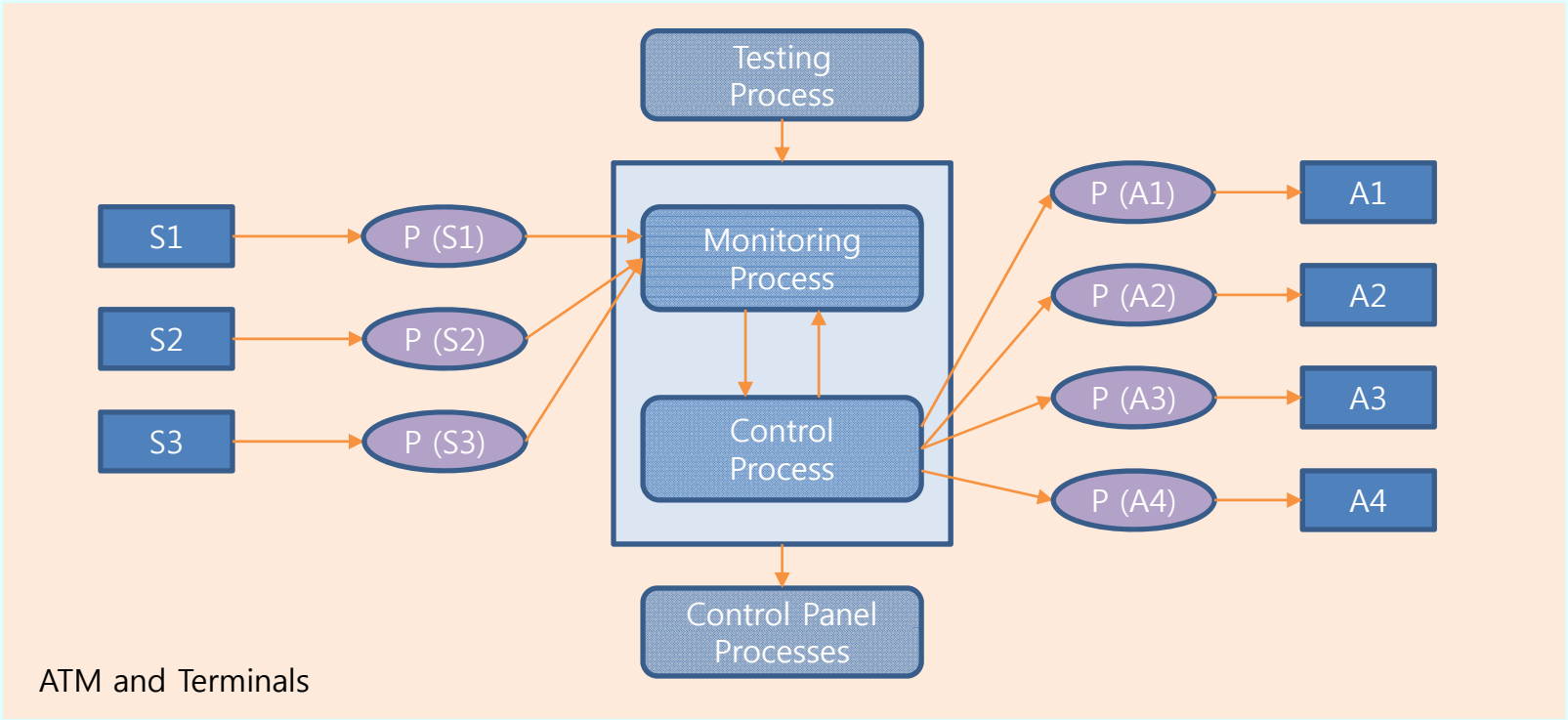
Process Switching

- The scheduler chooses the next process to be executed by the processor.
  - Depends on a scheduling strategy.
- The resource manager allocates memory and a processor for the process to be executed.
- The dispatcher takes the process from ready list, loads it onto a processor and starts execution.

- Scheduling strategies
  - Non pre-emptive scheduling
    - Once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O).
  - Pre-emptive scheduling
    - The execution of an executing processes may be stopped if a higher priority process requires service.
  - Scheduling algorithms
    - Round-robin , Rate monotonic , Shortest deadline first, many others.

Monitoring and Control Systems

- Continuously check sensors and take actions depending on sensor values.
- Monitoring systems examine sensors and report their results.
- Control systems take sensor values and control hardware actuators.



ATM and Terminals

Summary

- Real-time system correctness depends not just on what the system does but also on how fast it reacts.
- A general real-time system model involves associating processes with sensors and actuators.
- Real-time systems architectures are usually designed as a number of concurrent processes.
- Real-time operating systems are responsible for process and resource management.
- Monitoring and control systems poll sensors and send control signal to actuators.

Part IV. Development

Chapter 17.

Rapid Software Development

Objectives

- To explain how an iterative and incremental development process lead to faster delivery of more useful software
- To discuss the essence of agile development methods
- To explain the principles and practices of extreme programming
- To explain the roles of prototyping in software process

Rapid Software Development

- Rapidly changing business environments
  - make businesses have to respond to new opportunities and competition
  - Require rapid software development

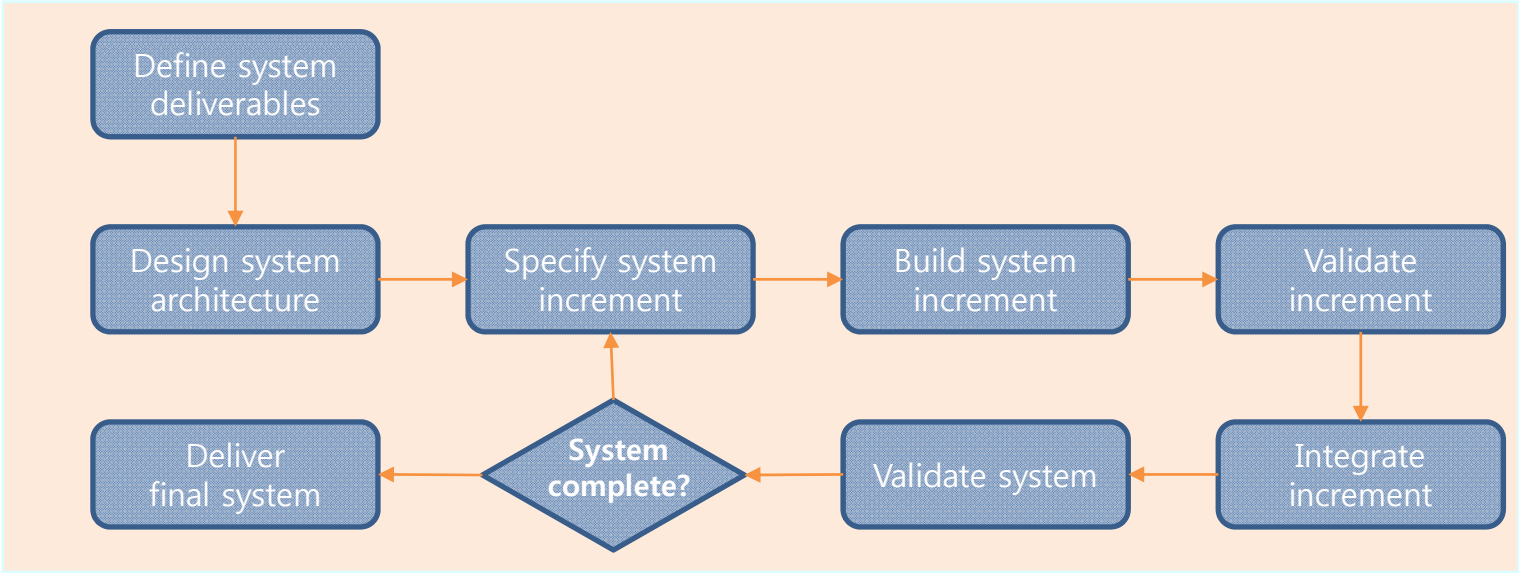
- Businesses may be willing to accept lower quality software if rapid delivery of essential functionality is possible.

- Because of the changing environment, it is often impossible to arrive at a stable and consistent set of system requirements.
- Therefore a waterfall model of development is impractical.

- Approach to development based on iterative specification and delivery is the only way to deliver software quickly.

Characteristics of Rapid Software Development Process

- System is developed in a series of increments.
  - Specification, design and implementation are performed concurrently.
  - End users evaluate each increment and make proposals for later increments.
  - No detailed specification and design documentation



Characteristics of Incremental Development

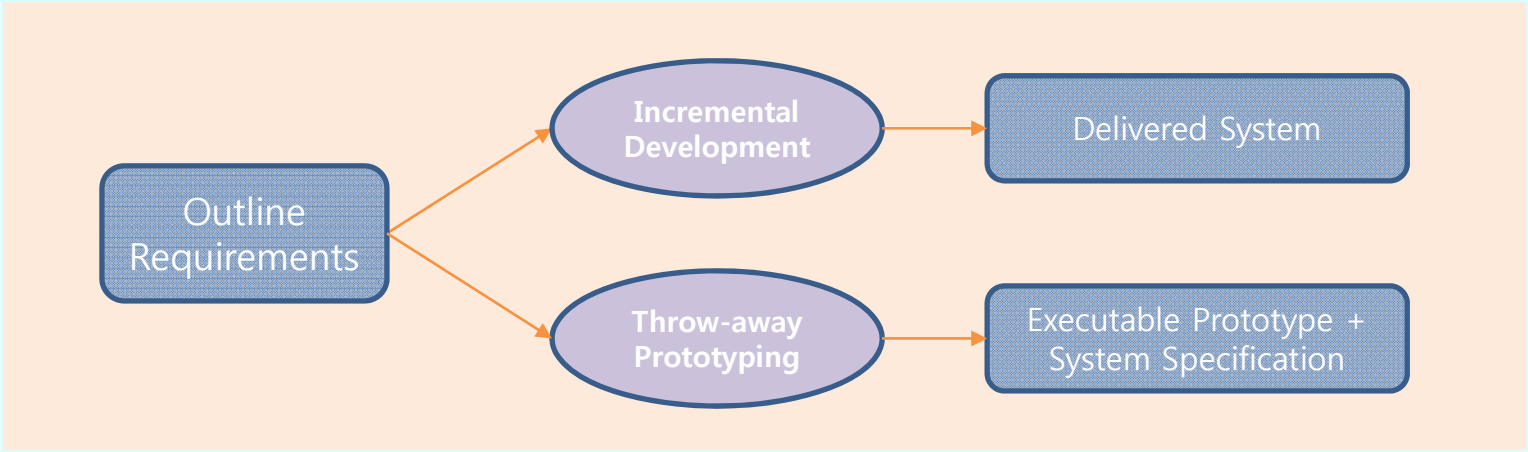
- Advantages:
  - Accelerated delivery of customer services
    - Each increment delivers the highest priority functionality to the customer.
  - User engagement with the system
    - Users have to be involved in the development to specify their requirements.

- Problems:
  - Management problems
    - No document makes the progress hard to be judged and problems hard to be found.
  - Contractual problems
    - The normal contract may include a specification, but it does not have it.
  - Validation problems
    - Without a specification, what is the system being tested against?
  - Maintenance problems
    - Continual change tends to corrupt software structure, and makes it more expensive to change and evolve to meet new requirements.

Prototyping

- For some large systems, incremental iterative development and delivery may be impractical.

- An experimental system is developed
  - as a basis for formulating the requirements, and
  - thrown away, when the system specification has been agreed.



Differences in Objectives

- Incremental development
  - To deliver a working system to end-users
  - Start with those requirements which are best understood
  - Example: Agile, XP

- Throw-away prototyping
  - To validate or derive system requirements.
  - Starts with those requirements which are poorly understood
  - Example: Prototyping

Agile Method

- From dissatisfaction with the overheads involved in design methods
  - Focus on the code rather than the design
  - Based on an iterative approach to software development
  - Intended to deliver working software quickly
  - Intended to evolve software quickly to meet changing requirements
  - Best suited to small/medium-sized business systems or PC products

Objectives	Description
Customer involvement	The customer should be closely involved throughout the development process. Their role is provide and prioritise 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 recognised and exploited. The team should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and design the system so that it can accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process used. Wherever possible, actively work to eliminate complexity from the system.

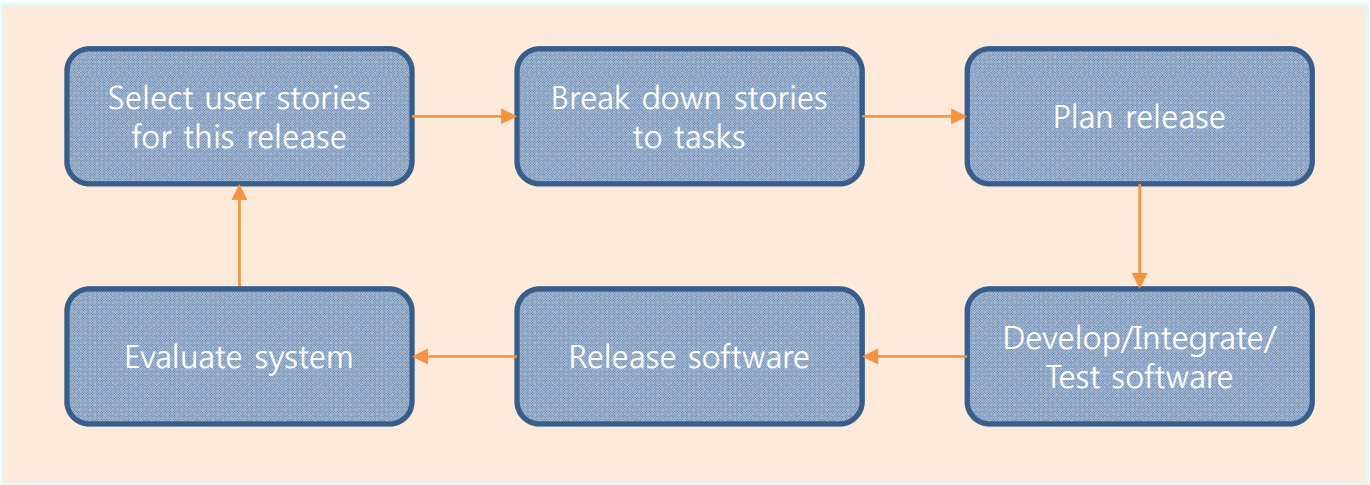
Problems with Agile Method

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

Extreme Programming

- Extreme Programming (XP) is the best-known agile method.
  - 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 release cycle:



Testing in XP

- XP is a test-first development.
  - Incremental tests are developed from scenarios.
  - Users are involved in test development and validation.
  - Automated test harnesses are used to run all component tests each time that a new release is built.

- Test-first development
  - Writing tests before code to clarify the requirements to be implemented.
  - Tests are written as programs rather than data so that they can be executed automatically.
  - Test includes a check that it has executed correctly.
  - All previous and new tests are automatically run when new functionality is added in order to check that the new functionality has not introduced errors.

Pair Programming in XP

- In XP, programmers work in pairs, sitting together to develop code.
  - Helps develop common ownership of code
  - Help spread 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 this
- Development productivity with pair programming is similar to that of two people working independently.

RAD (Rapid Application Development)

- Other RAD approaches except Agile methods have been used for many years.
  - Designed to develop data-intensive business applications
  - Rely on programming and presenting information from a database
  - RAD environment:
    - Database programming language
    - Interface generator
    - Links to office applications
    - Report generators

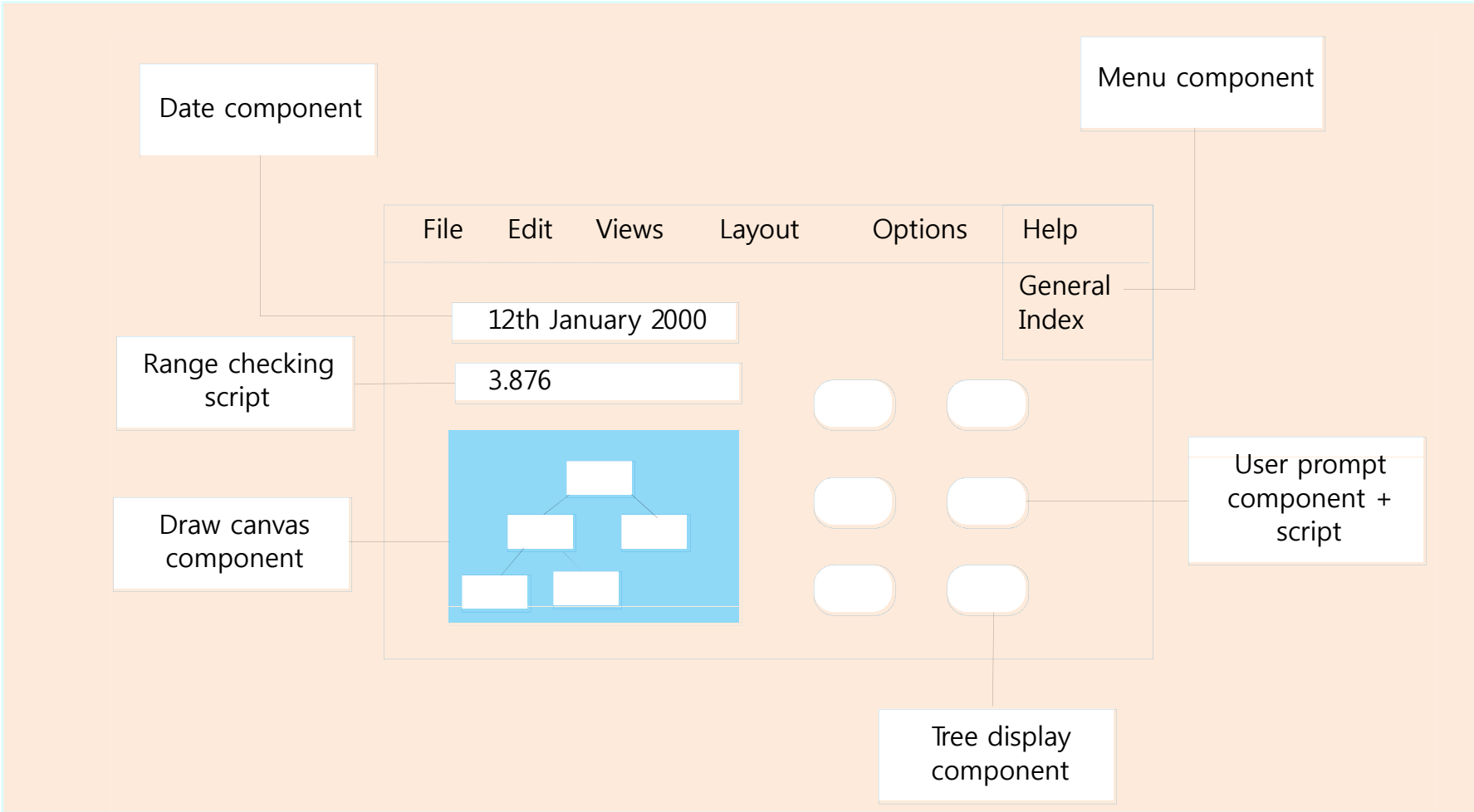
Interface Generation

- Many applications are based on complex forms
  - Developing forms manually is a time-consuming activity.

- RAD environments include support for screen generation including
  - Interactive form definition using drag and drop techniques
  - Form linking where the sequence of forms to be presented is specified
  - Form verification where allowed ranges in form fields is defined

- Visual Programming
  - Scripting languages such as Visual Basic support visual programming where the prototype is developed by creating a user interface from standard items and associating components with these items.
  - A large library of components exists to support this type of development.
  - May be tailored to suit the specific application requirements.

Visual Programming with Reuse



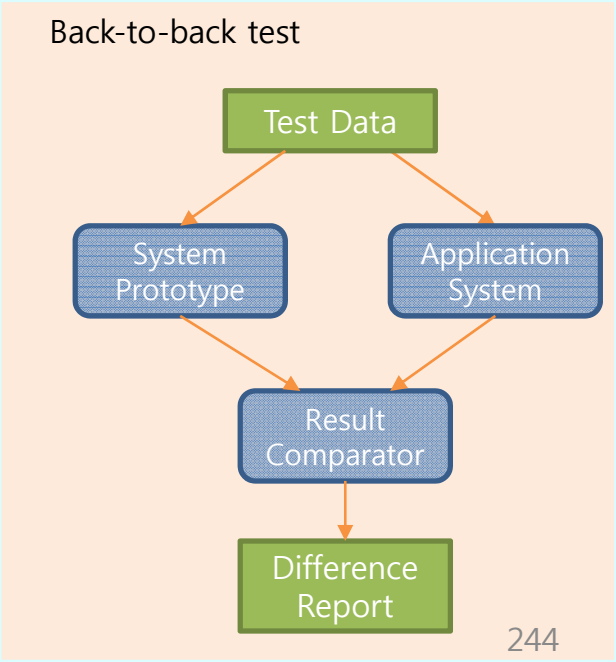
COTS Reuse

- An effective approach to rapid development is to configure and assemble existing off-the-shelf systems.

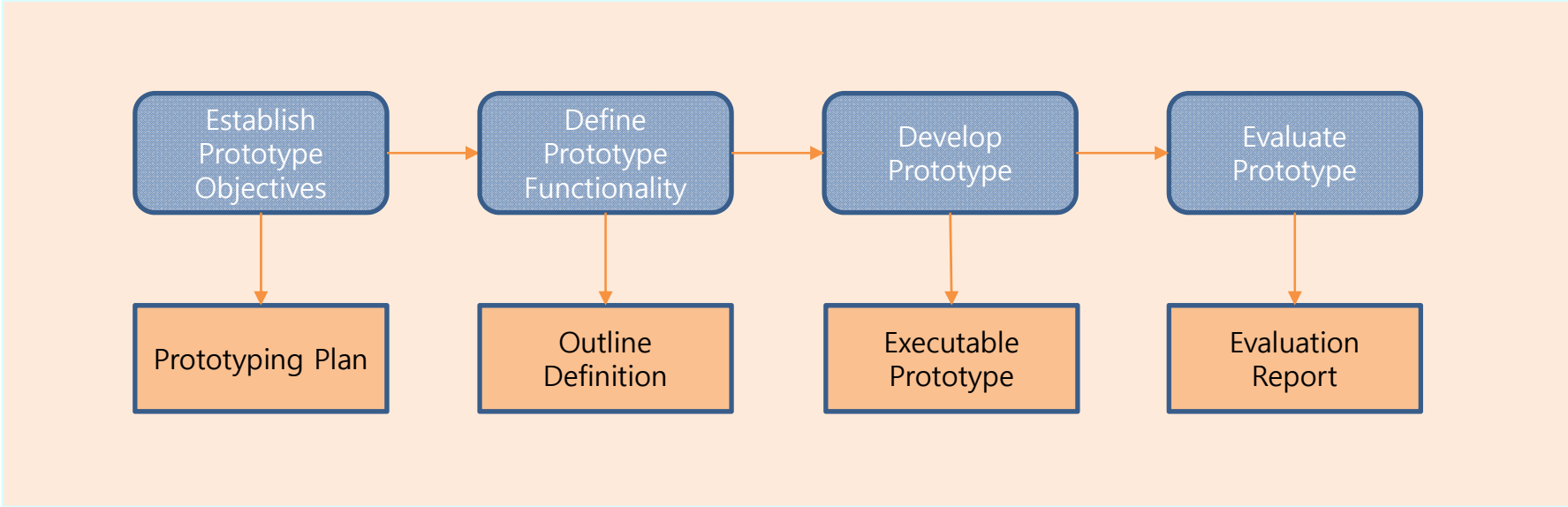
- For example, a requirements management system could be built by using
  - A database to store requirements
  - A word processor to capture requirements and format reports
  - A spreadsheet for traceability management

Software Prototyping

- A prototype is an initial version of a system used to demonstrate concepts and try out design options. (Throw-away prototyping)
- A prototype can be used
  - In the requirements engineering process to help with requirements elicitation and validation
  - In design processes to explore options and develop a UI design
  - In the testing process to run back-to-back tests
- Benefits of prototyping
  - Improved system usability
  - A closer match to users' real needs
  - Improved design quality
  - Improved maintainability
  - Reduced development effort



Prototyping Process



Summary

- An iterative approach to software development leads to faster delivery of software.
- Agile methods are iterative development methods that aim to reduce development overhead and so produce software faster.
- Extreme programming includes practices such as systematic testing, continuous improvement and customer involvement.
- Testing approach in XP is a particular strength where executable tests are developed before the code is written.
- Rapid application development (RAD) environments include database programming languages, form generation tools and links to office applications.
- A throw-away prototype is used to explore requirements and design options.

Chapter 18.

Software Reuse

Objectives

- To explain benefits of software reuse and some reuse problems
- To discuss several different ways to implement software reuse
- To explain how reusable concepts can be represented as patterns or embedded in program generators
- To discuss COTS reuse
- To describe the development of software product lines

Software Reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.
- To achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on systematic software reuse.

- Reuse-based software Engineering
  - Application system reuse
    - The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families.
  - Component reuse
    - Components of an application from sub-systems to single objects may be reused. Covered in Chapter 19.
  - Object and function reuse
    - Software components that implement a single well-defined object or function may be reused.

Benefits of Reuse

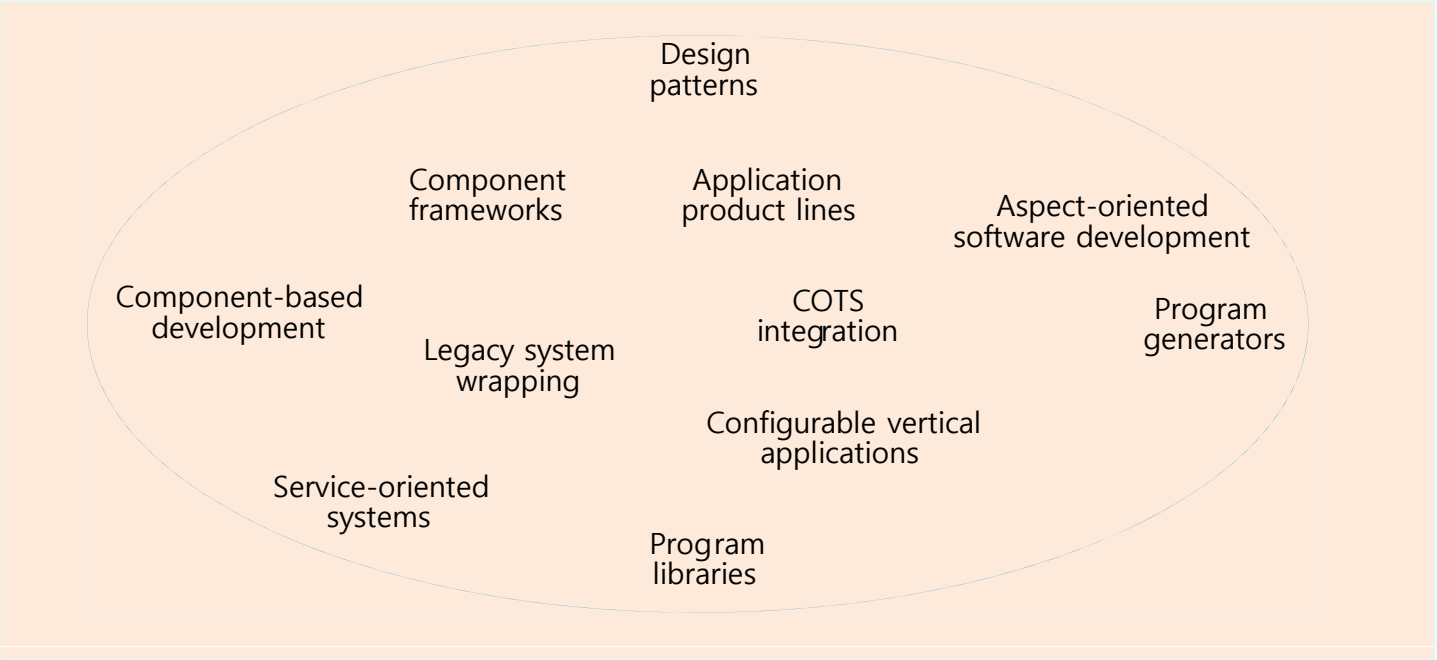
Benefits	Description
Increased dependability	Reused software, that has been tried and tested in working systems, should be more dependable than new software. The initial use of the software reveals any design and implementation faults. These are then fixed, thus reducing the number of failures when the software is reused.
Reduced process risk	If software exists, there is less uncertainty in the costs of reusing that software than in the costs of development. This is an important factor for project management as it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as sub-systems are reused.
Effective use of specialists	Instead of application specialists doing the same work on different projects, these specialists can develop reusable software that encapsulate their knowledge.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interfaces are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability as users are less likely to make mistakes when presented with a familiar interface.
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced.

Problems in Reuse

Problems	Description
Increased maintenance cost	If the source code of a reused software system or component is not available then maintenance costs may be increased as the reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account.
Not-invented-here syndrome	Some software engineers sometimes prefer to re-write components as they believe that they can improve on the reusable component. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.
Creating and maintaining a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.
Finding, understanding and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they will make routinely include a component search as part of their normal development process.

Reuse Landscape

- Reuse is possible at a range of levels from simple functions to complete application systems.
- The reuse landscape covers the range of possible reuse techniques.



Reuse Approaches

Reuse Approaches	Description
Design patterns	Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions.
Component-based development	Systems are developed by integrating components (collections of objects) that conform to component-model standards. This is covered in Chapter 19.
Application framework	Collections of abstract and concrete classes that can be adapted and extended to create application systems.
Legacy system wrapping	Legacy systems (see Chapter 2) that can be 'wrapped' by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Service-oriented systems	Systems are developed by linking shared services that may be externally provided.
Application product lines	An application type is generalised around a common architecture so that it can be adapted in different ways for different customers.
COTS integration	Systems are developed by integrating existing application systems.
Configurable vertical applications	A generic system is designed so that it can be configured to the needs of specific system customers.
Program libraries	Class and function libraries implementing commonly-used abstractions are available for reuse.
Program generators	A generator system embeds knowledge of a particular types of application and can generate systems or system fragments in that domain.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled.

Reuse: Concept Reuse

- When reuse program or design components, we have to follow the design decisions made by the original developer of the component.
  - May limit the opportunities for reuse

- Concept reuse is a more abstract form of reuse.
  - A particular approach is described in an implementation independently.
  - An implementation is then developed.

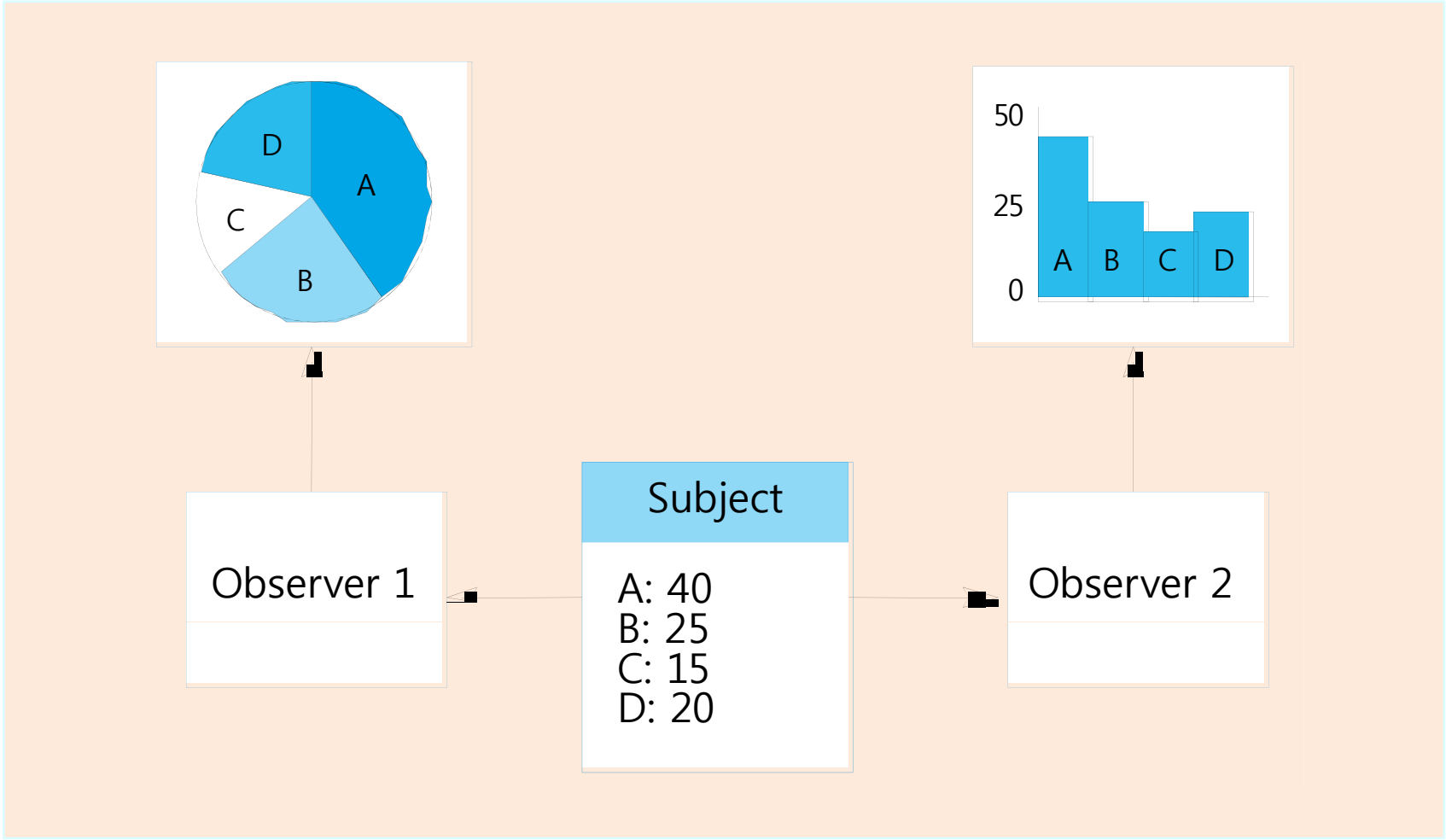
- Two main approaches to concept reuse
  - Design patterns
  - Generative programming (Program generator)

Design Pattern

- Design pattern is a way of reusing abstract knowledge about a problem and its solution.
  - A pattern is a description of the problem and the essence of its solution.
  - Should be sufficiently abstract to be reused in different settings
  - Patterns often rely on object characteristics such as inheritance and polymorphism.

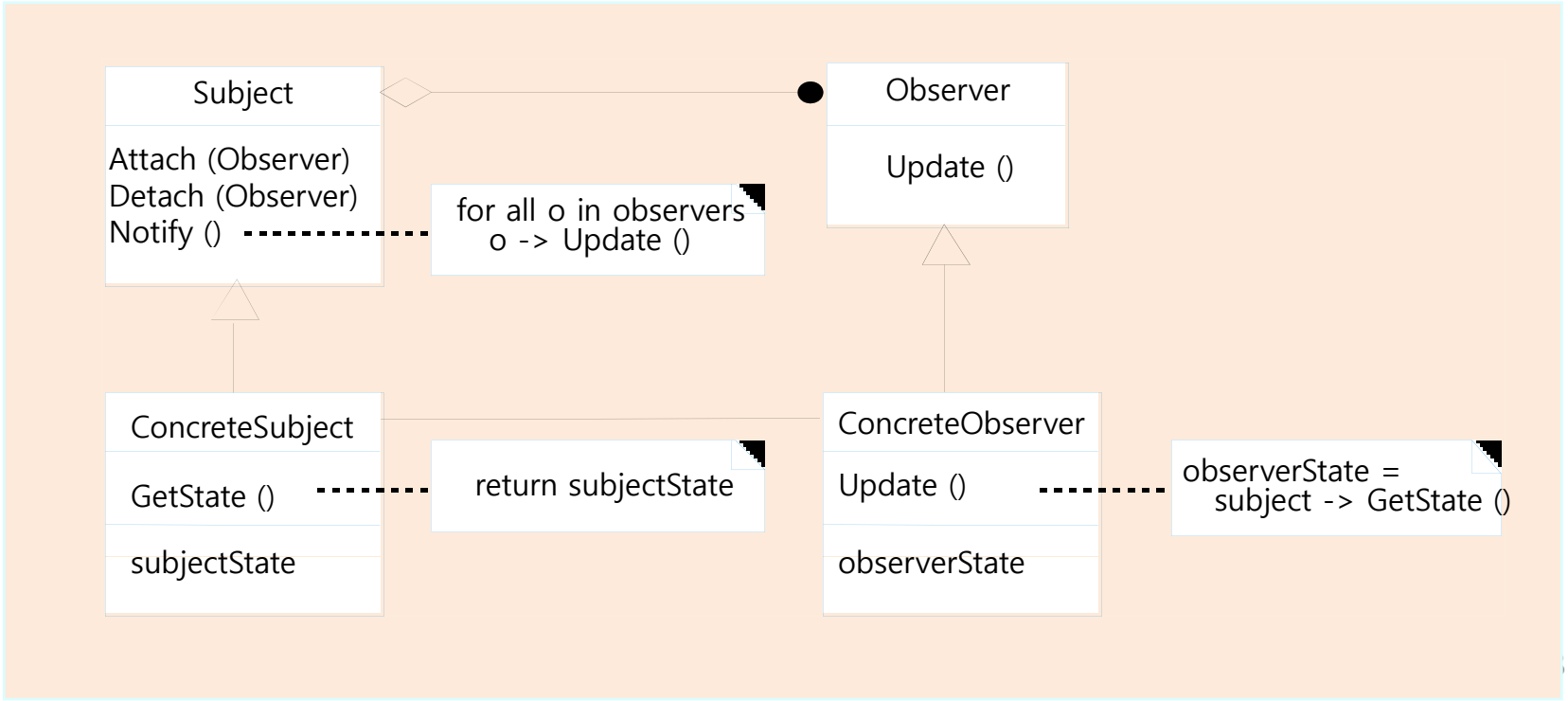
- Elements in design patterns
  - Name : Meaningful pattern identifier
  - Problem description
  - Solution description : Not a concrete design but a template for a design solution that can be instantiated in different ways
  - Consequences : Results and trade-offs of applying the pattern

Design Pattern Example: Multiple Displays



Observer Pattern

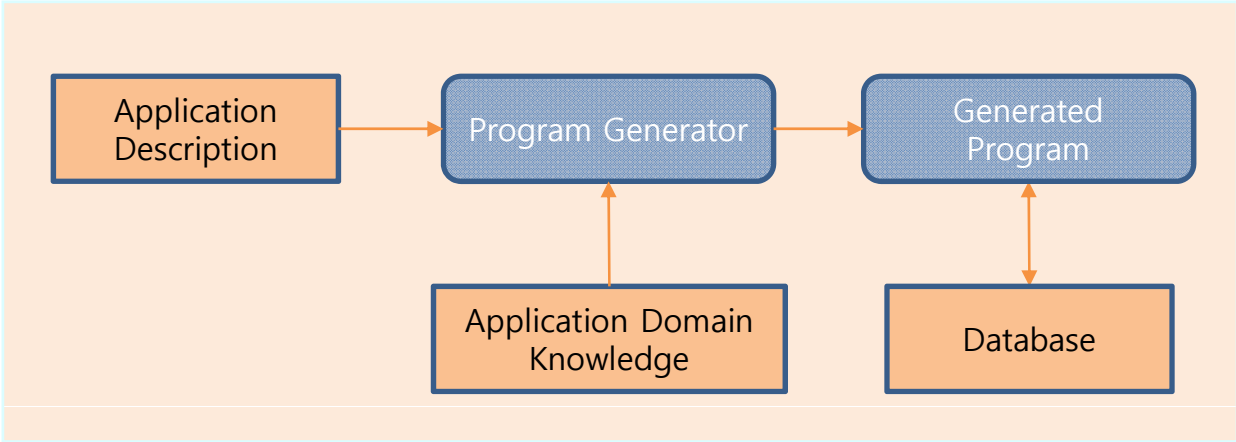
- Name : Observer
- Description : Separates the display of object state from the object itself
- Problem description : Used when multiple displays of state are needed
- Solution description : See slide with UML description
- Consequences : Optimizations to enhance display performance are impractical



Generator-Based Reuse

- Program generators involve the reuse of standard patterns and algorithms.
  - Embedded in the generator and parameterised by user commands. A program is then automatically generated.
  - Possible when domain abstractions and their mapping to executable code can be identified.

- A domain specific language is used to compose and control these abstractions.



Types of Program Generator

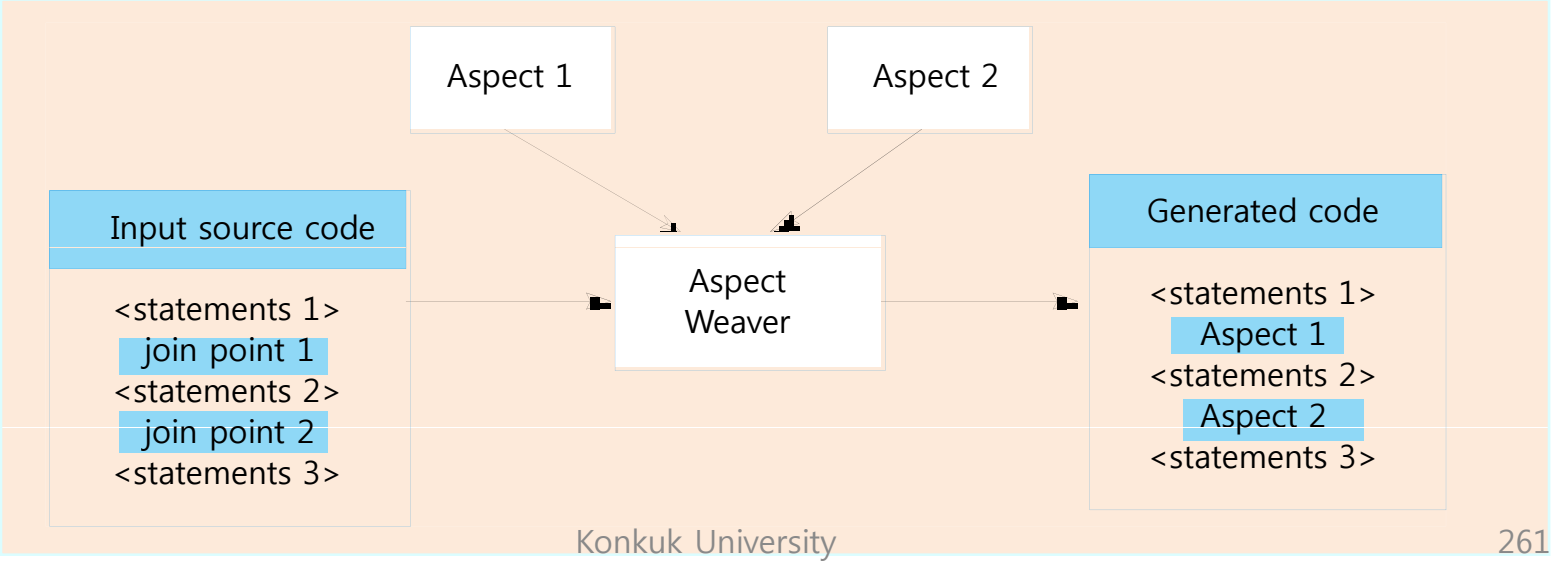
- Types of program generator
  - Application generators : for business data processing
  - Parser and lexical analyzer generators : for language processing
  - Code generators : in CASE tools

- Generator-based reuse is very cost-effective, but its applicability is limited to a relatively small number of application domains.

- It is easier for end-users to develop programs using generators compared to other component-based approaches to reuse.

Reuse: Aspect-Oriented Development

- Aspect-oriented development addresses a major software engineering problem - the separation of concerns.
  - Concerns are often not simply associated with application functionality but are cross-cutting, e.g. all components may monitor their own operation, all components may have to maintain security, etc.
  - Cross-cutting concerns are implemented as aspects and are dynamically woven into a program. The concern code is reused and the new system is generated by the aspect weaver.



Reuse: Application Frameworks

- Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them.
  - The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework.
- Frameworks are moderately large entities that can be reused.

- Framework classes
  - System infrastructure framework
    - Support the development of system infrastructures such as communications, user interfaces and compilers
  - Middleware integration framework
    - Standards and classes that support component communication and information exchange
  - Enterprise application framework
    - Support the development of specific types of application such as telecommunications or financial systems

Reuse: Application System Reuse

- Involves the reuse of entire application systems
  - by configuring a system for an environment
  - by integrating two or more systems to create a new application

- Two approaches
  - COTS product integration
  - Product line development

COTS Product Reuse

- COTS : Commercial Off-The-Shelf
- COTS systems are usually complete application systems offering APIs.
  - Build large systems by integrating COTS systems
  - Effective development strategy for some types of system such as E-commerce systems

- Key benefits
  - Faster application development
  - Usually lower development costs

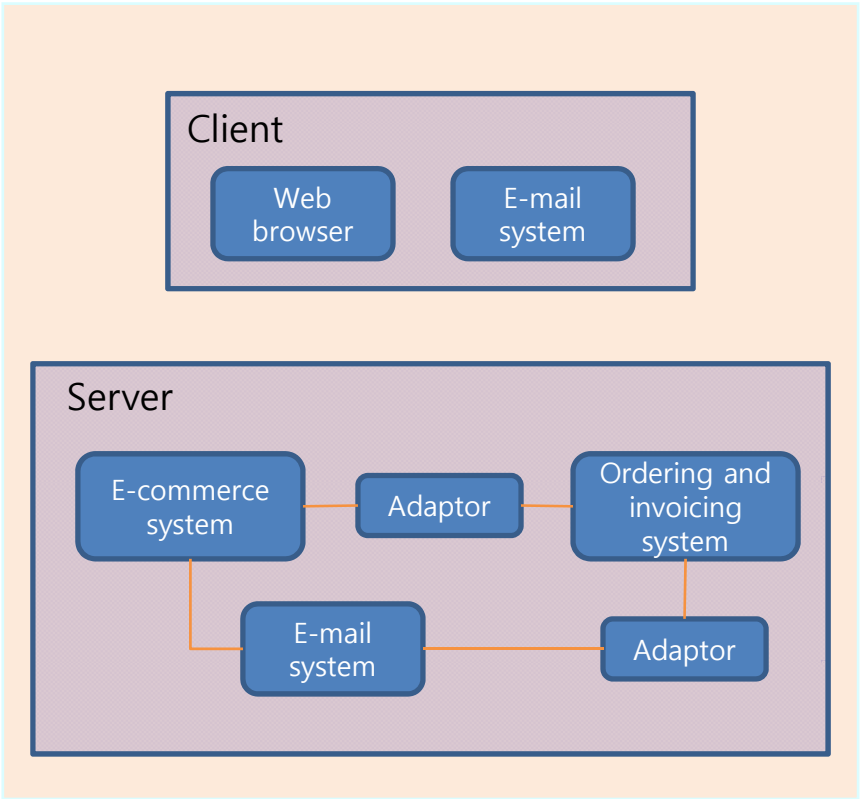
COTS Design Choices

- Which COTS products offer the most appropriate functionality?
  - There may be several similar products that may be used.
- How will data be exchanged?
  - Individual products use their own data structures and formats.
- What features of the product will actually be used?
  - Most products have more functionality than is needed.
  - You should try to deny access to unused functionality.

COTS System Integration Problems

- Lack of control over functionality and performance
  - COTS systems may be less effective than they appear.
- Problems with inter-operability
  - Different COTS systems may make different assumptions that means integration is difficult.
- No control over system evolution
  - COTS vendors do not control system evolution.
- Support from COTS vendors
  - COTS vendors may not offer support over the lifetime of the product.

Example: E-Procurement System



- On the client, standard e-mail and web browsing programs are used.
- On the server, an e-commerce platform has to be integrated with an existing ordering system.
  - Involves writing an adaptor so that they can exchange data.
  - An e-mail system is also integrated to generate e-mail for clients. This also requires an adaptor to receive data from the ordering and invoicing system.

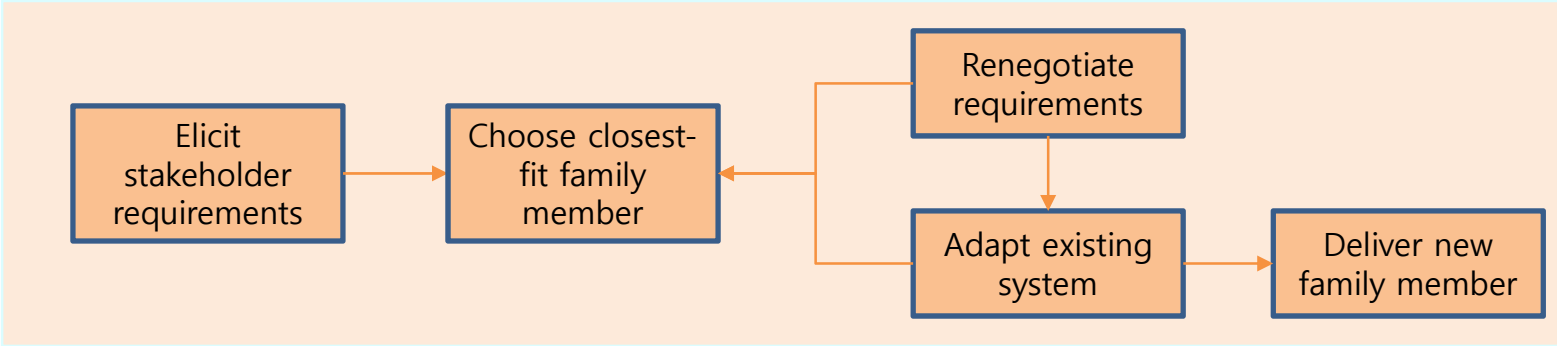
Software Product Line

- Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context.

- Adaptation may involve
  - Component and system configuration
  - Adding new components to the system
  - Selecting from a library of existing components
  - Modifying components to meet new requirements

Product Instance Development

- Elicit stakeholder requirements
  - Use existing family member as a prototype
- Choose closest-fit family member
  - Find the family member that best meets the requirements
- Re-negotiate requirements
  - Adapt requirements as necessary to capabilities of the software
- Adapt existing system
  - Develop new modules and make changes for family member
- Deliver new family member
  - Document key features for further member development



Summary

- Advantages of reuse are lower costs, faster software development and lower risks.
- Design patterns are high-level abstractions that document successful design solutions.
- Program generators are also concerned with software reuse - the reusable concepts are embedded in a generator system.
- Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialisation.
- COTS product reuse is concerned with the reuse of large, off-the-shelf systems.
- Problems with COTS reuse include lack of control over functionality, performance, and evolution and problems with inter-operation.
- Software product lines are related applications developed around a common core of shared functionality.

Chapter 19.

Component-Based Software Engineering

Objectives

- To explain that CBSE is concerned with developing standardized components and composing them into applications
- To describe components and component models
- To show principal activities in CBSE process
- To discuss approaches to component composition and problems that may arise

Component-Based Development

- Component-based software engineering (CBSE) is an approach to software development that relies on software reuse.
  - Emerged from the failure of object-oriented development to support effective reuse
  - Single object classes are too detailed and specific to reuse.

- Components are more abstract than object classes and can be considered to be stand-alone service providers.

CBSE Essentials

- CBSE essentials
  - Independent components specified by their interfaces
  - Component standards to facilitate component integration
  - Middleware that provides support for component inter-operability
  - Development process that is geared to reuse

- Apart from the benefits of reuse, CBSE is based on sound software engineering design principles
  - Components are independent so do not interfere with each other.
  - Component implementations are hidden.
  - Communication is through well-defined interfaces.
  - Component platforms are shared and reduce development costs.

CBSE Problems

- Component trustworthiness
  - How can a component with no available source code be trusted?
- Component certification
  - Who will certify quality of the components?
- Emergent property prediction
  - How can the emergent properties of component compositions be predicted?
- Requirements trade-offs
  - How do we do trade-off analysis between the features of one component and another?

Components

- Components provide a service without regard to where the component is executing or what its programming language is.
  - A component is an independent executable entity that can be made up of one or more executable objects.
  - The component interface is published and all interactions are through the published interface.

Councill and Heinmann:

A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

Szyperski:

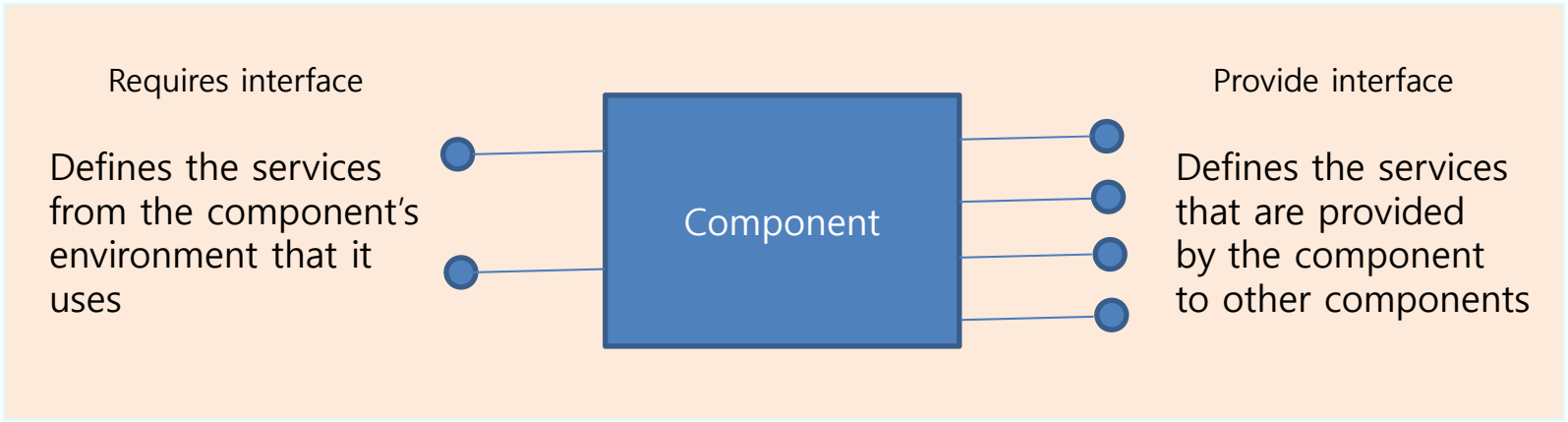
A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.

Characteristics of Components

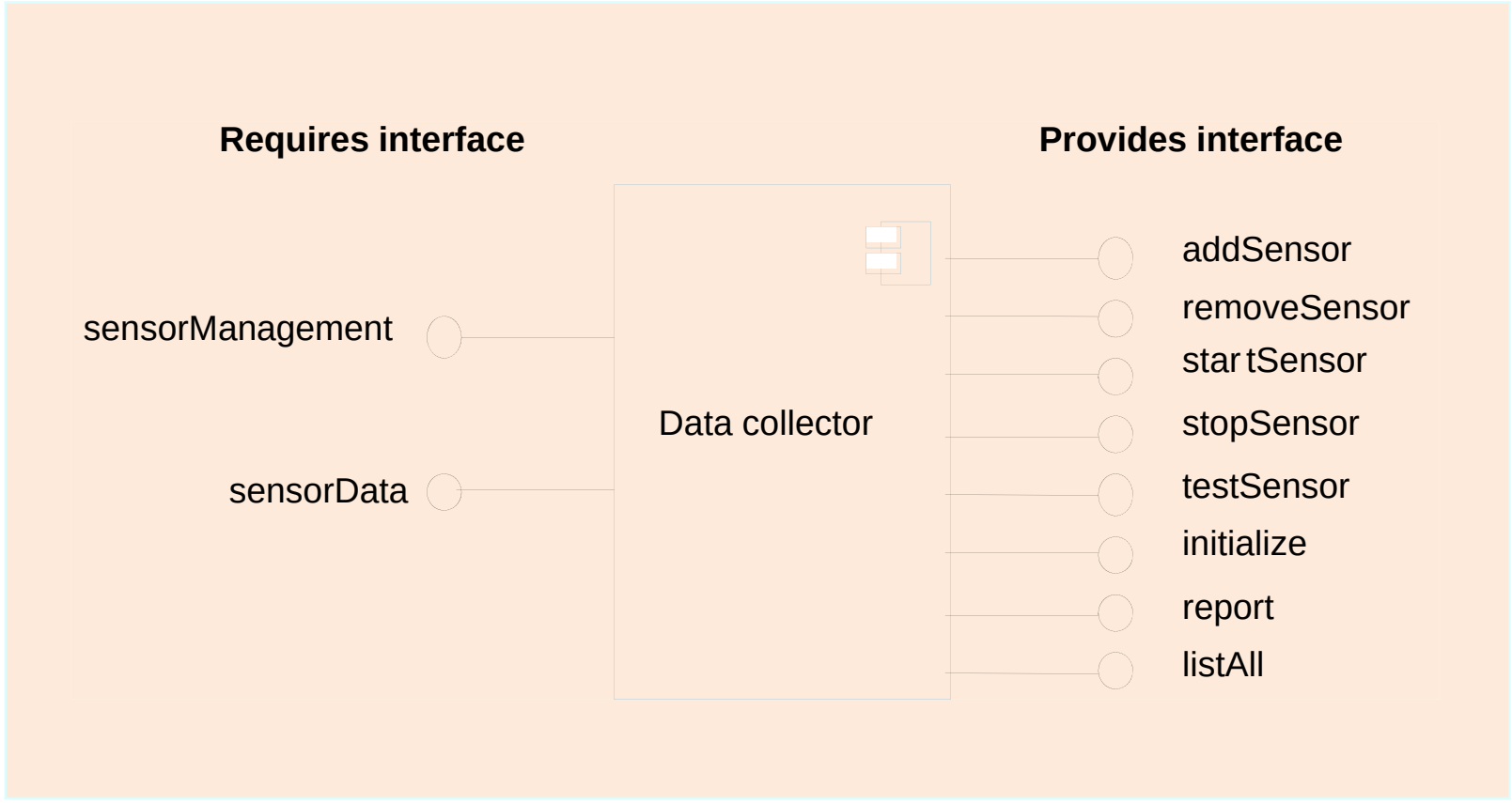
Characteristics	Description
Standardized	Component standardisation means that a component that is used in a CBSE process has to conform to some standardised component model. This model may define component interfaces, component meta-data, documentation, composition and deployment.
Independent	A component should be independent – it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification.
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity on some component platform that implements the component model. This usually means that the component is a binary component that does not have to be compiled before it is deployed.
Documented	Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified.

Component Interface

- Provides interface
  - Defines the services that are provided by the component to other components
- Requires interface
  - Defines the services that specifies what services must be made available for the component to execute as specified.

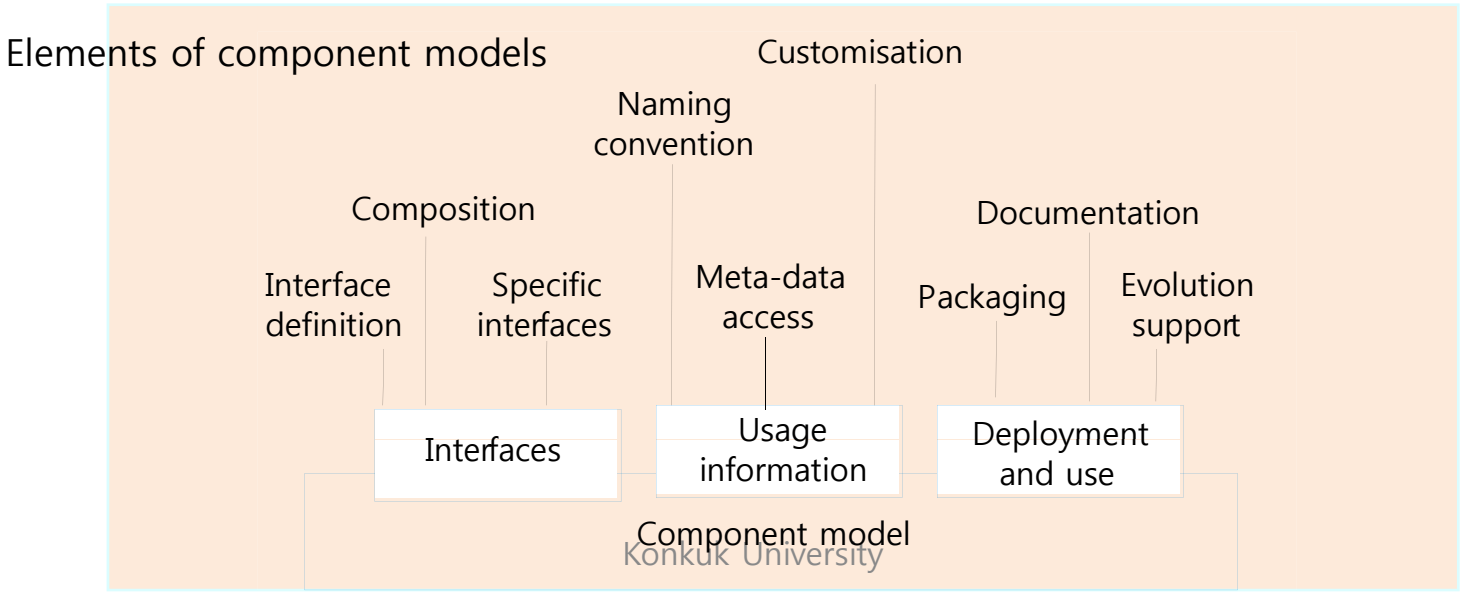


Example: A Data Collector Component Interface



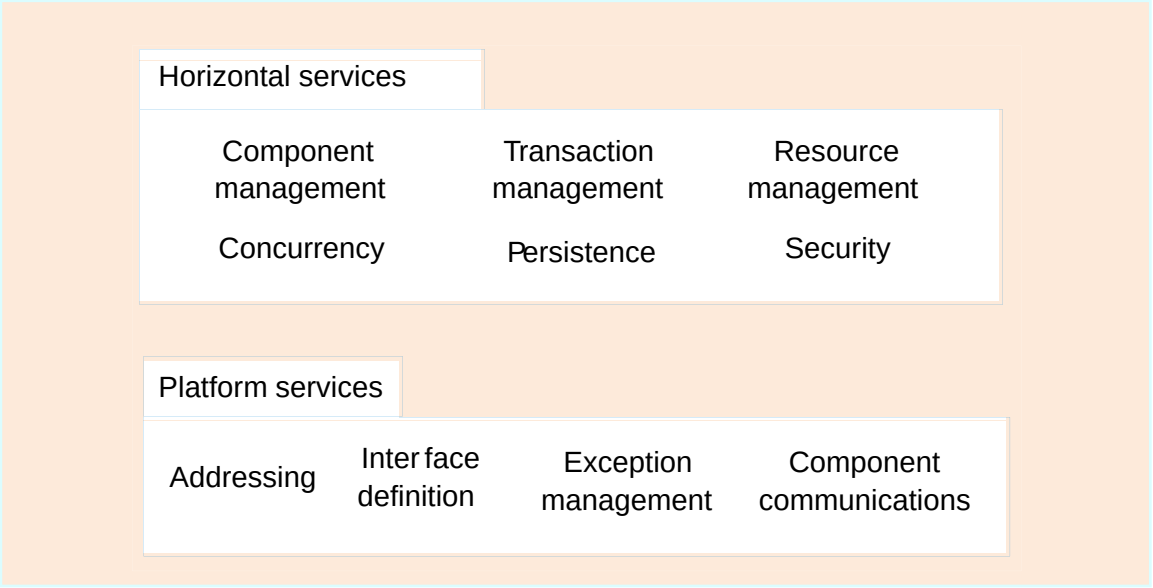
Component Model

- Component model is a definition of standards for component implementation, documentation and deployment.
  - EJB model (Enterprise Java Beans)
  - COM+ model (.NET model)
  - CORBA component Model
- Component model specifies how interfaces should be defined and the elements that should be included in the interface definition.



Middleware Support

- Middleware provides support for executing components.
  - Component models are the basis of middleware.
- Component model implementations provide
  - Platform services : allow components written according to the model to communicate
  - Horizontal services : application-independent services used by different components
- Container
  - A set of interfaces used to access the service implementations
  - To use services provided by a model



Component Development for Reuse

- Components developed for a specific application usually have to be generalized to make them reusable.
- A component is most likely to be reusable if it associated with a stable domain abstraction (business object).
  - In a hospital, stable domain abstractions are associated with the fundamental purpose - nurses, patients, treatments, etc.

- Component reusability
  - Should reflect stable domain abstractions
  - Should hide state representation
  - Should be as independent as possible
  - Should publish exceptions through the component interface

- Trade-off between reusability and usability
  - The more general the interface, the greater the reusability.
  - But it is then more complex and hence less usable.

Cost of Reusable Component

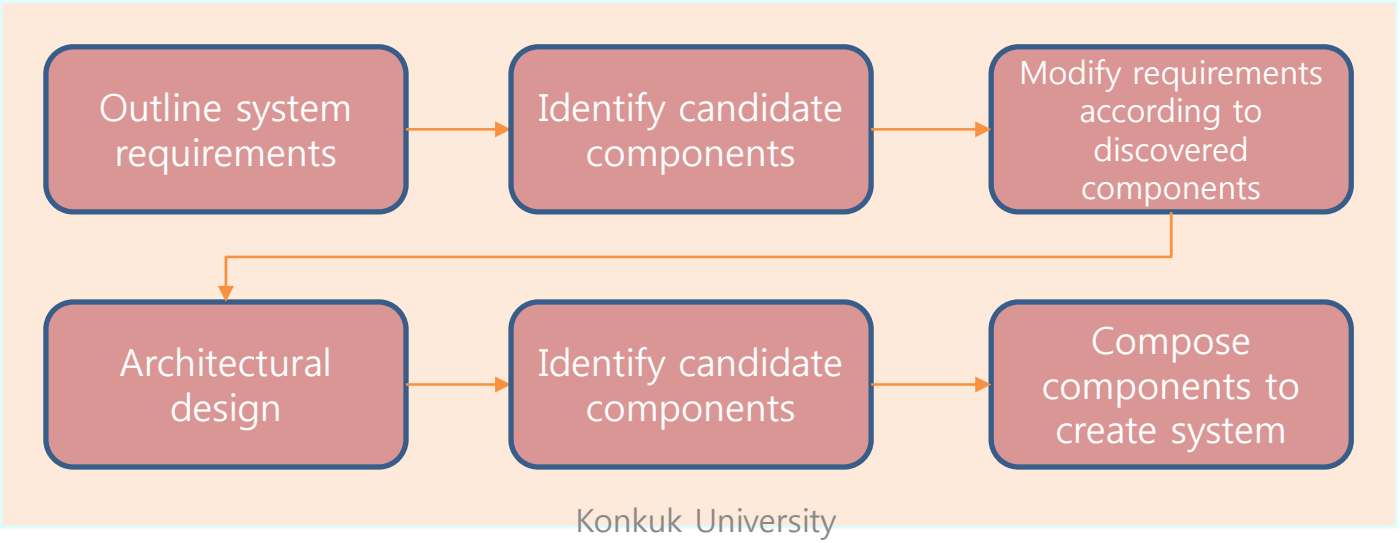
- The development cost of reusable components may be higher than the cost of specific equivalents.
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents.
- This extra reusability enhancement cost should be an organization cost rather than a project cost.

Legacy System Components

- Existing legacy systems that fulfill a useful business function can be re-packaged as components for reuse.
  - Involve writing a wrapper component that implements provides and requires interfaces to access the legacy system
  - Although costly, this can be much less expensive than rewriting the legacy system.

CBSE Process

- When reusing components, it is essential to make trade-offs between ideal requirements and the services actually provided by available components.
  - Developing outline requirements
  - Searching for components then modifying requirements according to available functionality
  - Searching again to find if there are better components that meet the revised requirements



Component Identification Issues

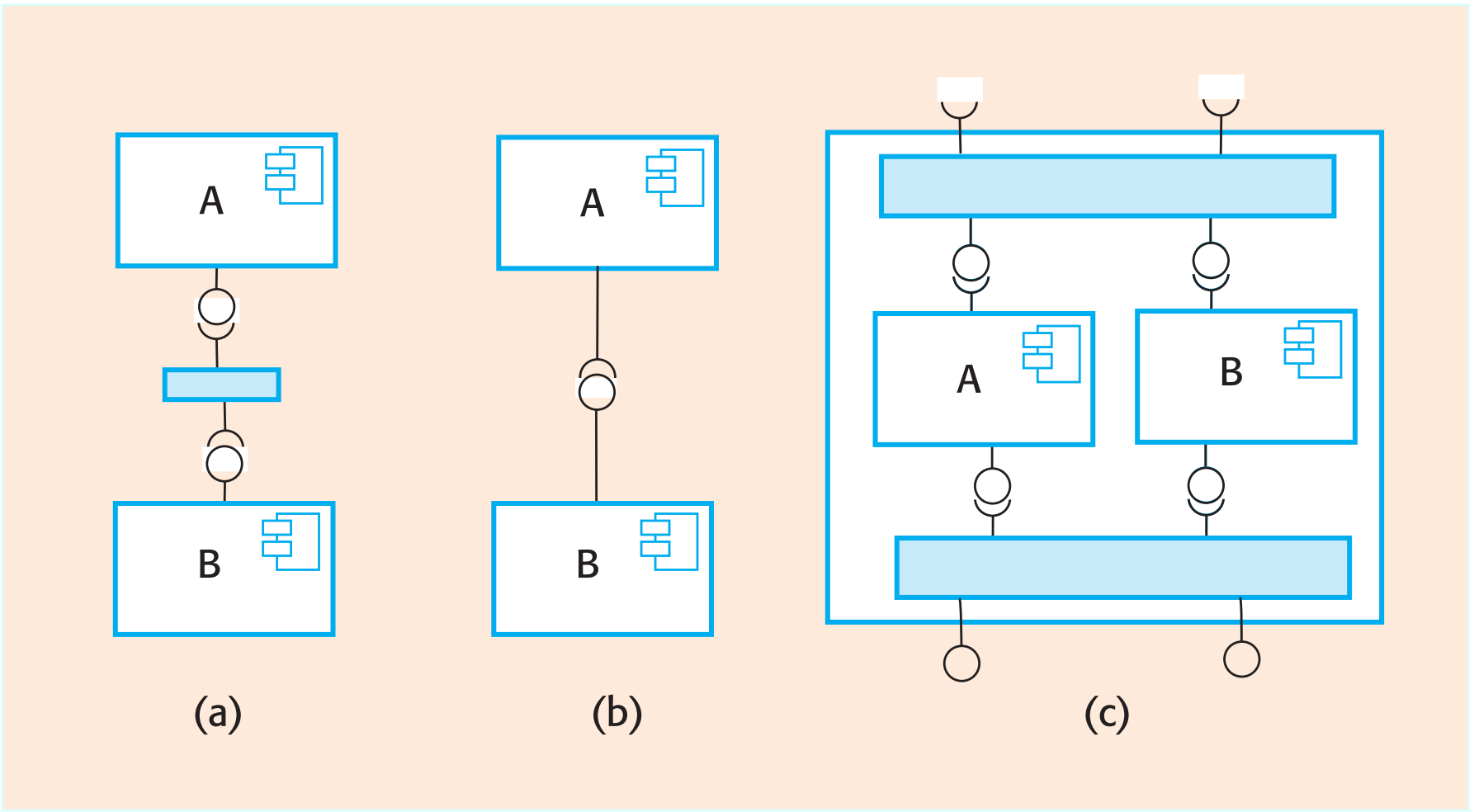
- Trust
  - You need to be able to trust the supplier of a component.
    - At best, an un-trusted component may not operate as advertised.
    - At worst, it can breach your security.
- Requirements
  - Different groups of components will satisfy different requirements.
- Validation
  - Component specification may not be detailed enough to allow comprehensive tests to be developed.
  - Components may have unwanted functionality. How can you test this will not interfere with your application?

Component Composition

- Process of assembling components to create a system
  - Involve integrating components with each other and with the component infrastructure
  - Normally have to write 'glue code' to integrate components

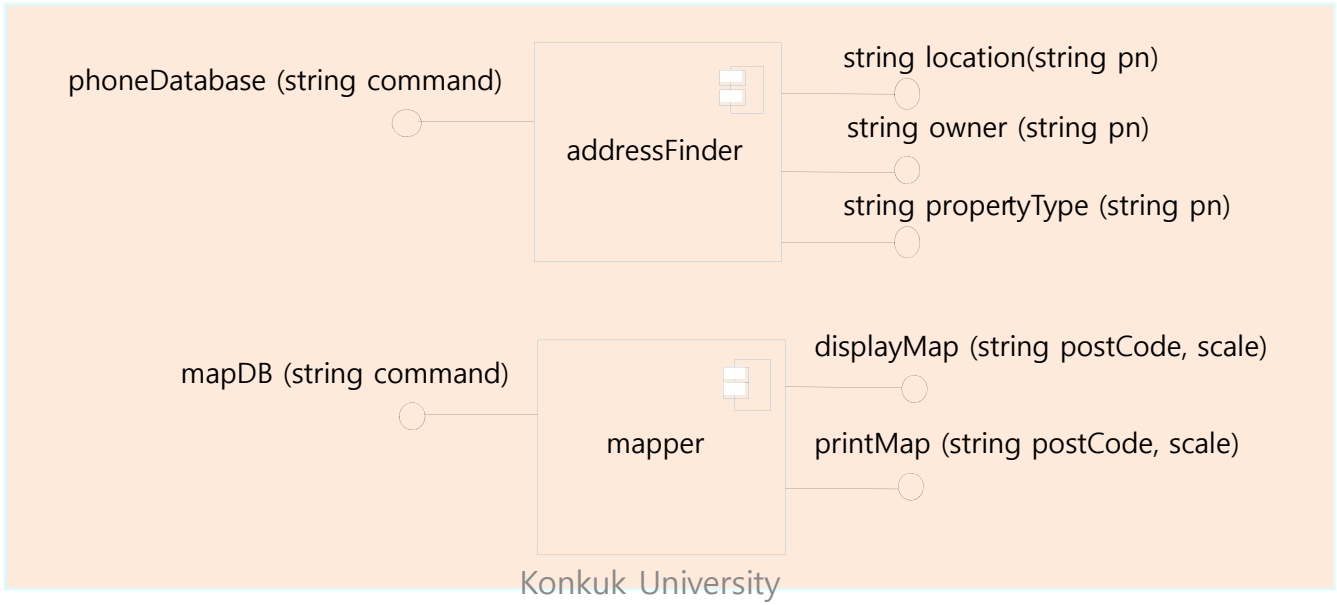
- Types of composition
  - Sequential composition
    - Where the composed components are executed in sequence
    - Involves composing the provides interfaces of each component
  - Hierarchical composition
    - Where one component calls on the services of another.
    - The provides interface of one component is composed with the requires interface of another
  - Additive composition
    - The interfaces of two components are put together to create a new component

Types of Composition



Interface Incompatibility

- Parameter incompatibility
  - Operations have the same name but are of different types.
- Operation incompatibility
  - Names of operations in the composed interfaces are different.
- Operation incompleteness
  - Provides interface of one component is a subset of the requires interface of another.



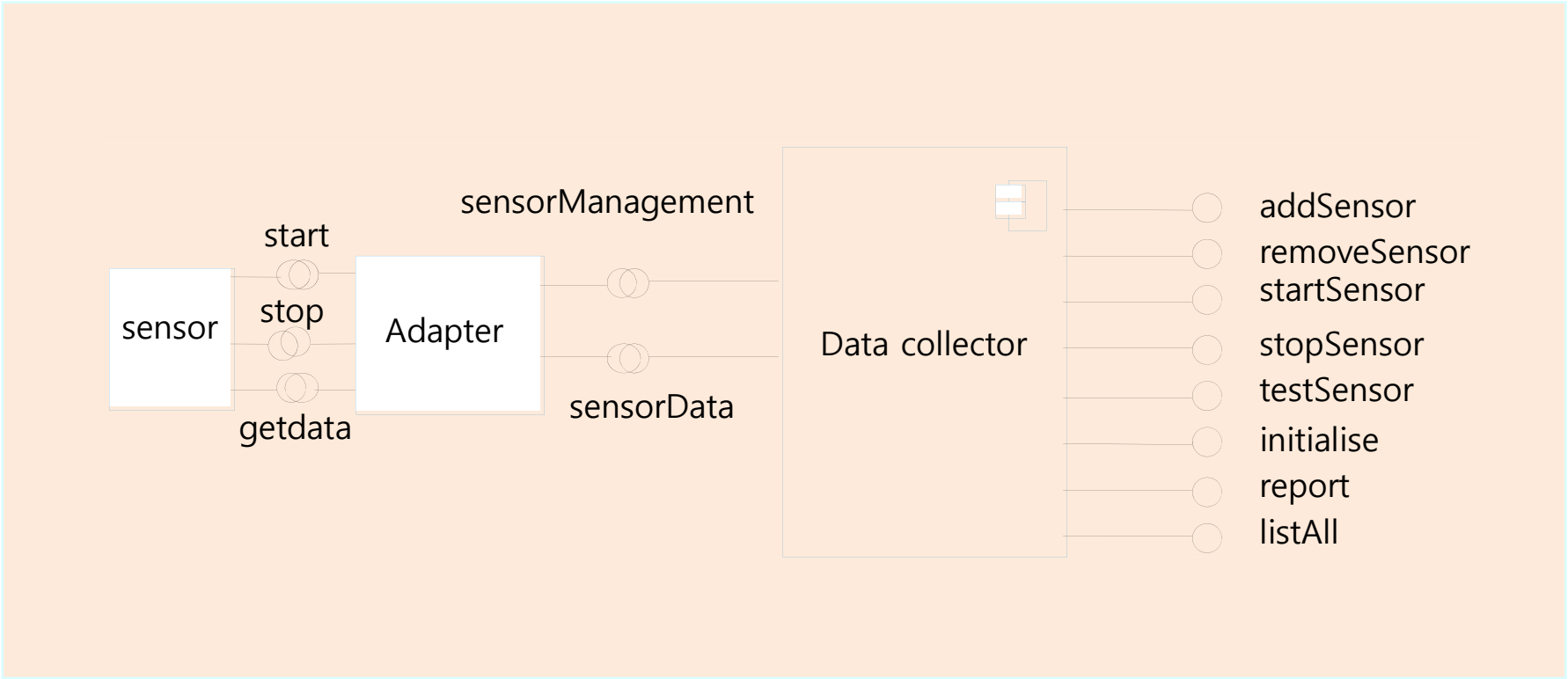
Adaptor Component

- Address the problem of component incompatibility by reconciling the interfaces of the components that are composed.
  - Different types of adaptor are required depending on the type of composition.

- An *addressFinder* and a *mapper* component may be composed through an adaptor that strips the postal code from an address and passes this to the mapper component.

```
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000)
```

Adaptor for Data Collector Component



Interface Semantics

- Have to rely on component documentation to decide if syntactically compatible interfaces are actually compatible

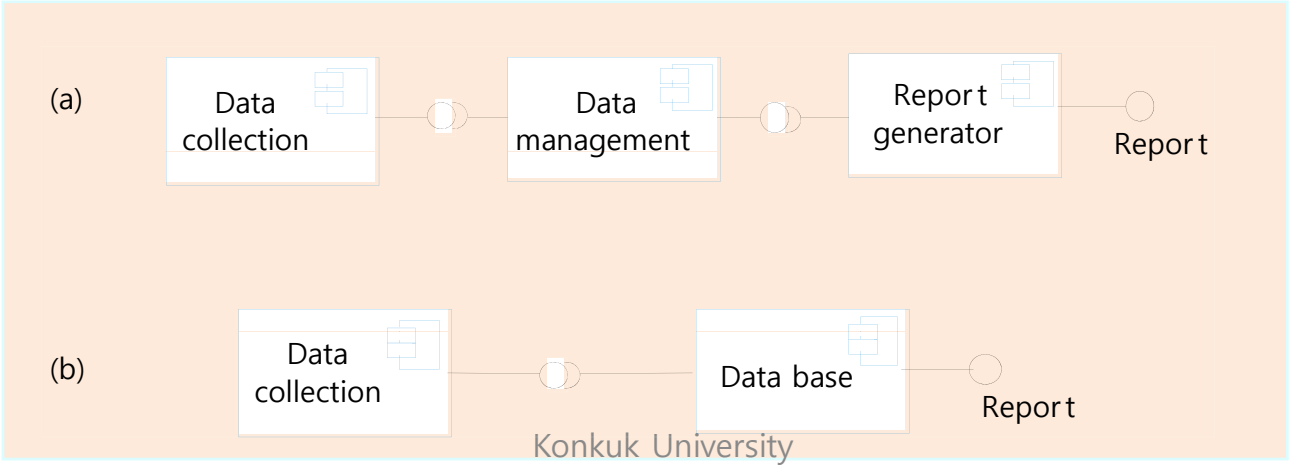
- Object Constraint Language (OCL)
  - Define constraints that are associated with UML models.
  - Based around the notion of pre and post condition specification - similar to the approach used in Z.

```
-- The context keyword names the component to which the conditions apply
context addItem

-- The preconditions specify what must be true before execution of addItem
pre:    PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

-- The postconditions specify what is true after execution
post:   libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

context delete

pre: PhotoLibrary.retrieve(pid) <> null ;

post: PhotoLibrary.retrieve(pid) = null
      PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
      PhotoLibrary.libSize() = libSize()@pre - 1
```

Trade-Offs in Composition

- When composing components, you may find
  - Conflicts between functional and non-functional requirements
  - Conflicts between the need for rapid delivery and system evolution

- You need to make decisions such as
  - What composition of components is effective for delivering the functional requirements?
  - What composition of components allows for future change?
  - What will be the emergent properties of the composed system?



Summary

- CBSE is a reuse-based approach to defining and implementing loosely coupled components into systems.
- A component is a software unit whose functionality and dependencies are completely defined by its interfaces.
- A component model defines a set of standards that component providers and composers should follow.
- Component composition is the process of 'wiring' components together to create a system.
- When composing reusable components, you normally have to write adaptors to reconcile different component interfaces.
- When choosing compositions, you have to consider required functionality, non-functional requirements and system evolution.

Part V. Verification & Validation

Chapter 22.

Verification and Validation

Objectives

- To introduce software verification and validation
- To discuss distinction between software verification and validation
- To describe program inspection process and its role in V & V
- To explain static analysis as verification technique
- To describe the Cleanroom software development process

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 Process

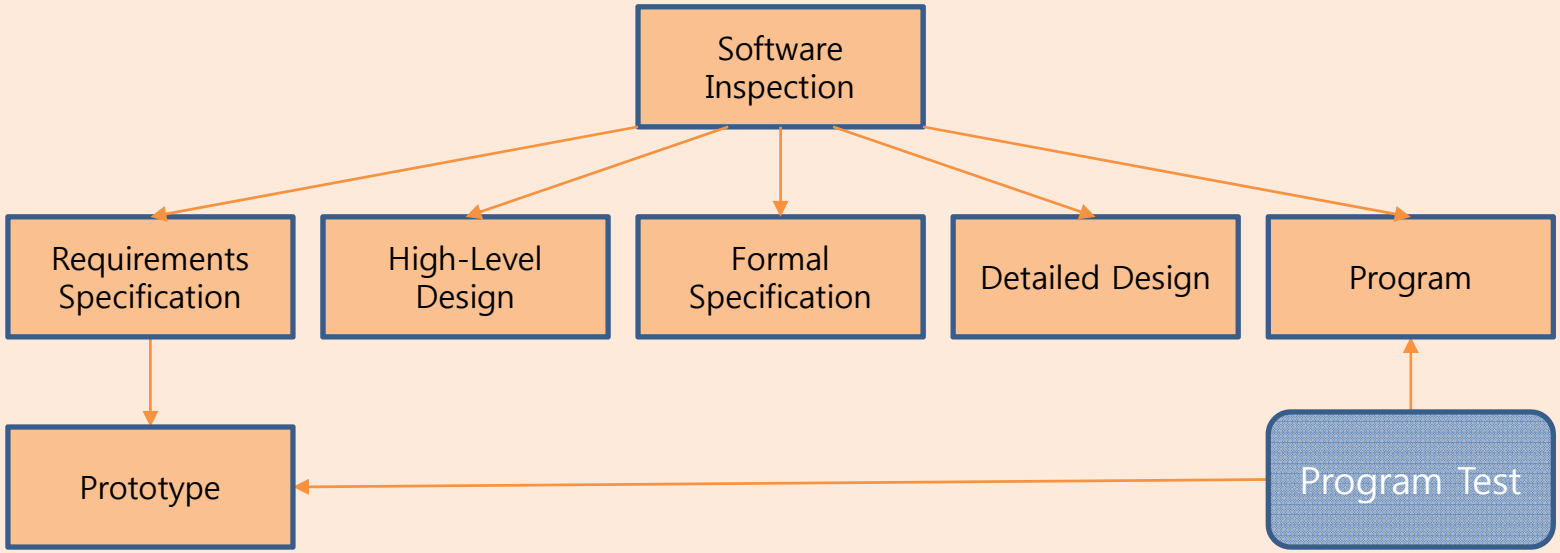
- V&V is a whole life-cycle process
  - Must be applied at each stage in the software process.
- Two principal objectives
  - Discovery of defects in a system
  - Assessment of whether or not the system is useful and useable in an operational situation
- Goals of V&V
  - V&V should establish confidence that the software is suitable for purpose.
  - Does not mean completely free of defects
  - Rather, it must be good enough for its intended use.
  - The type of use will determine the degree of confidence that is needed.

V & V Confidence

- V&V confidence depends on the system's purpose, user expectations, and marketing environment.
  - Software function
    - The level of confidence depends on how critical the software is to an organization.
  - 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.

Static and Dynamic Verification

- Software Inspection
  - Analyze static system representation to discover problems (Static Verification)
  - May be supplemented by tool-based document and code analysis
- Software Testing
  - Exercising and observing product behaviour (Dynamic Verification)
  - System is executed with test data and its operational behaviour is observed.



Program Testing

- Can reveal the presence of errors, NOT their absence
  - Can validate non-functional requirements as we can execute the software and see how it behaves
  - Should be used in conjunction with static verification to provide full V&V coverage

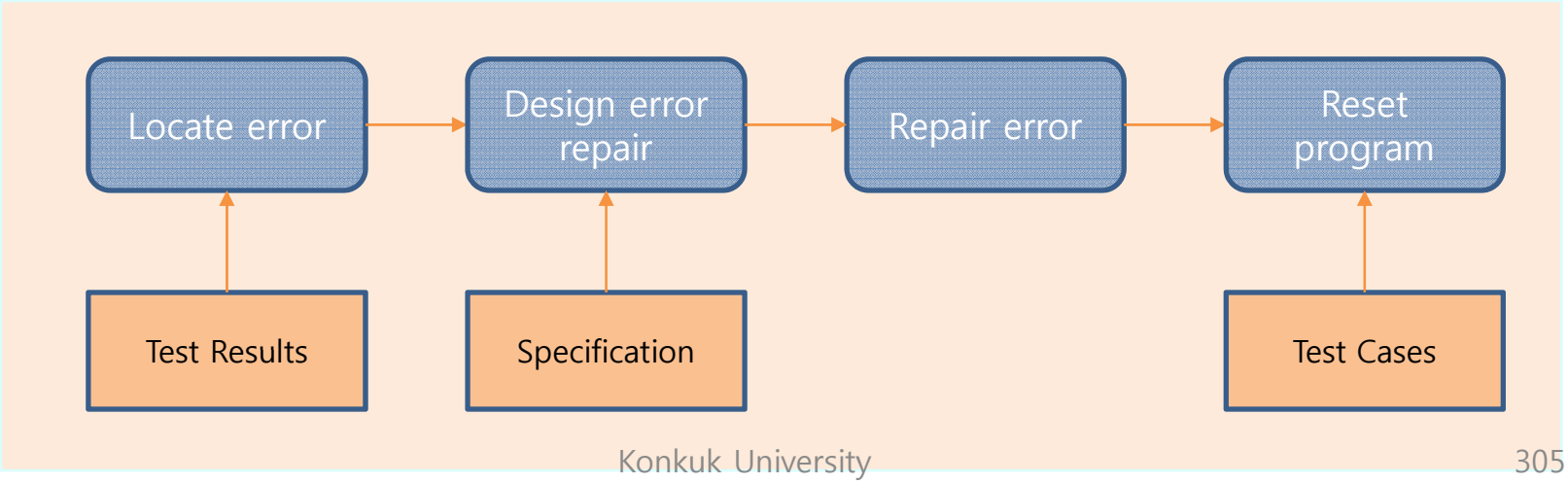
- Types of testing
  - Defect testing
    - Tests designed to discover system defects
    - A successful defect test is one which reveals the presence of defects in a system.
    - Covered in Chapter 23
  - Validation testing
    - Intended to show that the software meets its requirements.
    - A successful test is one that shows that a requirements has been properly implemented.

Testing and Debugging

- Defect testing and debugging are different.
  - Testing is concerned with establishing the existence of defects in a program.
  - Debugging is concerned with locating and repairing these errors.

- Debugging involves formulating a hypothesis about program behaviour and testing these hypotheses to find the system error.

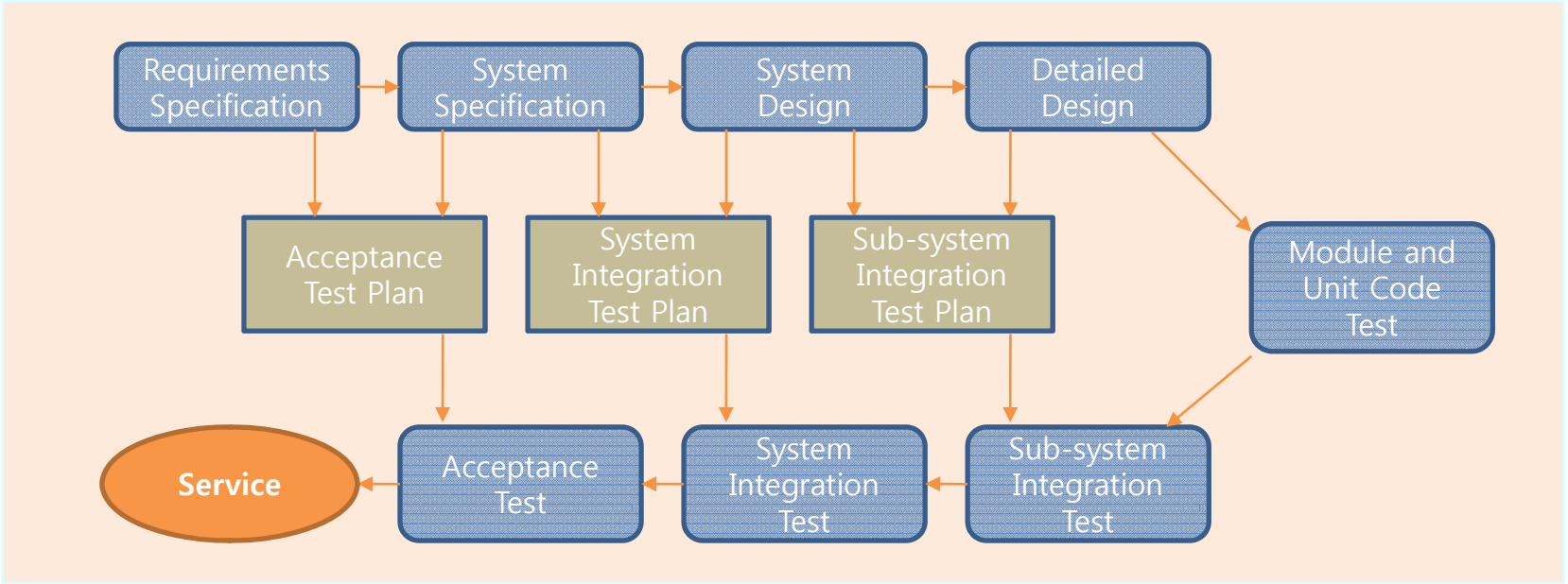
- Debugging process:



V & V Planning

- V&V Planning should start early in the development process.
  - The plan should identify the balance between static verification and testing.
  - Test planning is about defining standards for testing process rather than describing product tests.

- V-Model for Software Testing



Software Test Plan

Items to Consider	Description
Testing process	A description of the major phases of the testing process. These might be as described earlier in this chapter.
Requirements traceability	Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.
Tested items	The products of the software process that are to be tested should be specified.
Testing schedule	An overall testing schedule and resource allocation for this schedule. This, obviously, is linked to the more general project development schedule.
Test recording procedure	It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it been carried out correctly.
Hardware and software requirements	section should set out software tools required and estimated hardware utilisation.
Constraints	Constraints affecting the testing process such as staff shortages should be anticipated in this section.

Software Inspection

- Software inspection involves people examining the source representation with aim of discovering anomalies and defects.
  - Does not require execution of system
  - May be used before implementation
  - May be applied to any representation of the system (requirements, design, configuration data, test data, etc.)
  - Effective technique for discovering program errors

- Advantages:
  - Many different defects may be discovered in a single inspection.
  - In testing, one defect may mask another so several executions are required.
  - Using domain and programming knowledge, reviewers are likely to have seen the types of error that commonly arise.

Inspection and Testing

- Inspection and testing are complementary and not opposing verification techniques.
  - Both should be used during the V & V process.

- Inspections
  - Can check conformance with a specification but not conformance with the customer's real requirements
  - Cannot check non-functional characteristics such as performance, usability, etc.

Program Inspection

- Formalized approach to document reviews
  - Intended explicitly for detecting defects (not correction)

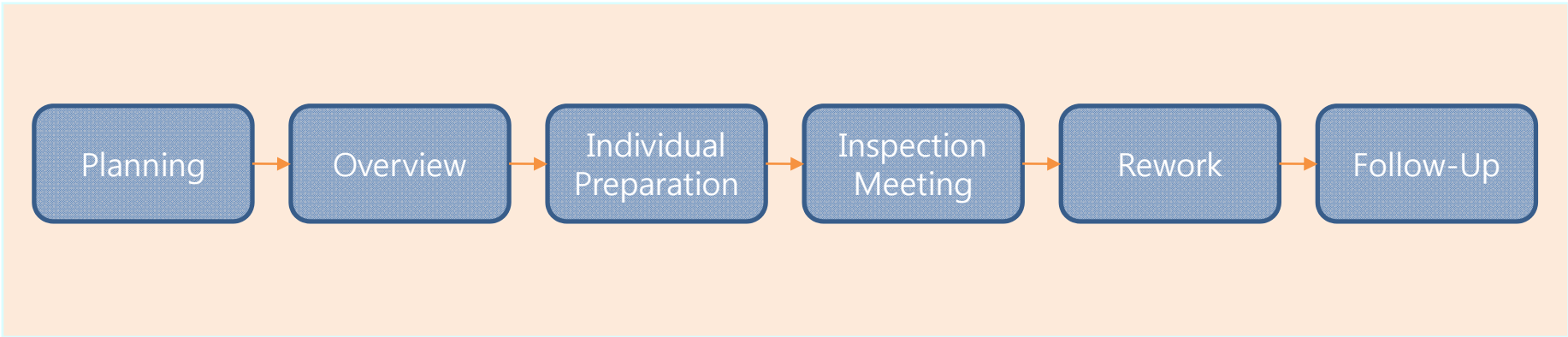
- Defects may be
  - Logical errors
  - Anomalies in the code that might indicate an erroneous condition (e.g. an uninitialized variable)
  - Non-compliance with standards

Pre-Conditions for Inspection

- A precise specification must be available.
- Team members must be familiar with the organization standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.
- Management must accept that inspection will increase costs early in the software process.
- Management should not use inspections for staff appraisal, i.e. finding out who makes mistakes.

Inspection Procedure

- Inspection procedure
  - Present system overview to inspection team
  - Code and associated documents are distributed to inspection team in advance
  - Inspection takes place and discovered errors are noted
  - Modifications are made to repair discovered errors
  - Re-inspection may or may not be required



Inspection Roles

Roles	Description
Author or Owner	The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.
Inspector	Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.
Reader	Presents the code or document at an inspection meeting
Scribe	Records the results of the inspection meeting
Chairman or Moderator	Manages the process and facilitates the inspection. Reports process results to the Chief moderator.
Chief Moderator	Responsible for inspection process improvements, checklist updating, standards development, etc.

Inspection Checklist

- Checklist of common errors should be used to drive inspections.
  - Depend on the programming languages
  - Reflect the characteristic errors that are likely to arise in the language

- In general, the weaker type checking language, the larger the checklist.

- Examples of common errors in checklists
  - Data faults
  - Control faults
  - Input/Output faults
  - Interface faults
  - Storage management faults
  - Exception management faults

Automated Static Analysis

- Static analyzers are software tools for source text processing.
  - Parse the program text and try to discover potentially erroneous conditions
  - Very effective as an aid to inspections
  - A supplement to inspections but not a replacement

Fault Class	Static Analysis Check
Data fault	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

Stages of Static Analysis

- All stages generate vast amounts of information, and must be used with care.

Stage	Description
Control Flow Analysis	Checks for loops with multiple exit or entry points, finds unreachable code, etc.
Data Use Analysis	Detects uninitialized variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.
Interface Analysis	Checks the consistency of routine and procedure declarations and their use
Information Flow Analysis	Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
Path Analysis	Identifies paths through the program and sets out the statements executed in that path. It is potentially useful in the review process.

Use of Static Analysis

- Particularly valuable when a language such as C is used.
  - C has weak typing and many errors are undetected by the C compiler.

- Less cost-effective for languages like Java
  - Java has strong type checking and can therefore detect many errors during compilation.

Verification through Formal Methods

- Formal methods can be used when a mathematical specification of system is prepared.
  - Ultimate static verification technique : formal verification
  - Involve detailed mathematical analysis of the specification
  - Develop formal arguments that a program conforms to its mathematical specification

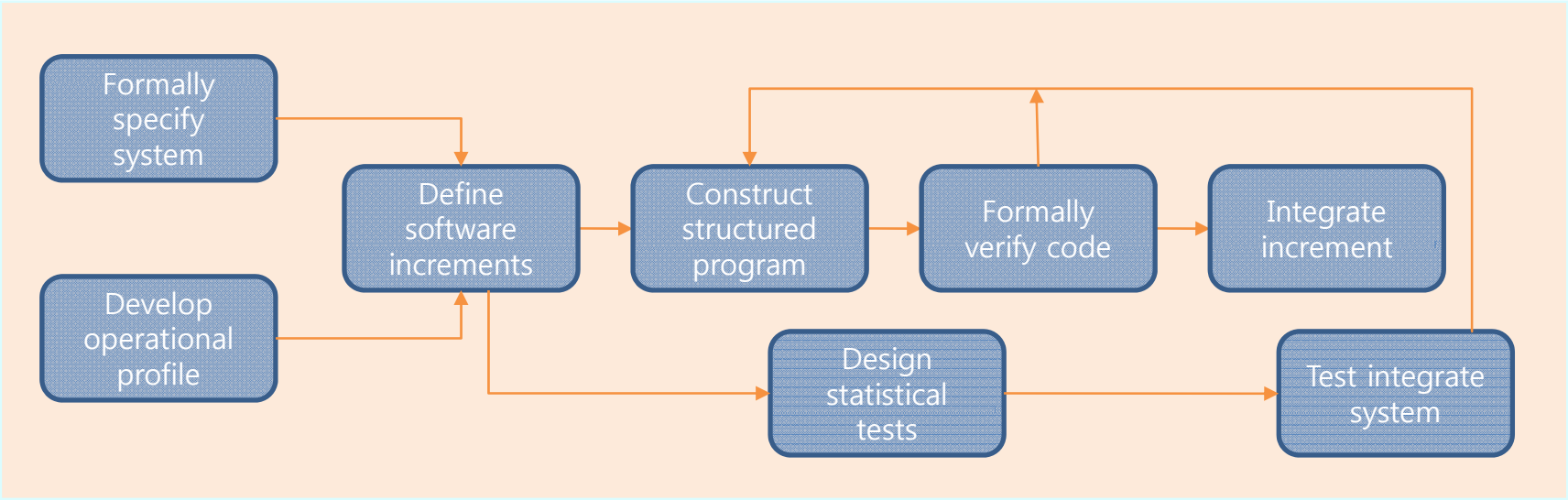
Arguments about Formal Methods

- Advantages:
  - Produce mathematical specifications which require detailed analysis of the requirements and this is likely to uncover errors
  - Detect implementation errors before testing, when the program is analyzed alongside the specification

- Disadvantages:
  - Require specialized notations that cannot be understood by domain experts
  - Very expensive to develop specification and even more expensive to show that the program meets that specification
  - May be possible to reach the same level of confidence more cheaply using other V & V techniques

Cleanroom Software Development

- Cleanroom process
  - Defect avoidance rather than defect removal
  - Based on
    - Incremental development
    - Formal specification
    - Static verification using correctness arguments
    - Statistical testing to determine program reliability



Characteristics of Cleanroom Process

- Cleanroom
  - Formal specification using a state transition model
  - Incremental development where the customer prioritises increments
  - Structured programming : Limited control and abstraction constructs are used in the program.
  - Static verification using rigorous inspections
  - Statistical testing of the system

- Team organization:
  - Specification team: Responsible for developing and maintaining the system specification.
  - Development team: Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process.
  - Certification team: Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models are used to determine when reliability is acceptable.

Evaluation of Cleamroom Process

- The results of using the Cleanroom process have been very impressive with few discovered faults in delivered systems.
  - Independent assessment shows that the process is no more expensive than other approaches.
  - There were fewer errors than in a 'traditional' development process.

- However, the process is not widely used.
  - It is not clear how this approach can be transferred to an environment with less skilled or less motivated software engineers.

Summary

- Verification and validation are not the same thing. Verification shows conformance with specification; validation shows that the program meets the customer's needs.
- Test plans should be drawn up to guide the testing process.
- Static verification techniques involve examination and analysis of the program for error detection.
- Program inspections are very effective in discovering errors.
- Program code in inspections is systematically checked by a small team to locate software faults.
- Static analysis tools can discover program anomalies which may be an indication of faults in the code.
- Cleanroom development process depends on incremental development, static verification and statistical testing.

Chapter 23.

Software Testing

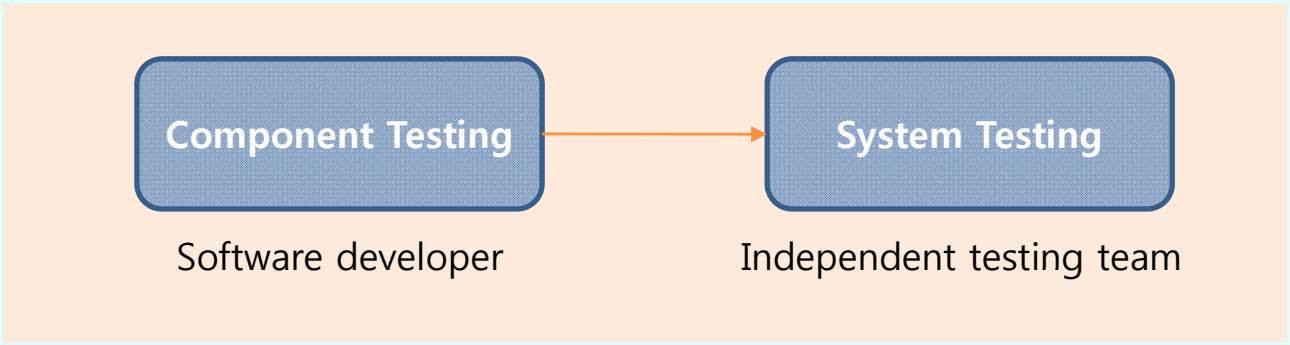
Objectives

- To discuss distinctions between validation testing and defect testing
- To describe principles of system and component testing
- To describe strategies for generating system test cases
- To understand essential characteristics of tools used for test automation

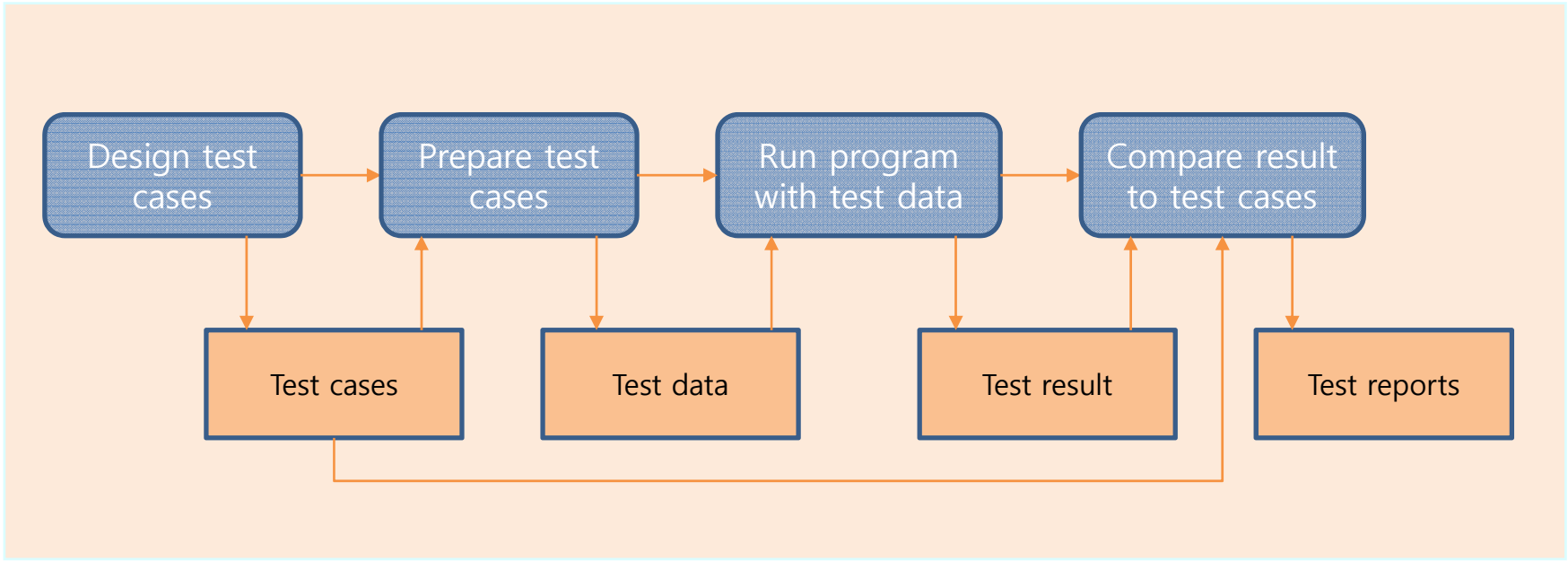
Software Testing

- Component testing
  - Testing of individual program components
  - Usually responsibility of developers
  - Tests are derived from the developer's experience.

- System testing
  - Testing of groups of components integrated to create a system or sub-system
  - Responsibility of independent testing team
  - Tests are based on system specification.



Software Testing Process



Goals of Software Testing

- Validation testing
  - To demonstrate to developer and system customer that the software meets its requirements
  - A successful test shows that the system operates as intended.

- Defect testing
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

System Testing

- System testing involves integrating components to create a system or sub-system

- Two phases:
  - Integration testing
    - Test team has access to system source code.
    - System is tested as components are integrated.
  - Release testing
    - Test team tests a complete system to be delivered as a black-box.

Integration Testing

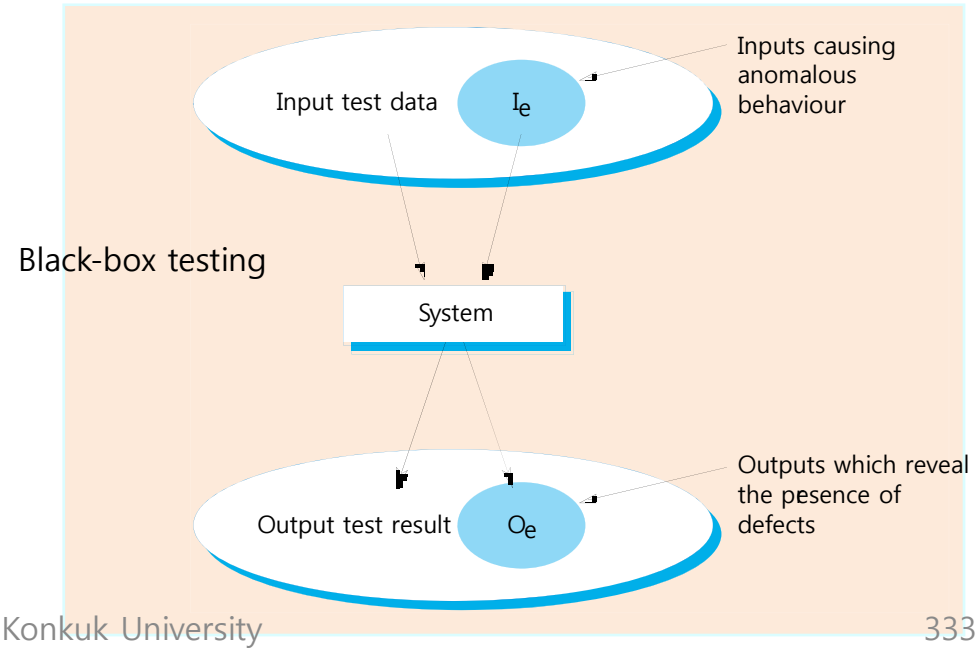
- Involves building a system from its components and testing it for problems that arise from component interactions.
  - Top-down integration
    - Develop the skeleton of the system and populate it with components
  - Bottom-up integration
    - Integrate infrastructure components then add functional components

Integration Testing Approaches

- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture.
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development.
- Test implementation
  - Often easier with bottom-up integration testing
- Test observation
  - Problems with both approaches
  - Extra code may be required to observe tests.

Release Testing

- Process of testing a system release that will be distributed to customers
  - To increase the supplier's confidence that the system meets its requirements
- Release testing is usually black-box or functional testing
  - Based on the system specification only
  - Testers do not have knowledge of the system implementation.
- Release testing may include
  - Performance testing
  - Stress testing



Performance Testing

- Release testing may involve testing emergent properties of system.
  - Performance
  - Reliability

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

Stress Testing

- Exercises the system beyond its maximum design load.
  - Stressing the system often causes defects to come to light.
- Stressing the system to test failure behaviour.
  - Systems should not fail catastrophically.
  - Stress testing checks for unacceptable loss of service or data too.

- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as network becomes overloaded.

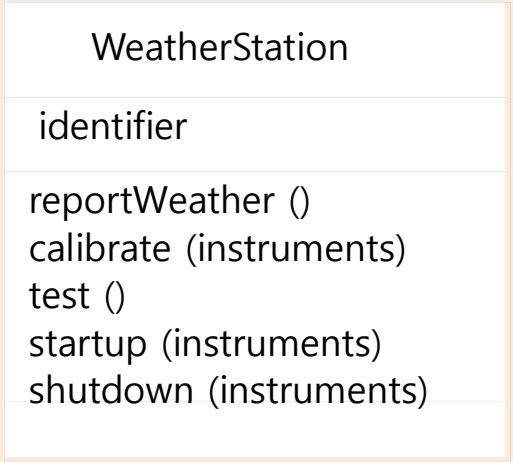
Component Testing

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

- Components 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

Object Class Testing

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising object in all possible states
- Inheritance makes it more difficult to design object class tests
  - Since the information to be tested is not localized.



Need to define test cases for all methods
- reportWeather, calibrate,
- test, startup and shutdown

Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions

For example:
Waiting -> Calibrating -> Testing -> Transmitting -> Waiting

Interface Testing

- To detect faults due to interface errors or invalid assumptions about interfaces
  - Particularly important for object-oriented development as objects are defined by their interfaces

- Guidelines for interface testing
  - Design tests so that parameters to 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

Interface Types

- Interface types
  - Parameter interfaces
    - Data passes from one 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 errors
  - Interface misuse
    - Calling component calls another component and makes an error in its use of its interface.
    - e.g. parameters in the wrong order
  - Interface misunderstanding
    - Calling component embeds assumptions about the behaviour of the called component which are incorrect.
  - Timing errors
    - Called and calling component operate at different speeds and out-of-date information is accessed.

Test Case Design

- Involves designing the test cases (inputs and outputs) used to test the system
  - To create a set of tests that are effective in validation and defect testing

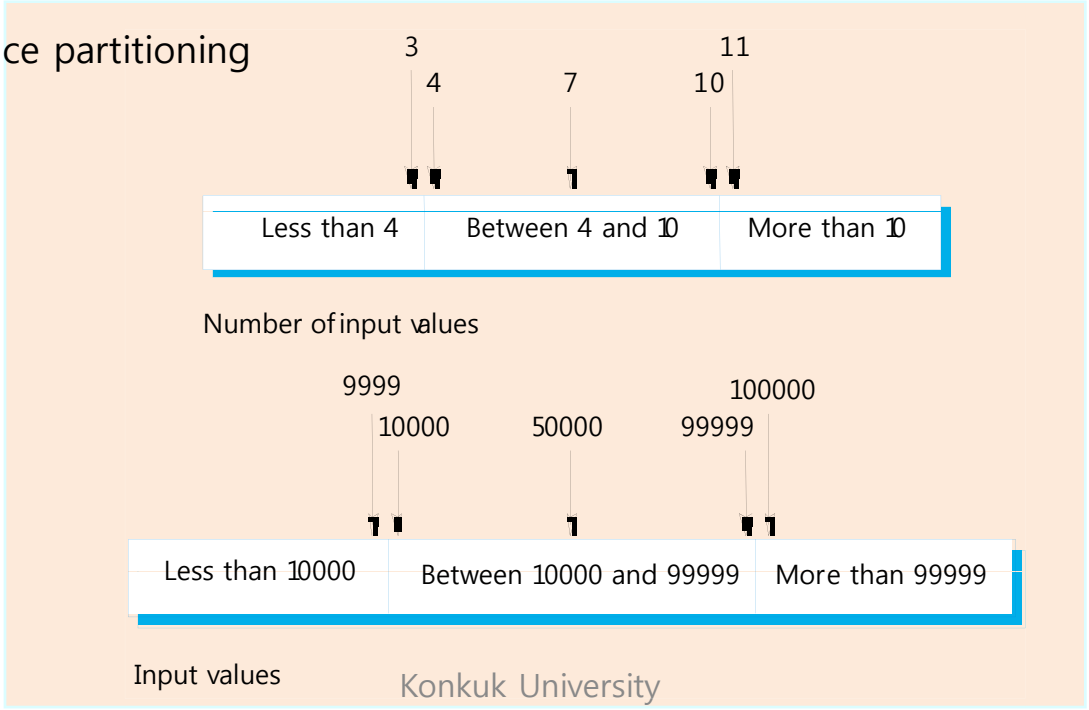
- Test case design approaches
  - Requirements-based testing
  - Partition testing
  - Structural testing

Requirements based Testing

- A general principle of requirements engineering is that requirements should be testable.
- Requirements-based testing is a validation testing technique where you consider each requirement and derive a set of tests for that requirement.

Partition Testing

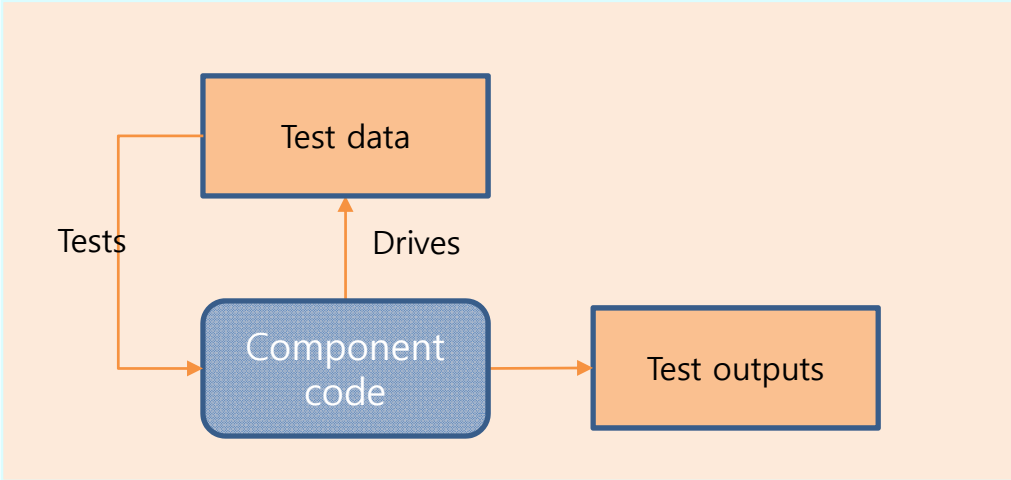
- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
  - Test cases should be chosen from each partition.



Structural Testing

- Sometime called white-box testing.
  - Derives test cases according to program structure.
  - Knowledge of the program is used to identify additional test cases.

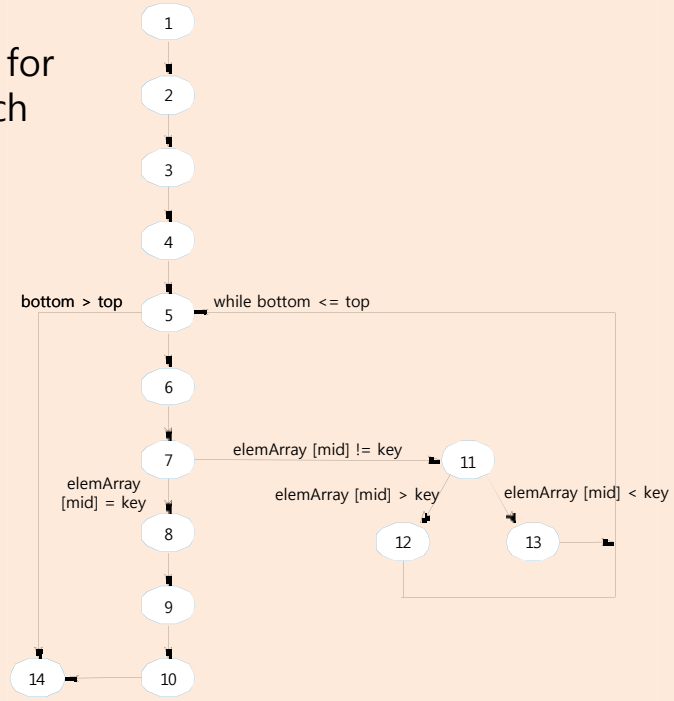
- Objective is to exercise all program statements.
  - A number of structural testing techniques exist, i.e. path testing
  - A number of testing coverage exist.



Path Testing

- To ensure that all the paths in the programs are executed
  - Starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
  - Statements with conditions become nodes in the flow graph.

Flow-graph for binary search



Independent test paths:

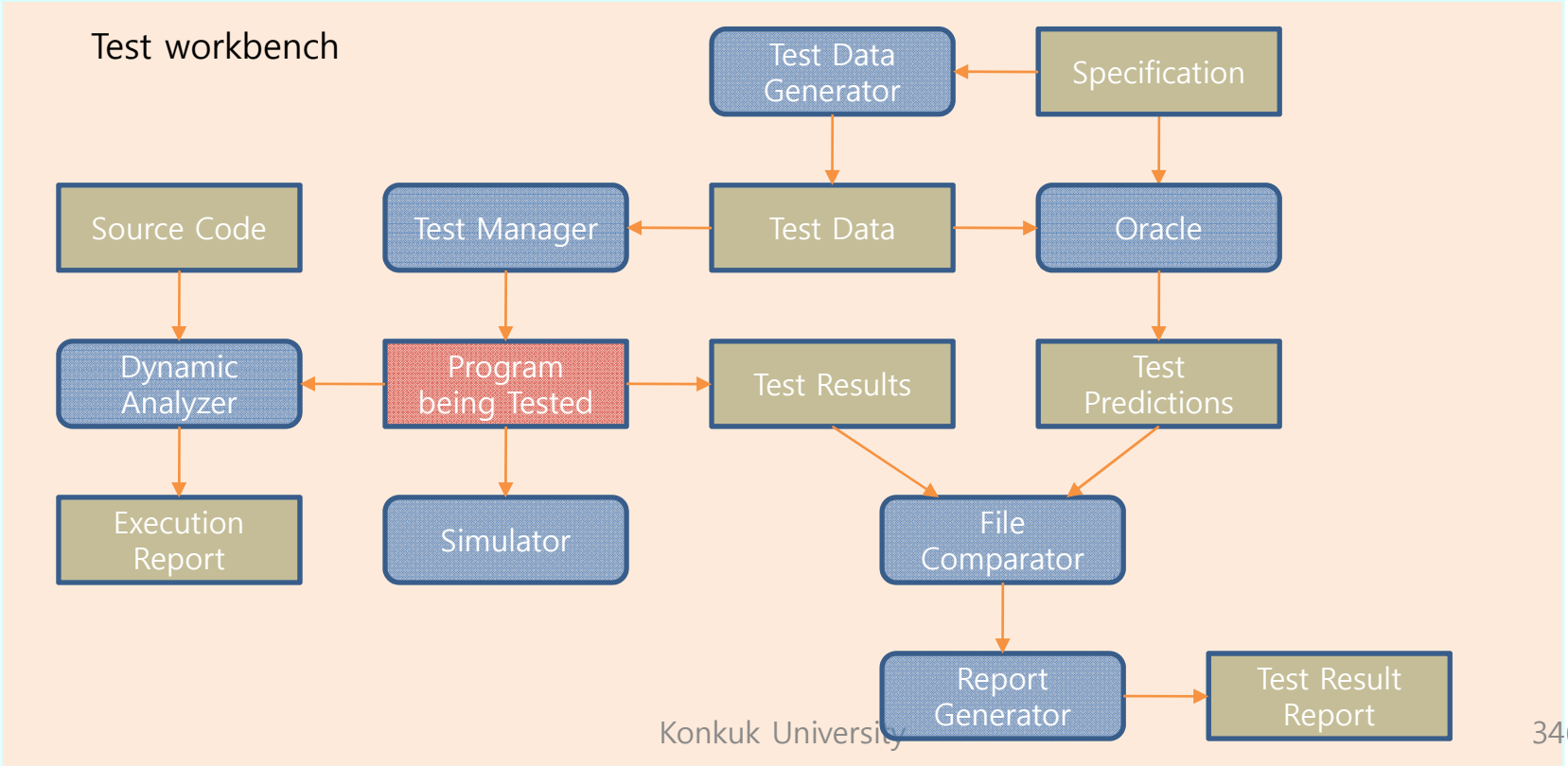
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
1, 2, 3, 4, 5, 14
1, 2, 3, 4, 5, 6, 7, 11, 12, 5, …
1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …

Test cases should be derived so that all of these paths are executed.

A dynamic program analyzer may be used to check that paths have been Executed.

Test Automation

- Testing workbenches provide a range of tools to reduce the time required and total testing costs.
  - Most are open systems, because testing needs are organization-specific.
  - Sometimes difficult to integrate with closed design and analysis workbenches.



Summary

- Testing can show the presence of faults in a system, but it cannot prove there are no remaining faults.
- Component developers are responsible for component testing. System testing is the responsibility of a separate team.
- Integration testing is testing increments of the system. Release testing involves testing a system to be released to a customer.
- Interface testing is designed to discover defects in the interfaces of composite components.
- Equivalence partitioning is a way of discovering test cases - all cases in a partition should behave in the same way.
- Structural analysis relies on analysing a program and deriving tests from this analysis.
- Test automation reduces testing costs by supporting the test process with a range of software tools.

Part VI. Managing People

Chapter 27.

Quality Management

Objectives

- To introduce the quality management process and key quality management activities
- To explain the role of standards in quality management
- To explain the concept of a software metric, predictor metrics and control metrics
- To explain how measurement may be used in assessing software quality and the limitations of software measurement

Software Quality Management

- Concerned with ensuring that the required level of quality is achieved in a software product.
  - Involves defining appropriate quality standards and procedures, and ensuring that these are followed.
  - Should aim to develop a 'quality culture' where quality is seen as everyone's responsibility.

- Quality ?
  - Means a product should meet its specification.

- Quality problems in software systems
  - There is a tension between customer quality requirements (efficiency, reliability, ...) and developer quality requirements (maintainability, reusability, ...)
  - Some quality requirements are difficult to specify in an unambiguous way.
  - Software specifications are usually incomplete and often inconsistent.

Quality Compromise

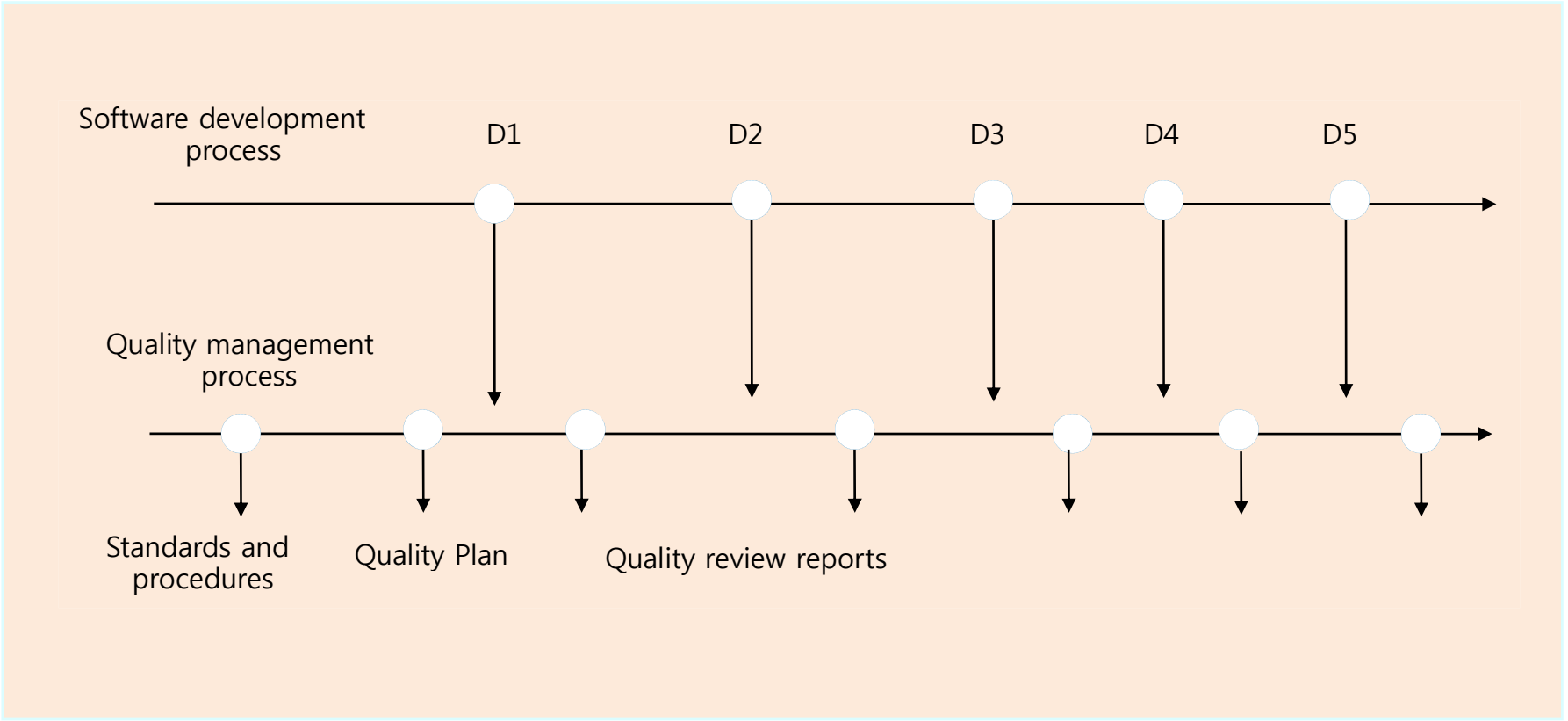
- Cannot wait for specifications to be improved before paying attention to quality management.
- We must put quality management procedures into place to improve quality in spite of imperfect specification.

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

Quality Management Activities

- Quality assurance
  - Establish organisational procedures and standards for quality.
- Quality planning
  - Select applicable procedures and standards for a particular project and modify these as required.
- Quality control
  - Ensure that procedures and standards are followed by the software development team.
- Quality management should be separate from project management to ensure independence.

Quality Management and Software Development



Process and Product Quality

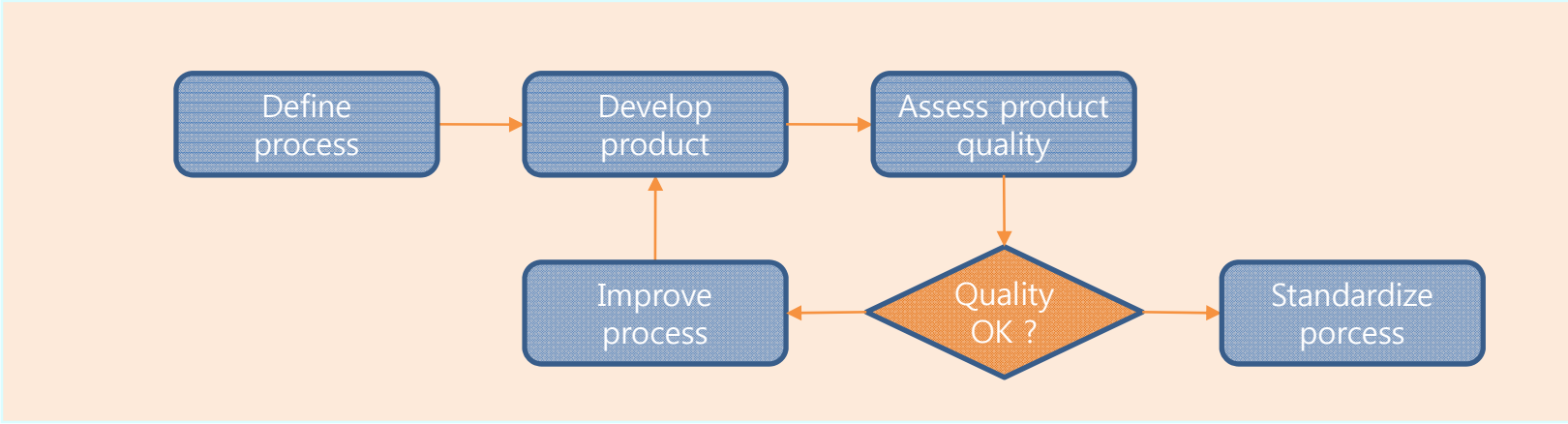
- The quality of a developed product is influenced by the quality of the production process.
  - Important in software development as some product quality attributes are hard to assess.

- However, there is a very complex and poorly understood relationship between software processes and product quality.

Process-based Quality

- There is a straightforward link between process and product in manufactured goods, but more complex for software.
  - Application of individual skills and experience is particularly imporant in software development.
  - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

- Must be careful not to impose inappropriate process standards.



Quality Assurance and Standards

- Standards are the key to effective quality management.
  - May be international, national, organizational or project standards.
  - Encapsulations of best practice

- Product standards
  - define characteristics that all components should exhibit, e.g. a common programming style.
- Process standards
  - define how the software process should be enacted.

Product and Process Standards

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of documents to CM
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Problems with Standards

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

Standards Development

- Standard development involves practitioners in development.
  - Engineers should understand the rationale underlying the standard.
- Standard should be reviewed regularly.
  - Standards can quickly become outdated and this reduces their credibility amongst practitioners.
- Detailed standards should have associated tool support.
  - Excessive clerical work is the most significant complaint against standards.

ISO 9000

- An international set of standards for quality management.
  - Applicable to a range of organizations from manufacturing to service industries.

- ISO 9001
  - Applicable to organisations which design, develop and maintain products.
  - A generic model of the quality process that must be instantiated for each organization using the standard.

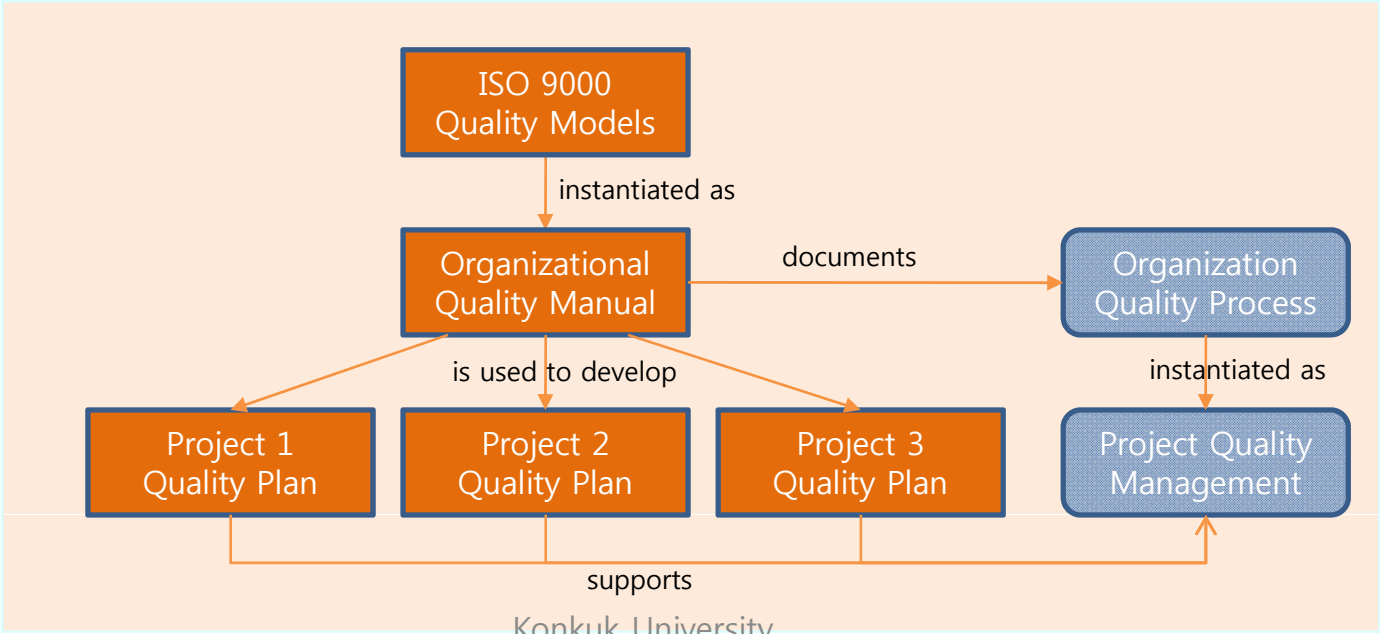
ISO 9001

Management responsibility	Quality system
Control of non-conforming products	Design control
Handling, storage, packaging and delivery	Purchasing
Purchaser-supplied products	Product identification and traceability
Process control	Inspection and testing
Inspection and test equipment	Inspection and test status
Contract review	Corrective action
Document control	Quality records
Internal quality audits	Training
Servicing	Statistical techniques

ISO 9000 Certification

- Quality standards and procedures should be documented in an organizational quality manual.
- An external body may certify that an organization's quality manual conforms to ISO 9000 standards.

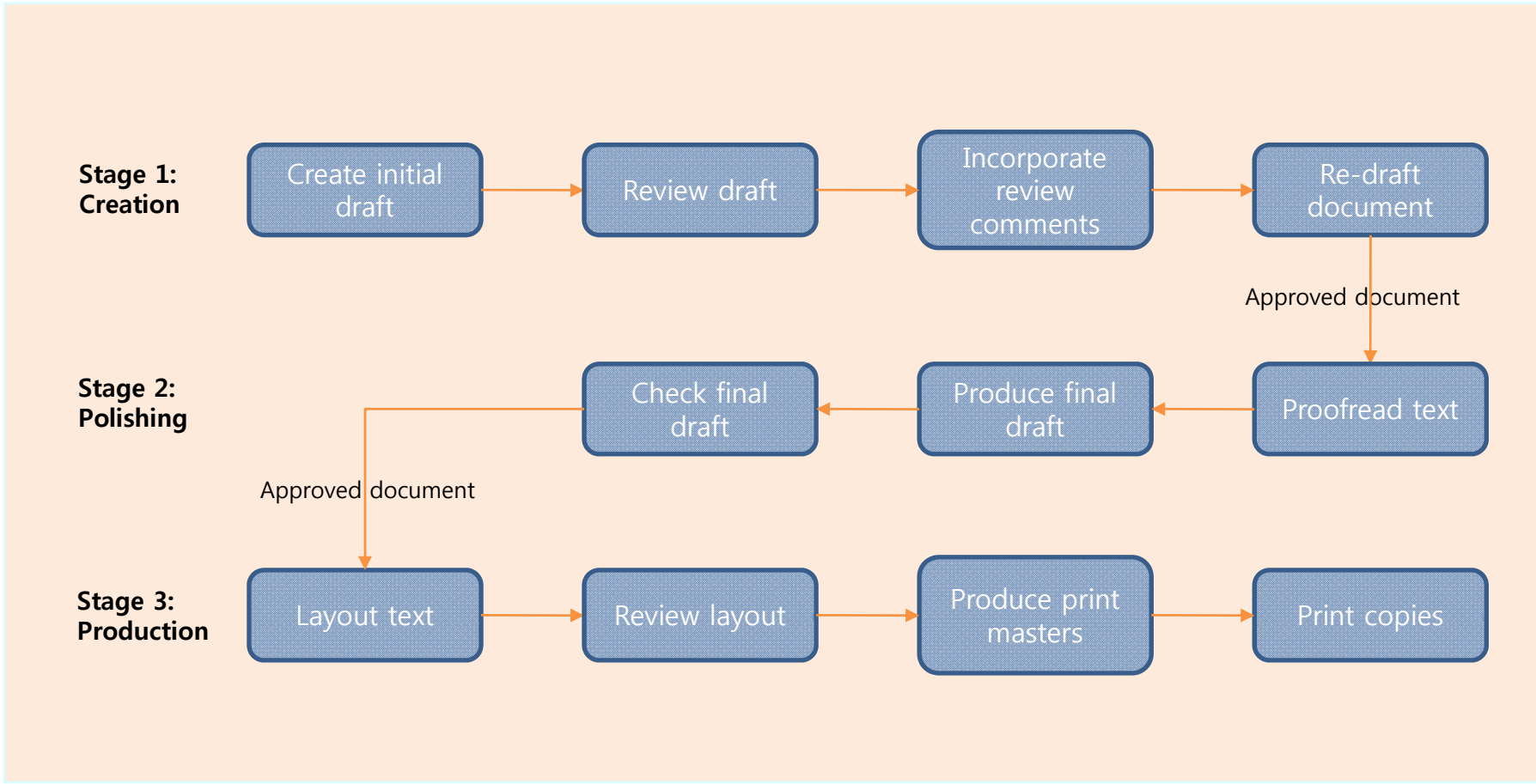
- Some customers require suppliers to be ISO 9000 certified.



Documentation Standards

- Particularly important - documents are the tangible manifestation of the software.
- Documentation process standards
  - Concerned with how documents should be developed, validated and maintained.
- Document standards
  - Concerned with document contents, structure, and appearance.
- Document interchange standards
  - Concerned with the compatibility of electronic documents.

Documentation Process



Document Standards

- Document identification standards
  - How documents are uniquely identified.
- Document structure standards
  - Standard structure for project documents
- Document presentation standards
  - Define fonts and styles, use of logos, etc.
- Document update standards
  - Define how changes from previous versions are reflected in a document.

Document Interchange Standards

- Document interchange standards allow electronic documents to be exchanged, mailed, etc.
  - Needed to define conventions for their use e.g. use of style sheets and macros.

- Need for archiving.
  - The lifetime of word processing systems may be much less than the lifetime of the software being documented.
  - An archiving standard may be defined to ensure that the document can be accessed in future.

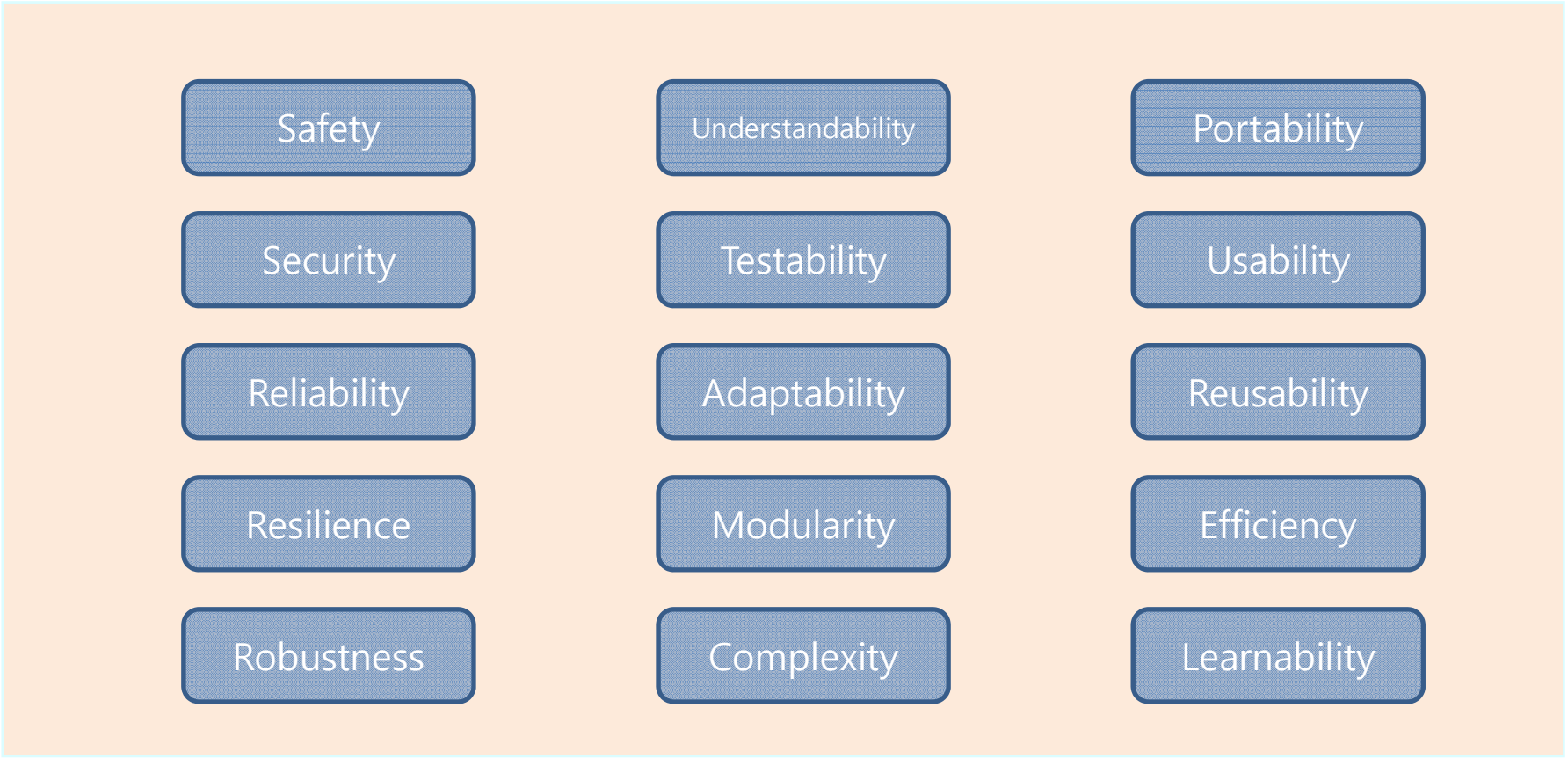
Quality Planning

- Quality plans
  - Set out the desired product qualities and how these are assessed.
  - Defines the most significant quality attributes.
  - Should define the quality assessment process.
  - Set out which, where and when organizational standards be applied.

- Quality plan structure
  - Product introduction
  - Product plans
  - Process descriptions
  - Quality goals
  - Risks and risk management

- Quality plans should be short, succinct documents
  - If they are too long, no-one will read them.

Software Quality Attributes



Quality Control

- Quality control involves checking the software development process to ensure that procedures and standards are being followed.

- Two approaches to quality control
  - Quality reviews
  - Automated software assessment and software measurement

Quality Reviews

- Quality reviews are the principal method of validating the quality of a process or of a product.
- A group examines part or all of a process or system and its documentation to find potential problems.

- Different types of review with different objectives
  - Inspections : for defect removal (product)
  - Reviews : for progress assessment (product and process)
  - Quality reviews (product and standards).

Types of Review

Property	Principal Purpose
Design or Program Inspections	To detect detailed errors in the requirements, design or code. A checklist of possible errors should drive the review.
Progress Reviews	To provide information for management about the overall progress of the project. This is both a process and a product review and is concerned with costs, plans and schedules.
Quality Reviews	To carry out a technical analysis of product components or documentation to find mismatches between the specification and the component design, code or documentation and to ensure that defined quality standards have been followed.

Quality Reviews

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

- Any documents produced in the process may be reviewed.
- Review teams should be relatively small and reviews should be fairly short.
- Records should always be maintained of quality reviews.

Review functions

- Quality function
  - Part of the general quality management process
- Project management function
  - Provide information for project managers.
- Training and communication function
  - Product knowledge is passed between development team members.

Review Results

- Comments made during the review should be classified
  - No action : No change to the software or documentation is required.
  - Refer for repair : Designer or programmer should correct an identified fault.
  - Reconsider overall design : The problem identified in the review impacts other parts of the design. Some overall judgement must be made about the most cost-effective way of solving the problem.

- Requirements and specification errors may have to be referred to the client.

Software Measurement and Metrics

- Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
  - Allows for objective comparisons between techniques and processes.

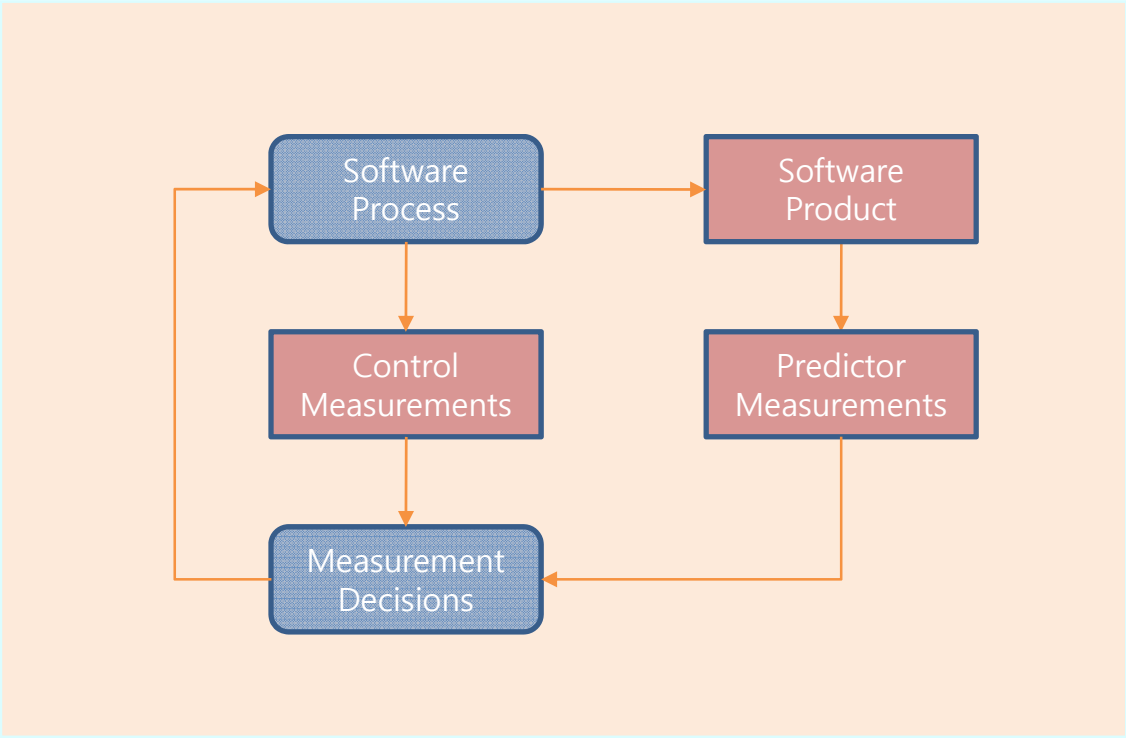
- Although some companies have introduced measurement programs, most organizations still don't make systematic use of software measurement.
- Few established standards in this area.

Software Metric

- Any type of measurement which relates to a software system, process or related documentation
  - Lines of code in a program
  - The Fog index
  - Number of person-days required to develop a component
  - etc.

- Allow the software and the software process to be quantified.
  - May be used to predict product attributes
  - May be used to control the software process.
  - Can be used for general predictions.
  - Can be used to identify anomalous components.

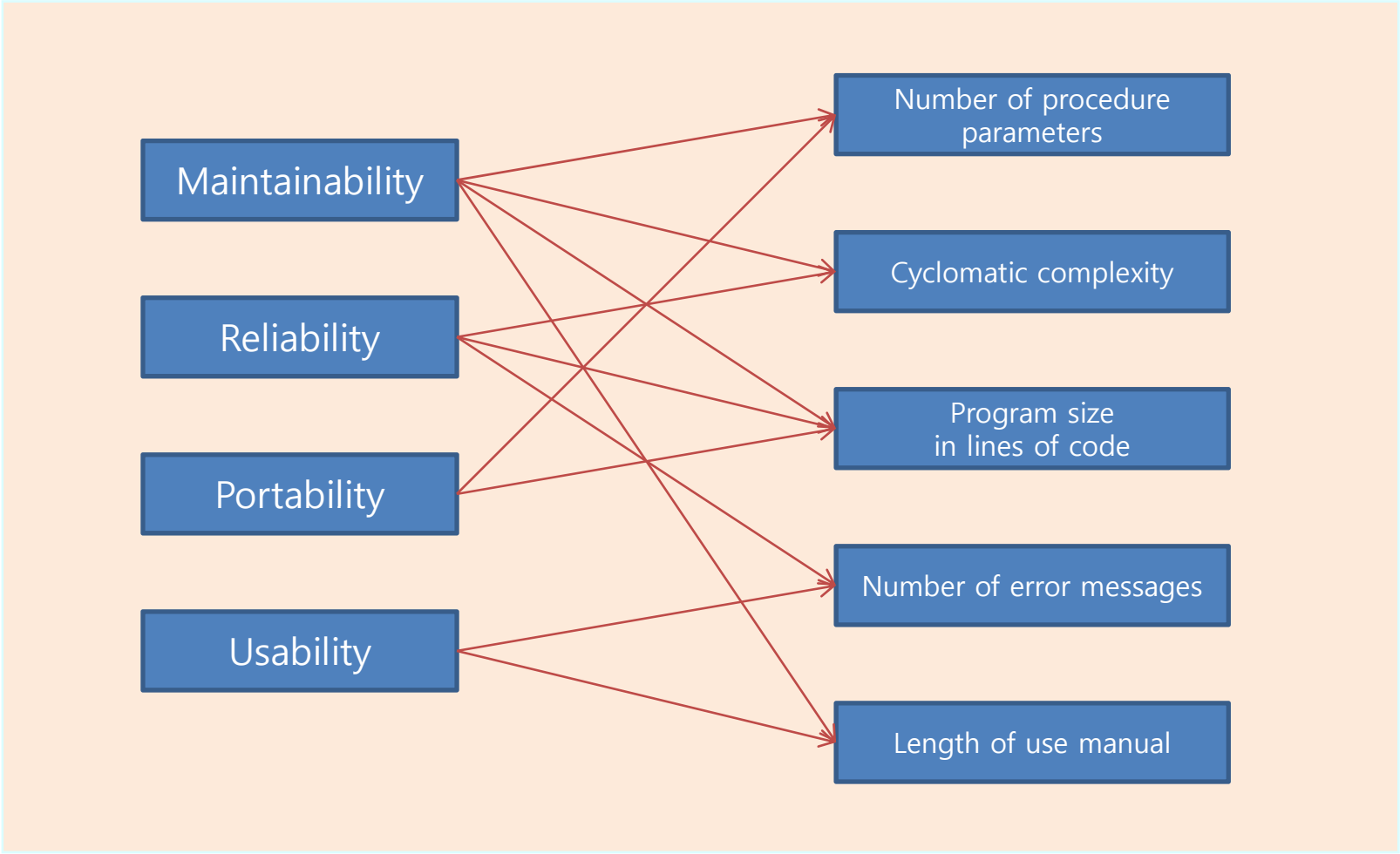
Predictor and Control Metrics



Metrics Assumptions

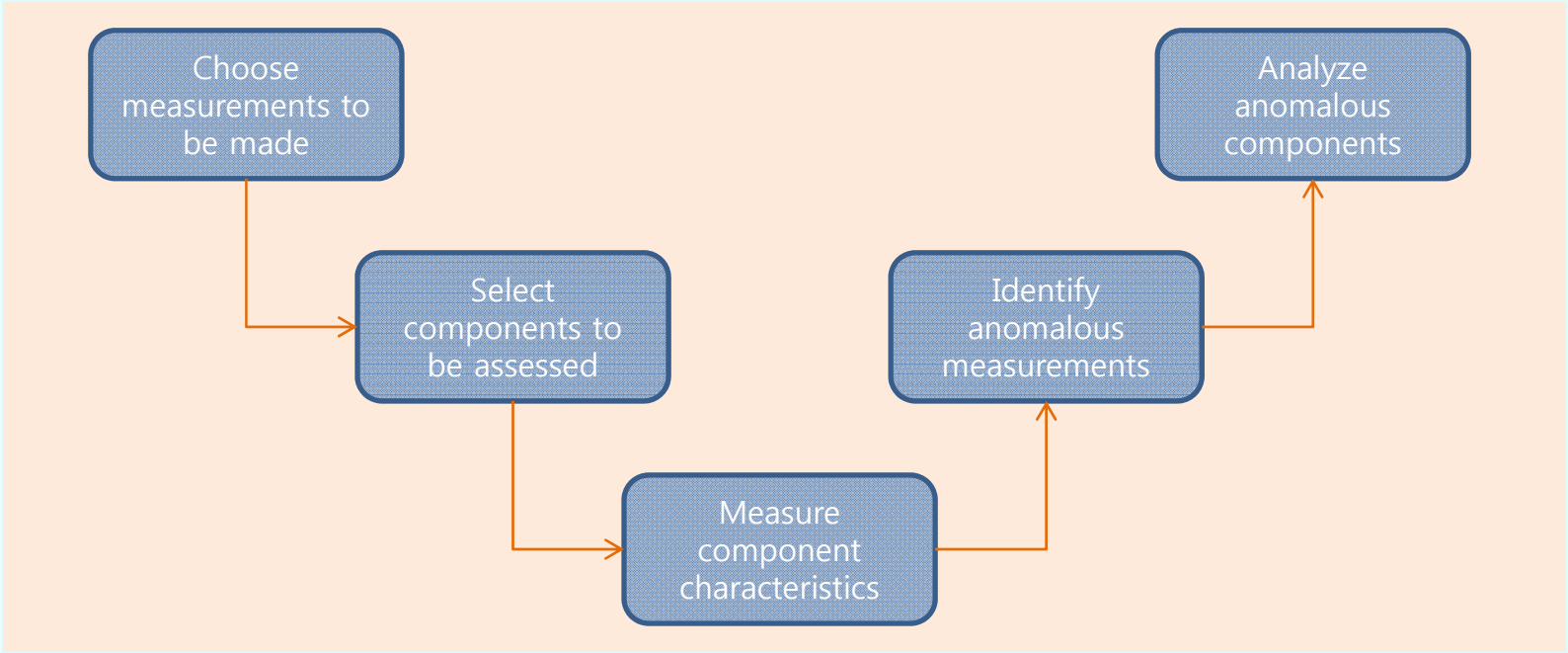
- A software property can be measured.
- The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- This relationship has been formalized and validated.
- It may be difficult to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

Internal and External Attributes



Measurement Process

- A software measurement process may be a part of a quality control process.
  - Data collected during this process should be maintained as an organizational resource.
  - Once a measurement database has been established, comparisons across projects become possible.



Data Collection

- A metrics programme should be based on a set of product and process data.
- Data should be collected immediately (not in retrospect) and, if possible, automatically.

- Three types of automatic data collection
  - Static product analysis
  - Dynamic product analysis
  - Process data collation

Data Accuracy

- Don't collect unnecessary data
  - The questions to be answered should be decided in advance and the required data identified.
- Tell people why the data is being collected.
  - It should not be part of personnel evaluation.
- Don't rely on memory
  - Collect data when it is generated not after a project has finished.

Product Metrics

- A quality metric should be a predictor of product quality.

- Classes of product metric
  - Dynamic metrics : Collected by measurements made of a program in execution.
  - Static metrics : Collected by measurements made of the system representations

  - Dynamic metrics help assess efficiency and reliability.
  - Static metrics help assess complexity, understandability and maintainability.

Dynamic and Static Metrics

- Dynamic metrics are closely related to software quality attributes
  - Relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).

- Static metrics have an indirect relationship with quality attributes
  - Need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

Software Product Metrics

Software Metric	Description
Fan-in / Fan-out	Fan-in is a measure of the number of functions or methods that call some other function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss how to compute cyclomatic complexity in Chapter 22.
Length of identifiers	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document is to understand.

Object-Oriented Metrics

Object-Oriented Metric	Description
Depth of inheritance tree	This represents the number of discrete levels in the inheritance tree where sub-classes inherit attributes and operations (methods) from super-classes. The deeper the inheritance tree, the more complex the design. Many different object classes may have to be understood to understand the object classes at the leaves of the tree.
Method fan-in/fan-out	This is directly related to fan-in and fan-out as described above and means essentially the same thing. However, it may be appropriate to make a distinction between calls from other methods within the object and calls from external methods.
Weighted methods per class	This is the number of methods that are included in a class weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1 and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be more difficult to understand. They may not be logically cohesive so cannot be reused effectively as super-classes in an inheritance tree.
Number of overriding operations	This is the number of operations in a super-class that are over-ridden in a sub-class. A high value for this metric indicates that the super-class used may not be an appropriate parent for the sub-class.

Measurement Analysis

- It is not always obvious what data means
  - Analysing collected data is very difficult.

- Professional statisticians should be consulted if available.
- Data analysis must take local circumstances into account.

Summary

- Software quality management is concerned with ensuring that software meets its required standards.
- Quality assurance procedures should be documented in an organizational quality manual.

- Software standards are an encapsulation of best practice.
- Reviews are the most widely used approach for assessing software quality.
- Software measurement gathers information about both the software process and the software product.
- Product quality metrics should be used to identify potentially problematical components.
- There are no standardized and universally applicable software metrics.

Chapter 28.

Process Improvement

Objectives

- To explain the principles of software process improvement
- To explain how software process factors influence software quality and productivity
- To explain how to develop simple models of software processes
- To explain the notion of process capability and the CMMI process improvement model

Process Improvement

- Understanding existing processes and introducing process changes to improve product quality, reduce costs or accelerate schedules.

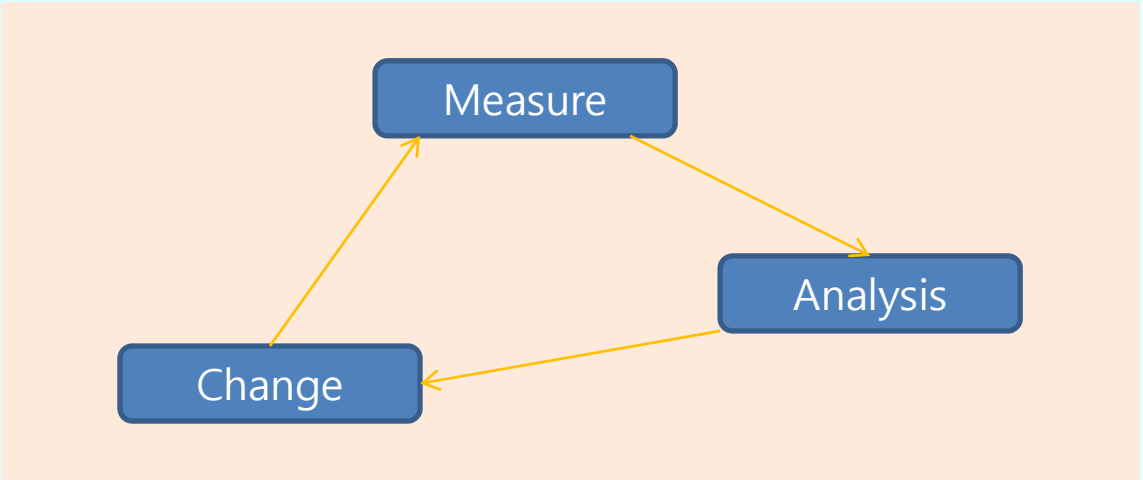
- Most process improvement work so far has focused on defect reduction. This reflects the increasing attention paid by industry to quality.
- However, other process attributes can also be the focus of improvement.

Process Attributes

Process Attributes	Description
Understandability	To what extent is the process explicitly defined and how easy is it to understand the process definition?
Visibility	Do the process activities culminate in clear results so that the progress of the process is externally visible?
Supportability	To what extent can CASE tools be used to support the process activities?
Acceptability	Is the defined process acceptable to and usable by the engineers responsible for producing the software product?
Reliability	Is the process designed in such a way that process errors are avoided or trapped before they result in product errors?
Robustness	Can the process continue in spite of unexpected problems?
Maintainability	Can the process evolve to reflect changing organisational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?

Process Improvement Cycle

- Process measurement
  - Attributes of the current process are measured. These are a baseline for assessing improvements.
- Process analysis
  - The current process is assessed and bottlenecks and weaknesses are identified.
- Process change
  - Changes to the process that have been identified during the analysis are introduced.

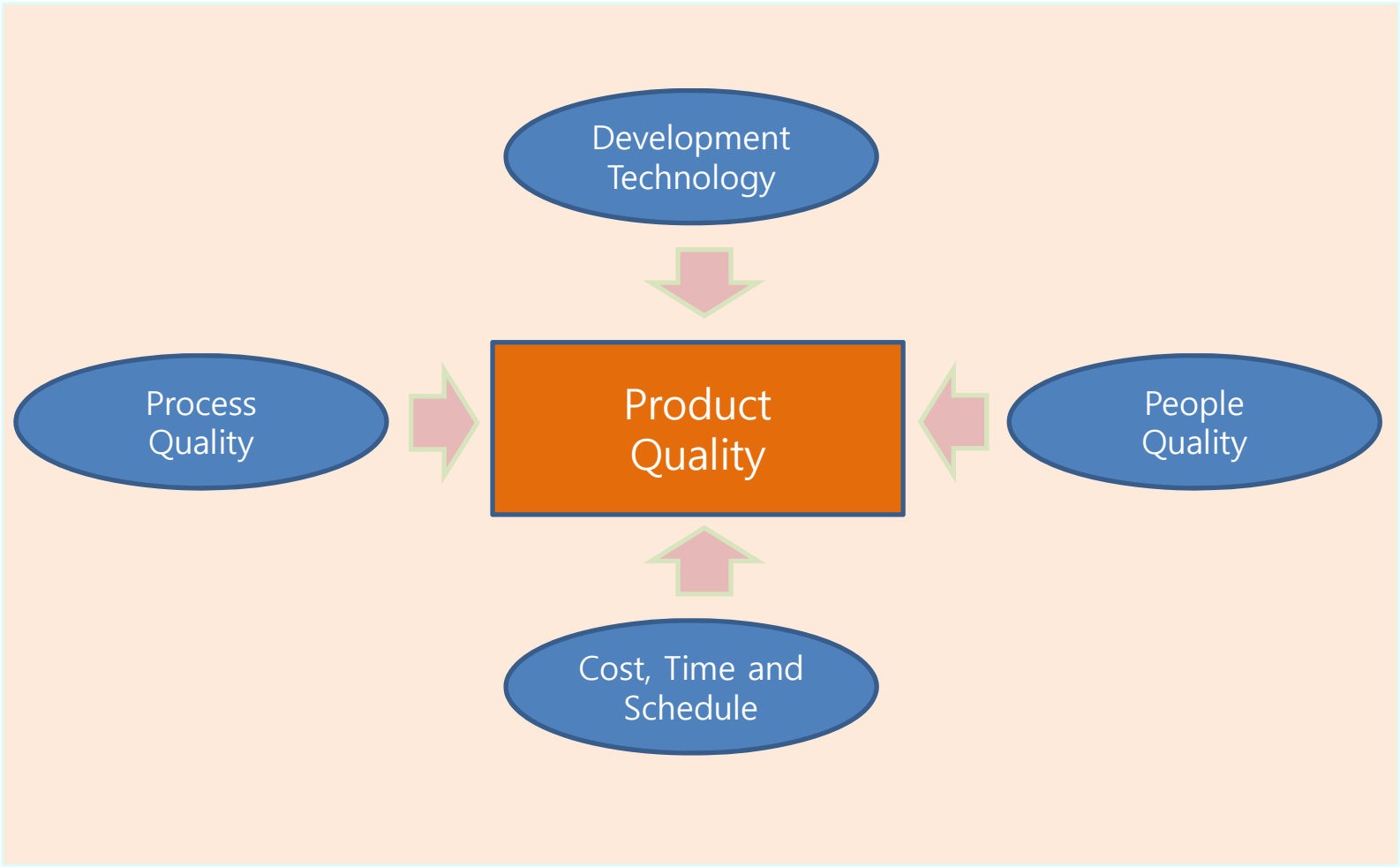


Process and Product Quality

- Process quality and product quality are closely related.
  - The quality of the product depends on its development process.

- A good process is usually required to produce a good product.
  - For manufactured goods, process is the principal quality determinant.
  - For design-based activity, other factors are also involved especially the capabilities of the designers.

Principal Product Quality Factors



Quality Factors

- For large projects with 'average' capabilities, the development process determines product quality.

- For small projects, the capabilities of the developers is the main determinant.
- The development technology is particularly significant for small projects.

- In all cases, if an unrealistic schedule is imposed then product quality will suffer.

Process Classification

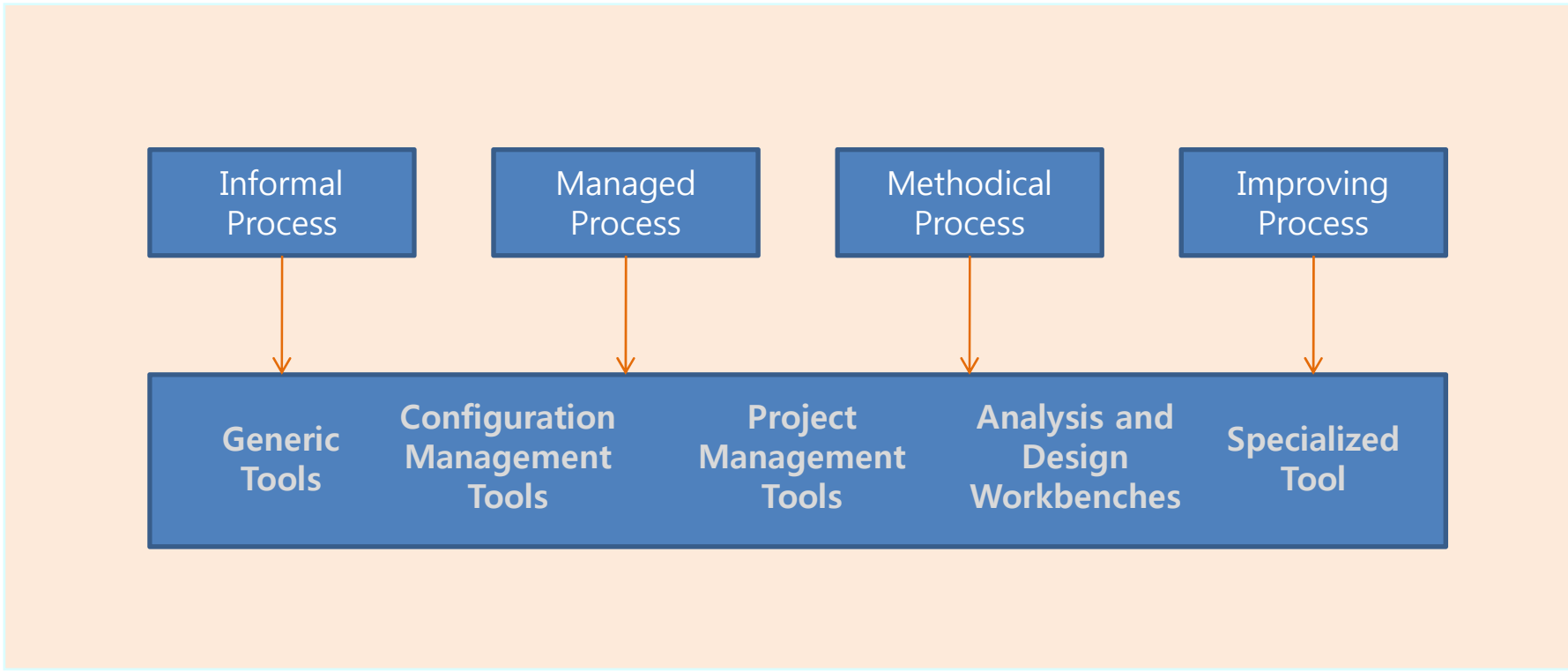
- Informal
  - No detailed process model.
  - Development team chose their own way of working.
- Managed
  - Defined process model which drives the development process.
- Methodical
  - Processes supported by some development method such as the RUP.
- Supported
  - Processes supported by automated CASE tools.

Process Choice

- Process used should depend on type of product being developed
  - For large systems, management is usually the principal problem so we need a strictly managed process.
  - For smaller systems, more informality is possible.

- No uniformly applicable process which should be standardized within an organisation
  - High costs may be incurred if you force an inappropriate process on a development team.
  - Inappropriate methods can also increase costs and lead to reduced quality.

Process Tool Support



Process Measurement

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

- Process measurements should be used to assess process improvements
  - But, this does not mean that measurements should drive the improvements.
  - The improvement driver should be the organizational objectives.

Classes of Process Measurement

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

Goal-Question-Metric Paradigm

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

Process Analysis and Modelling

- Process analysis
  - Study existing processes to understand the relationships between parts of the process and to compare them with other processes.

- Process modelling
  - Documentation of a process which records the tasks, the roles and the entities used
  - May be presented from different perspectives.

- Study an existing process to understand its activities.
- Produce an abstract model of the process.
  - Normally represent the model graphically.
  - Several different views (e.g. activities, deliverables, etc.) may be required.
- Analyse the model to discover process problems.
  - Involves discussing process activities with stakeholders and discovering problems and possible process changes.

Process Analysis Techniques

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

Process Model Elements 1

Process Model Elements	Graphical Notation	Description
Activity	A round-edged rectangle with no drop shadow	An activity has a clearly defined objective, entry and exit conditions. Examples of activities are preparing a set of test data to test a module, coding a function or a module, proof-reading a document, etc. Generally, an activity is atomic i.e. it is the responsibility of one person or group. It is not decomposed into sub-activities.
Process	A round-edged rectangle with drop shadow	A process is a set of activities which have some coherence and whose objective is generally agreed within an organisation. Examples of processes are requirements analysis, architectural design, test planning, etc.
Deliverable	A rectangle with drop shadow	A deliverable is a tangible output of an activity that is predicted in a project plan.
Condition	A parallelogram	A condition is either a pre-condition that must hold before a process or activity can start or a post-condition that holds after a process or activity has finished.
Role	A circle with drop	A role is a bounded area of responsibility. Examples of roles might be configuration manager, test engineer, software designer, etc. One person may have several different roles and a single role may be associated with several different people.
Exception	May be represented as a double edged box	An exception is a description of how to modify the process if some anticipated or unanticipated event occurs. Exceptions are often undefined and it is left to the ingenuity of the project managers and engineers to handle the exception.
Communication	An arrow	An interchange of information between people or between people and supporting computer systems. Communications may be informal or formal. Formal communications might be the approval of a deliverable by a project manager; informal communications might be the interchange of electronic mail to resolve ambiguities in a document.

Process Exceptions

- Software processes are complex and process models cannot effectively represent how to handle exceptions
  - Several key people becoming ill just before a critical review.
  - A breach of security that means all external communications are out of action for several days.
  - Organizational reorganization
  - A need to respond to an unanticipated request for new proposals.

- Under these circumstances, the model is suspended and managers use their initiative to deal with the exception.
- We have to avoid the exceptions or change the process itself.

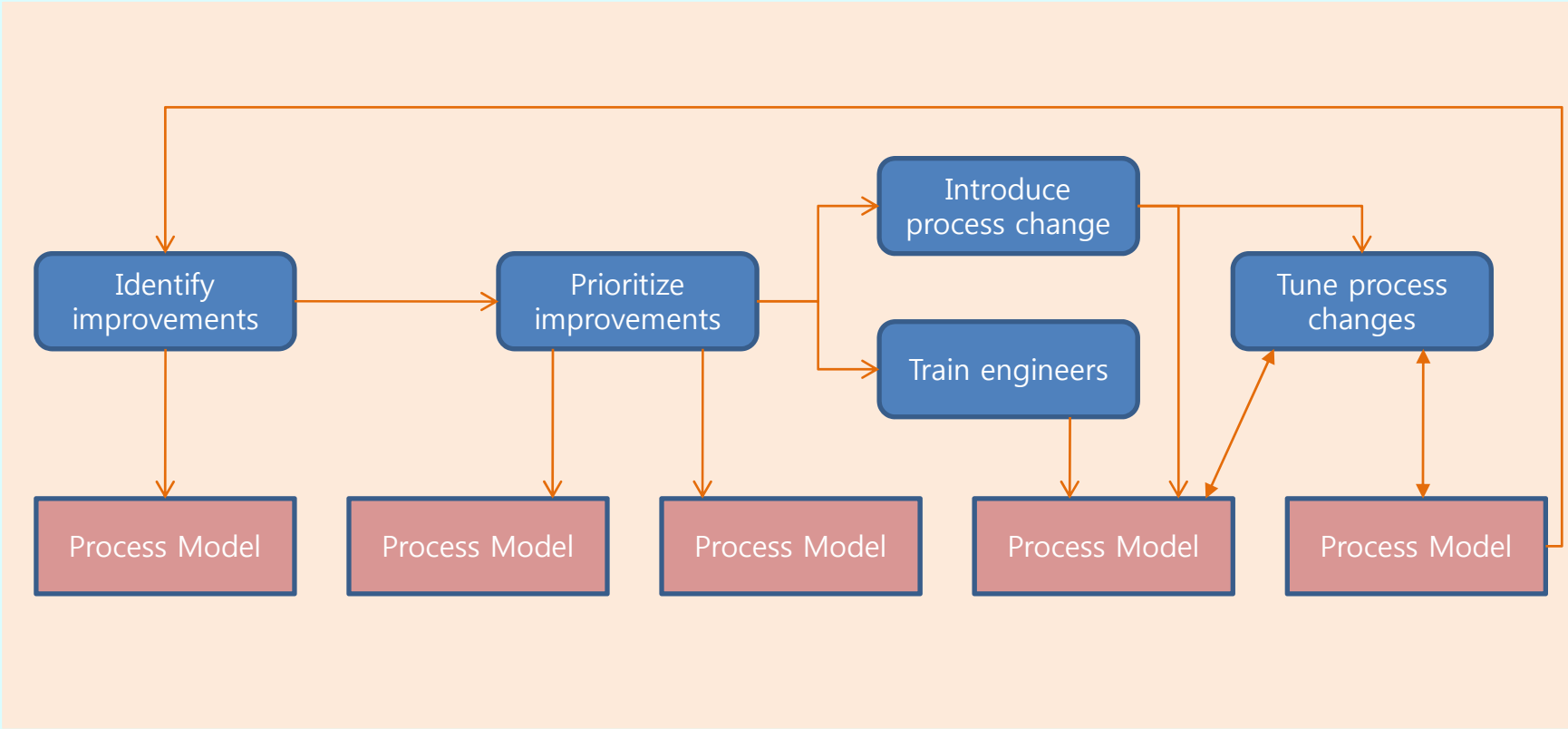
Process Change

- Process changes involve making modifications to existing processes.
  - Introduce new practices, methods or processes.
  - Change the ordering of process activities.
  - Introduce or remove deliverables.
  - Introduce new roles or responsibilities.

- Change should be driven by measurable goals.

- Process change stages
  - Improvement identification
  - Improvement prioritization
  - Process change introduction
  - Process change training
  - Change tuning

Process Change Process



The CMMI Framework

- The CMMI framework is the current stage of work on process assessment and improvement.
  - Started at the SEI(Software Engineering Institute) in the 1980s.
  - The SEI's mission is to promote software technology transfer particularly to US defence contractors.

- It has had a profound influence on process improvement
  - Capability Maturity Model introduced in the early 1990s.
  - Revised maturity framework (CMMI) introduced in 2001.

Process Capability Assessment

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

The SEI Capability Maturity Model

- Initial
  - Essentially uncontrolled.
- Repeatable
  - Product management procedures defined and used.
- Defined
  - Process management procedures and strategies defined and used.
- Managed
  - Quality management strategies defined and used.
- Optimizing
  - Process improvement strategies defined and used.

Problems with the CMM

- Practices associated with model levels
  - Companies could be using practices from different levels at the same time, but if all practices from a lower level were not used, it was not possible to move beyond that level.
- Discrete rather than continuous
  - Did not recognize distinctions between the top and the bottom of levels.
- Practice-oriented
  - Concerned with how things were done (the practices) rather than the goals to be achieved.

The CMMI Model

- An integrated capability model that includes software and systems engineering capability assessment.

- Two instantiations
  - Staged where the model is expressed in terms of capability levels.
  - Continuous where a capability rating is computed.

CMMI model components

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

CMMI Process Areas

CMMI Process Area	Description
Process Management	Organisational process definition Organisational process focus Organisational training Organisational process performance Organisational innovation and deployment
Project Management	Project planning Project monitoring and control Supplier agreement management Integrated project management Risk management Integrated teaming Quantitative project management
Engineering	Requirements management Requirements development Technical solution Product integration Verification Validation
Support	Configuration management Process and product quality management Measurement and analysis Decision analysis and resolution Organisational environment for integration Causal analysis and resolution

CMMI Goals

CMMI Goals	Process Area
Corrective actions are managed to closure when the project's performance or results deviate significantly from the plan.	**Project Monitoring and control**
Actual performance and progress of the project is monitored against the project plan.	**Project monitoring and control**
The requirements are analysed and validated and a definition of the required functionality is developed.	**Requirements development**
Root causes of defects and other problems are systematically determined.	**Causal analysis and resolution**
The process is institutionalised as a defined process.	**Generic goal**

CMMI Practices

Practice	Associated Goal
Analyse derived requirements to ensure that they are necessary and sufficient	The requirements are analysed and validated and a definition of the required functionality is developed.
Validate requirements to ensure that the resulting product will perform as intended in the user's environment using multiple techniques as appropriate.	
Select the defects and other problems for analysis.	Root causes of defects and other problems are systematically determined.
Perform causal analysis of selected defects and other problems and propose actions to address them.	
Establish and maintain an organisational policy for planning and performing the requirements development process.	The process is institutionalised as a defined process.
Assign responsibility and authority for performing the process, developing the work products and providing the services of the requirements development process.	

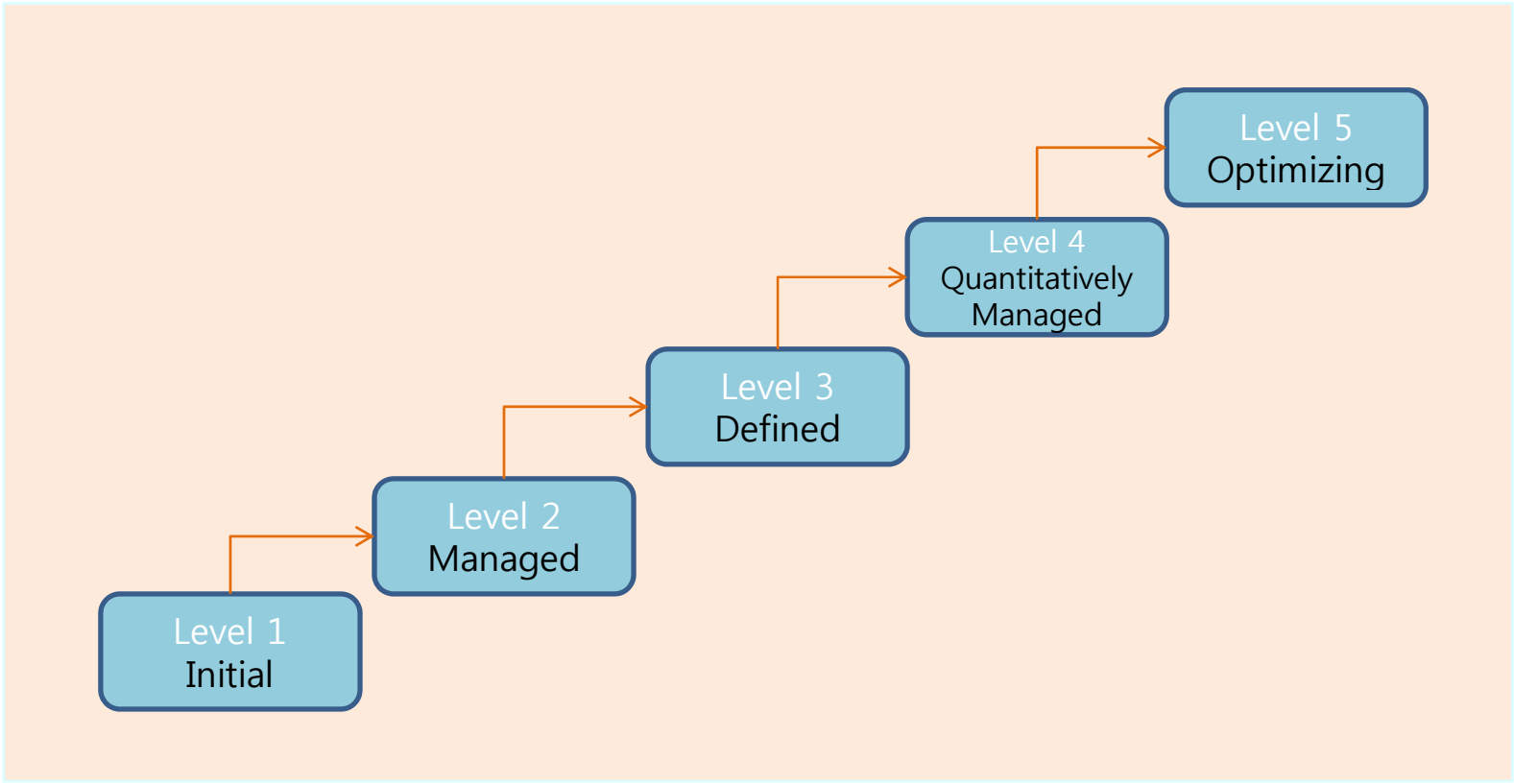
CMMI Assessment

- Examines the processes used in an organization and assesses their maturity in each process area.

- Based on a 6-point scale (6 levels)
  - Not performed
  - Performed
  - Managed
  - Defined
  - Quantitatively managed
  - Optimizing

The Staged CMMI Model

- Comparable with the software CMM.
- Each maturity level has process areas and goals.



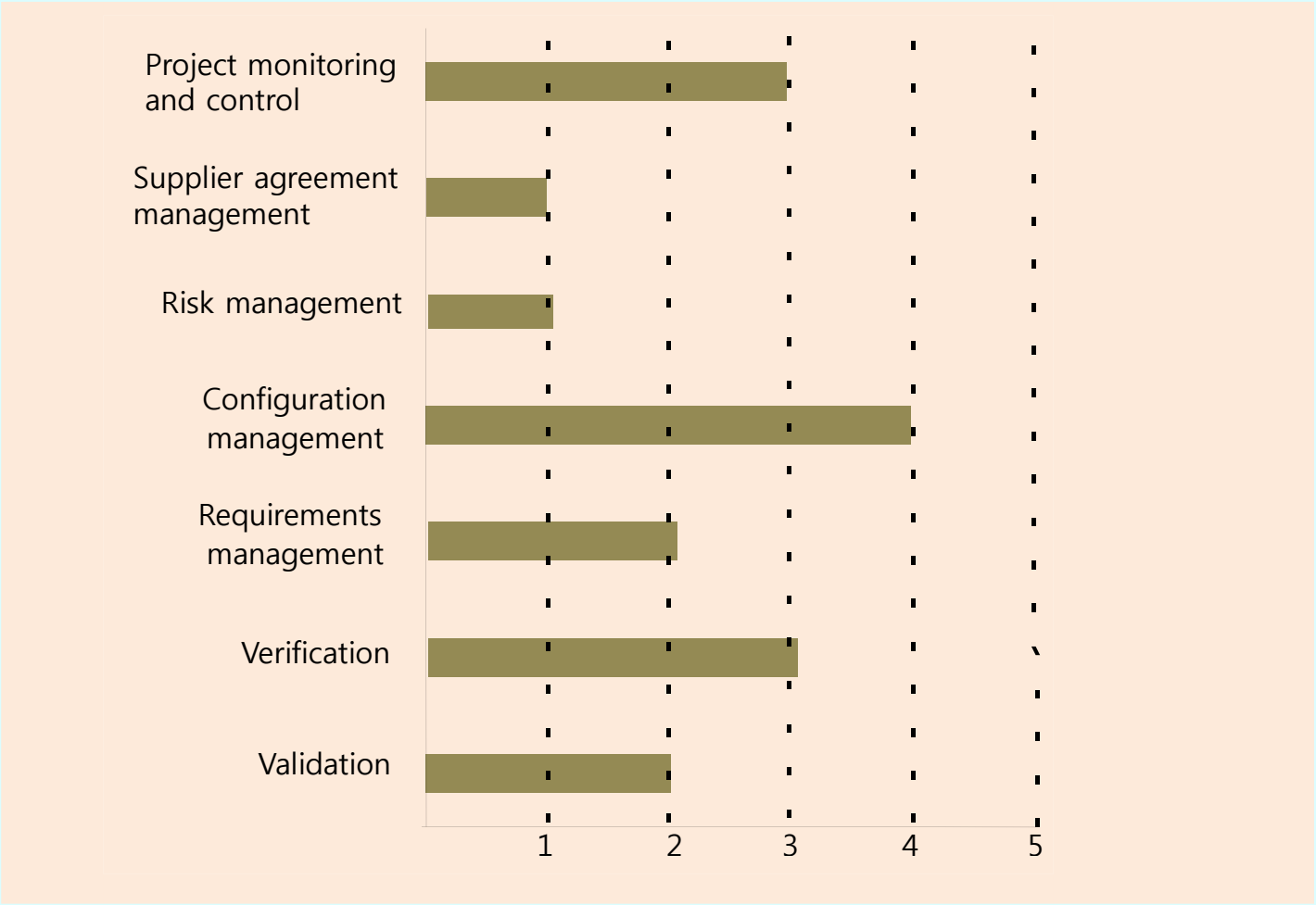
Institutional Practices

- Institutions operating at the managed level should have institutionalized practices that are geared to standardization. (Level 2 → Level 3)
  - Establish and maintain policy for performing the project management process.
  - Provide adequate resources for performing the project management process.
  - Monitor and control the project planning process.
  - Review the activities, status and results of the project planning process.

The Continuous CMMI Model

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

A Process Capability Profile



Summary

- Process improvement involves process analysis, standardisation, measurement and change.
- Processes can be classified as informal, managed, methodical and improving. This classification can be used to identify process tool support.
- The process improvement cycle involves process measurement, process analysis and process change.
- Process measurement should be used to answer specific process questions, based on organisational improvement goals.
- The three types of process metrics used in the measurement process are time metrics, resource utilisation metrics and event metrics.
- Process models include descriptions of tasks, activities, roles, exceptions, communications, deliverables and other processes.
- The CMMI process maturity model integrates software and systems engineering process improvement.
- Process improvement in the CMMI model is based on reaching a set of goals related to good software engineering practice.

Chapter 29.

Configuration Management

Objectives

- To explain the importance of software configuration management (CM)
- To describe key CM activities namely CM planning, change management, version management and system building
- To discuss the use of CASE tools to support configuration management processes

Configuration Management

- New versions of software systems are created as they change
  - For different machines/OS
  - Offering different functionality
  - Tailored for particular user requirements

- Configuration management(CM) is concerned with managing evolving software systems
  - System change is a team activity.
  - Aims to control the costs and effort involved in making changes.
  - Involves the development and application of procedures and standards to manage an evolving software product.
  - May be seen as part of a more general quality management process.
  - When released to CM, software systems are sometimes called baselines.

CM Standards

- CM should always be based on a set of standards which are applied within an organization.
  - Standards should define how items are identified, how changes are controlled and how new versions are managed.
  - Standards may be based on external CM standards (e.g. IEEE standard for CM).
  - Some existing standards are based on a waterfall process model.
  - New CM standards are needed for evolutionary development.

Frequent System Building

- Frequent system building
  - A new version of a system is built from components by compiling and linking them.
  - This new version is delivered for testing using pre-defined tests.
  - Faults that are discovered during testing are documented and returned to the system developers.

- It is easier to find problems that stem from component interactions early in the process.
  - This encourages thorough unit testing - developers are under pressure not to 'break the build'.
  - A stringent change management process is required to keep track of problems that have been discovered and repaired.

Configuration Management Planning

- All products of the software process may have to be managed
  - Specifications
  - Designs
  - Programs
  - Test data
  - User manuals

- Thousands of separate documents may be generated for a large, complex software system.

The Configuration Management Plan

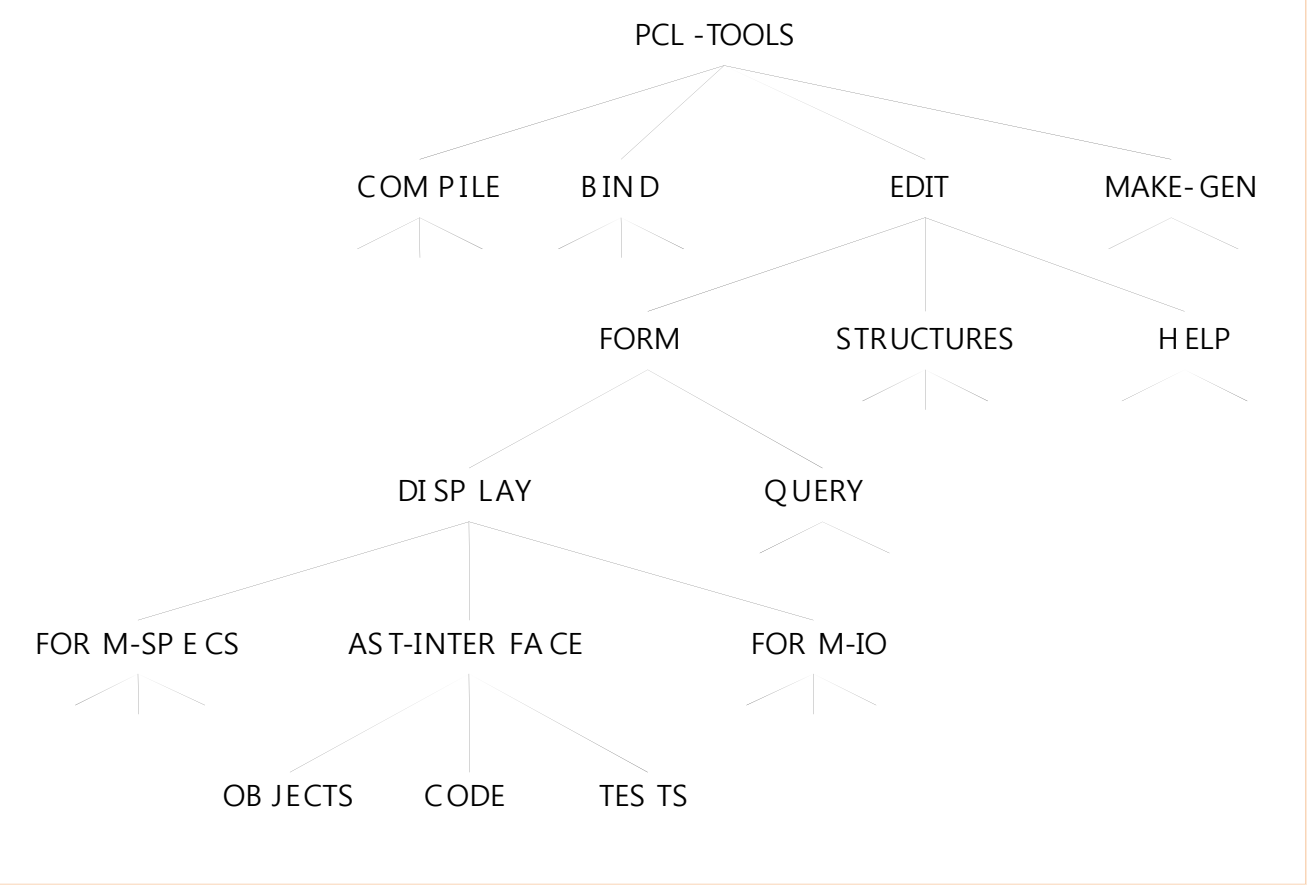
- Defines the types of documents to be managed and a document naming scheme.
- Defines who takes responsibility for the CM procedures and creation of baselines.
- Defines policies for change control and version management.
- Defines the CM records which must be maintained.
- Describes the tools which should be used to assist the CM process and any limitations on their use.
- Defines the process of tool use.
- Defines the CM database used to record configuration information.
- May include information such as the CM of external software, process auditing, etc.

Configuration Item Identification

- Large projects typically produce thousands of documents which must be uniquely identified.
- Some of these documents must be maintained for the lifetime of the software.

- Document naming scheme should be defined so that related documents have related names.
- A hierarchical scheme with multi-level names is probably the most flexible approach.
  - PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE

Configuration Hierarchy



Configuration Database

- All CM information should be maintained in a configuration database.

- This should allow queries about configurations to be answered
  - Who has a particular system version?
  - What platform is required for a particular version?
  - What versions are affected by a change to component X?
  - How many reported faults in version T?

- The CM database should preferably be linked to the software being managed.

CM Database Implementation

- May be part of an integrated environment to support software development.
  - The CM database and the managed documents are all maintained on the same system.
- CASE tools may be integrated.
  - A close relationship between the CASE tools and the CM tools.
- More commonly, the CM database is maintained separately as this is cheaper and more flexible.

Change Management

- Software systems are subject to continual change requests
  - from users
  - from developers
  - from market forces

- Change management is concerned with
  - Keeping track of these changes
  - Ensuring that they are implemented in the most cost-effective way.

Change Management Process

Request change by completing a change request form
Analyze change request
if change is valid**then**
    Assess how change might be implemented
    Assess change cost
    Submit request to change control board
  if change is accepted**then**
     repeat
        make changes to software
        submit changed software for quality approval
    until software quality is adequate
    create new system version
else
    reject change request
else
    reject change request

Change Request Form

- A change request form records
  - The change proposed
  - Requestor of change
  - The reason why change was suggested
  - The urgency of change (from requestor of the change)

- It also records
  - Change evaluation
  - Impact analysis
  - Change cost
  - Recommendations from system maintenance staff

Change Request Form

Change Request Form

Project: Proteus/PCL-Tools **Number:** 23/02
Change requester: I. Sommerville **Date:** 1/12/02
Requested change: When a component is selected from the structure, display the name of the file where it is stored.

Change analyser: G. Dean **Analysis date:** 10/12/02
Components affected: Display-Icon.Select, Display-Icon.Display

Associated components: FileTable

Change assessment: Relatively simple to implement as a file name table is available. Requires the design and implementation of a display field. No changes to associated components are required.

Change priority: Low

Change implementation:
Estimated effort: 0.5 days

Date to CCB: 15/12/02 **CCB decision date:** 1/2/03
CCB decision: Accept change. Change to be implemented in Release 2.1.
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments

Change Tracking Tools

- A major problem in change management is tracking change status.

- Change tracking tools
  - Keep track the status of each change request .
  - Ensure automatically that change requests are sent to the right people at the right time.
  - Integrated with E-mail systems allowing electronic change request distribution.

Change Control Board

- Changes should be reviewed by an external group who decide whether or not they are cost-effective from a strategic and organizational viewpoint rather than a technical viewpoint.

- The group is called a change control board(CCB).
  - May include representatives from client and contractor staff.

Derivation History

- Derivation history is a record of changes applied to a document or code component.
  - Should record, in outline,
    - The change made
    - The rationale for the change
    - Who made the change
    - When it was implemented.
  - May be included as a comment in code.

Component Header Information

```
// BANKSEC project (IST 6087)
//
// BANKSEC-TOOLS/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: N. Perwaiz
// Creation date: 10th November 2002
//
// © Lancaster University 2002
//
// Modification history
// Version  Modifier     Date        Change       Reason
// 1.0      J. Jones     1/12/2002   Add header   Submitted to CM
// 1.1      N. Perwaiz   9/4/2003    New field    Change req. R07/02
```

Version and Release Management

- Version and release management
  - Invent an identification scheme for system versions.
  - Plan when a new system version is to be produced.
  - Ensure that version management procedures and tools are properly applied.
  - Plan and distribute new system releases.

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

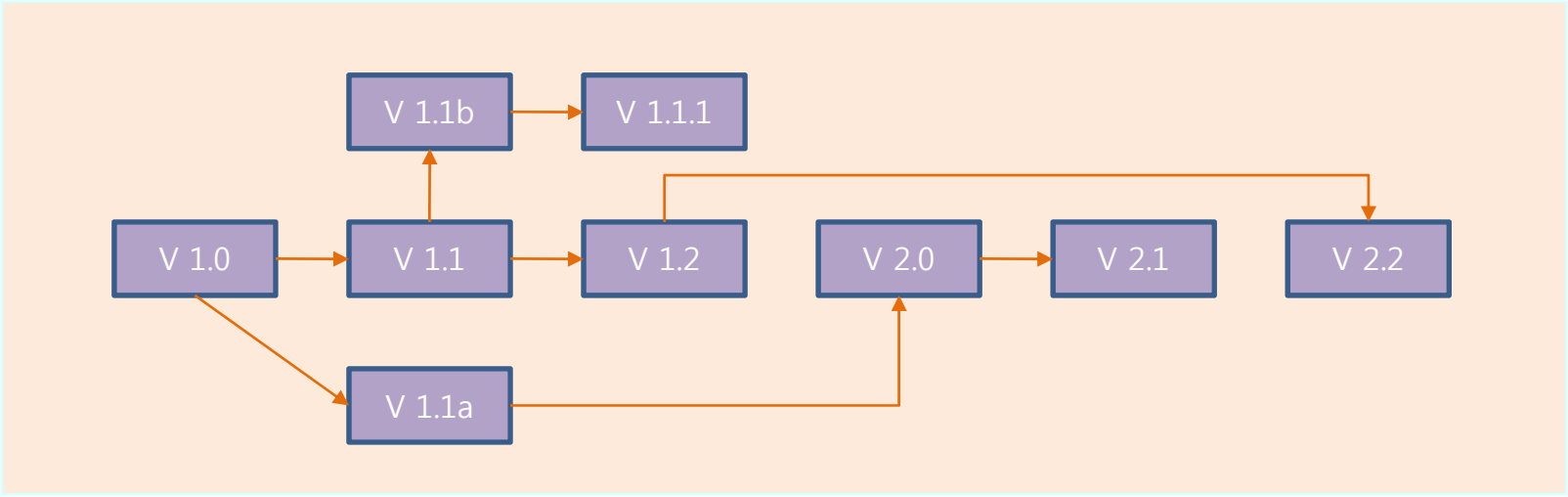
Version Identification

- Version identification should define an unambiguous way of identifying component versions.

- Three basic techniques for component identification
  - Version numbering
  - Attribute-based identification
  - Change-oriented identification

Version Numbering

- Simple naming scheme uses a linear derivation.
  - V1, V1.1, V1.2, V2.1, V2.2 etc.
- The actual derivation structure is a tree or a network rather than a sequence.
  - Version names are not meaningful.
  - A hierarchical naming scheme leads to fewer errors in version identification.



Attribute-Based Identification

- Attributes can be associated with a version with the combination of attributes identifying that version
  - Examples of attributes are Date, Creator, Programming Language, Customer, Status etc.

- More flexible than an explicit naming scheme for version retrieval.
  - May cause problems with uniqueness.
  - The set of attributes have to be chosen so that all versions can be uniquely identified.

- In practice, a version also needs an associated name for easy reference.
  - Example: AC3D (language =Java, platform = XP, date = Jan 2003)

Change-Oriented Identification

- Change-oriented identification integrates versions and the changes made to create these versions.
  - Used for systems rather than components.
  - Each proposed change has a change set that describes changes made to implement that change.
  - Change sets are applied in sequence so that, in principle, a version of the system that incorporates an arbitrary set of changes may be created.

Release Management

- Releases must incorporate changes forced on the system by errors discovered by users and by hardware changes.
  - Must also incorporate new system functionality.

- Release planning is concerned with when to issue a system version as a release.

System Releases

- System release is not just a set of executable programs
- May also include
  - Configuration files defining how the release is configured for a particular installation
  - Data files needed for system operation
  - An installation program or shell script to install the system on target hardware
  - Electronic and paper documentation
  - Packaging and associated publicity

Release Decision Making

- All files required for a release should be re-created when a new release is installed.

- Preparing and distributing a system release is an expensive process.
- Factors such as the technical quality of the system, competition, marketing requirements and customer change requests should all influence the decision of when to issue a new system release.

System Release Strategy

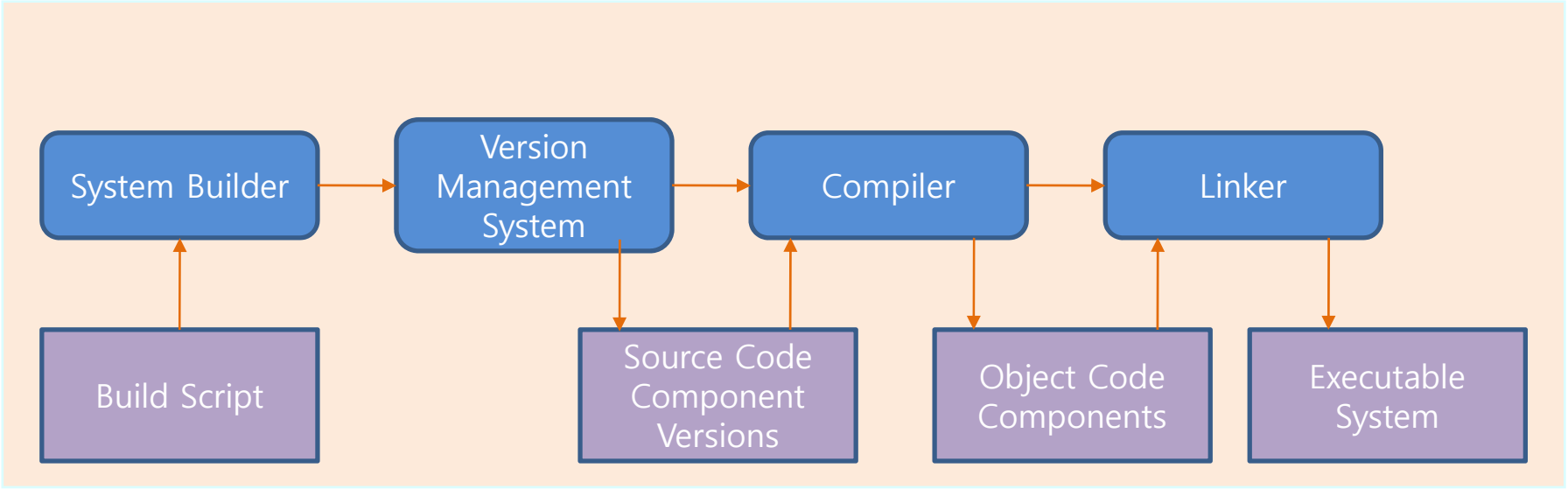
Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. However, minor system faults may be repaired by issuing patches (often distributed over the Internet) that can be applied to the current release of the system.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Lehman's fifth law (See chapter 21)	This suggests that the increment of functionality that is included in each release is approximately constant. Therefore, if there has been a system release with significant new functionality, then it may have to be followed by a repair release.
Competition	A new system release may be necessary because a competing product is available.
Marketing requirements	The marketing department of an organisation may have made a commitment for releases to be available at a particular date.
Customer change proposals	For customised systems, customers may have made and paid for a specific set of system change proposals and they expect a system release as soon as these have been implemented.

Release Creation

- Release creation involves collecting all files and documentation required to create a system release.
  - Configuration descriptions have to be written for different hardware.
  - Installation scripts have to be written.
  - The specific release must be documented to record exactly what files were used to create it. This allows it to be re-created if necessary.

System Building

- The process of compiling and linking software components into an executable system
  - Different systems are built from different combinations of components.
  - Now always supported by automated tools that are driven by 'build scripts'.



System Building Problems

- Do the build instructions include all required components?
  - When there are many hundreds of components making up a system, it is easy to miss one out. This should normally be detected by the linker.
- Is the appropriate component version specified?
  - A more significant problem. A system built with the wrong version may work initially but fail after delivery.
- Are all data files available?
  - The build should not rely on 'standard' data files. Standards vary from place to place.

System Building Problems

- Are data file references within components correct?
  - Embedding absolute names in code almost always causes problems as naming conventions differ from place to place.
- Is the system being built for the right platform
  - Sometimes you must build for a specific OS version or hardware configuration.
- Is the right version of the compiler and other software tools specified?
  - Different compiler versions may actually generate different code and the compiled component will exhibit different behaviour.

CASE Tools for Configuration Management

- CASE tool support for CM is essential, because
  - CM processes are standardized and involve applying pre-defined procedures.
  - Large amounts of data must be managed.

- Mature CASE tools to support configuration management are available ranging from stand-alone tools to integrated CM workbenches.

CM Workbenches

- Open workbenches
  - Tools for each stage in the CM process are integrated through organizational procedures and scripts.
  - Gives flexibility in tool selection.

- Integrated workbenches
  - Provide whole-process, integrated support for configuration management.
  - More tightly integrated tools so easier to use.
  - However, the cost is less flexibility in the tools used.

Change Management Tools

- Change management is a procedural process so it can be modelled and integrated with a version management system.

- Change management tools
  - Form editor to support processing the change request forms
  - Workflow system to define who does what and to automate information transfer
  - Change database that manages change proposals and is linked to a VM system
  - Change reporting system that generates management reports on the status of change requests

Version Management Tools

- Version and release identification
  - Systems assign identifiers automatically when a new version is submitted to the system.
- Storage management.
  - System stores the differences between versions rather than all the version code.
- Change history recording
  - Record reasons for version creation.
- Independent development
  - Only one version at a time may be checked out for change. Parallel working on different versions.
- Project support
  - Can manage groups of files associated with a project rather than just single files.

System Building

- Building a large system is computationally expensive and may take several hours.
- Hundreds of files may be involved.

- System building tools may provide
  - A dependency specification language and interpreter
  - Tool selection and instantiation support
  - Distributed compilation
  - Derived object management

Summary

- Configuration management is the management of system change to software products.
- A formal document naming scheme should be established and documents should be managed in a database.
- The configuration data base should record information about changes and change requests.
- A consistent scheme of version identification should be established using version numbers, attributes or change sets.
- System releases include executable code, data, configuration files and documentation.
- System building involves assembling components into a system.
- CASE tools are available to support all CM activities.
- CASE tools may be stand-alone tools or may be integrated systems which integrate support for version management, system building and change management.