

Introduction: Software and Software Engineering

Chapter 1

**King Saud University
College of Computer and Information Sciences
Department of Computer Science**

Dr. S. HAMMAMI

**S.H
2016**

Objectives

- ❖ Understand what software engineering is and why it is important;
- ❖ Know the answers to key questions that provide an introduction to software engineering;
- ❖ Understand some ethical and professional issues that are important for software engineers.
- ❖ To introduce the Systems Development Life Cycle “SDLC”

FAQs about software engineering

- ❖ What is software?
- ❖ What is software engineering?
- ❖ What is the difference between software engineering and computer science?
- ❖ What is the difference between software engineering and system engineering?
- ❖ What is a software process?
- ❖ What is **a software process model?**
- ❖ What are the costs of software engineering?
- ❖ What are software engineering methods?
- ❖ What is CASE (Computer-Aided Software Engineering)
- ❖ What are the attributes of good software?
- ❖ What are the key challenges facing software engineering?



What is Software?

- The economies of ALL developed nations are dependent on software
- More and more systems are software controlled
- *Software engineering* is concerned with theories, methods and tools for professional software development
- Expenditure on software represents a significant fraction of GNP in all developed countries

What is Software?

- The software industry's impact on the US GDP is massive, accounting for some \$1.14 trillion in 2016, according to a report from Software.org: the BSA Foundation. What's more, the industry boosted the economy in all 50 states.
- According to the report, that impact on the GDP marks a \$70 billion increase over the past two years. While the US economy as a whole grew 6.7%, software's direct economic impact grew by 18.7%, the report found.
- Jobs were also heavily impacted by the software industry. Some 2.9 million people in the US are directly employed in software, but the industry indirectly supports 10.5 million US jobs.

(<https://www.techrepublic.com/article/software-industry-boasts-us-gdp-by-1-14-trillion-grows-economy-in-all-50-states/>)

What is Software?

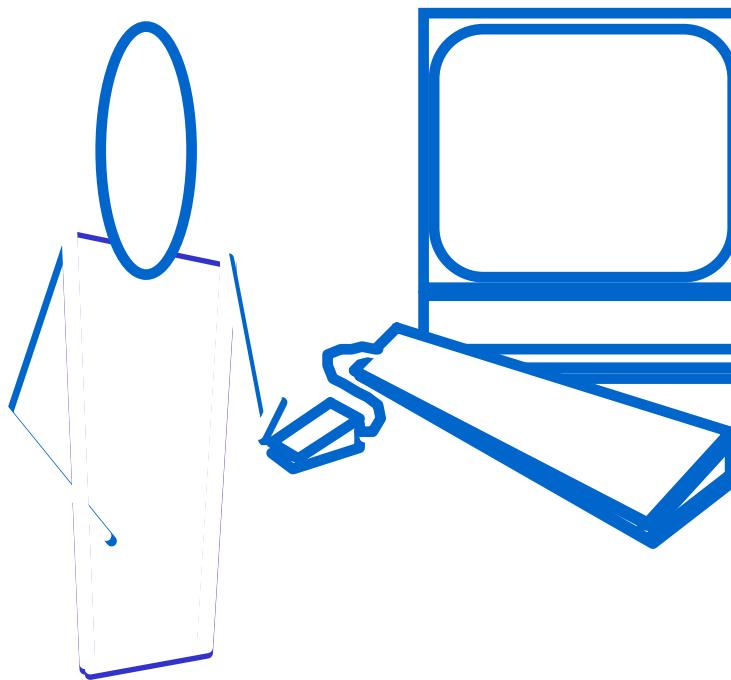
- The economies of ALL developed nations are dependent on software
- More and more systems are software controlled
- *Software engineering* is concerned with theories, methods and tools for professional software development
- Expenditure on software represents a significant fraction of GNP in all developed countries

What is Software?

Software is a set of items or objects
that form a “configuration” that
includes

- programs
- documents
- data ...

that is needed to
make these programs
operate correctly



The Nature of Software...

Software is abstract and intangible

- Hard to understand development effort

Software is easy to reproduce

- Cost is in its *development*
—in other engineering products, manufacturing is the costly stage

The industry of the software is labor-intensive

- The process of the development is hard to automate

Types of Software...

Software products may be developed for a particular customer or may be developed for a general market:

Custom (Bespoke)

- For a specific customer according to their specification.
- Examples: Gmail, KSU application(نظام مدار){<https://madar.ksu.edu.sa/>)

Generic

- Sold on open market
- Examples: databases, word processors,

Embedded

- Built into hardware
- Hard to change
- Examples: Software in Camera, Microwave, Elevator etc

Types of Software

Real time software

- E.g. control and monitoring systems
- Must react immediately
- Safety often a concern
- Examples: Fire Alarm and Control System, Anti missile system

Data processing software

- Used to run businesses
- Accuracy and security of data are key

Some software has both aspects

Variety of Software Products

Examples

Real time:

air traffic control

Embedded systems:

digital camera, GPS

Data processing:

telephone billing, pensions

Information systems:

web sites, digital libraries

Sensors:

weather data

System software:

operating systems, compilers

Communications:

routers, mobile telephones

Offices:

word processing, video conferences

Scientific:

simulations, weather forecasting

Graphical:

film making, design

etc., etc., etc.,

Software engineering diversity

- ✧ There are many different types of software system and there is no universal set of software techniques that is applicable to all of these
- ✧ The software engineering methods and tools used depend on the type of application being developed, the requirements of the customer and the background of the development team



Essential attributes of good software

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and Security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

What is Software Engineering?...

The process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints.

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

❖ **Engineering discipline**

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

❖ **All aspects of software production**

- Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

Importance of software engineering

- ✧ More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

- ✧ It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.



Software engineering fundamentals

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

Frequently asked questions about software engineering

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

Frequently asked questions about software engineering

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

Software process activities

- A software process is a set of activities and their output, which result in a software product:

Requirements and specification

Includes

- Domain analysis
- Defining the problem
- Requirements gathering
 - » Obtaining input from as many sources as possible
- Requirements analysis
 - » Organizing the information
- Requirements specification
 - » Writing detailed instructions about how the software should behave

Software process activities

Design

Deciding how the requirements should be implemented, using the available technology

Includes:

- Systems engineering: Deciding what should be in hardware and what in software
- Software architecture: Dividing the system into subsystems and deciding how the subsystems will interact
- Detailed design of the internals of a subsystem
- User interface design
- Design of databases

Software process activities

Implementation

Translate designs into a working system

- Coding
- Testing
- Documentation
- Data conversion (from old to new system)
- Training
- Installation

Software process activities

Maintenance: Evolving system

- Requirements WILL CHANGE to reflect dynamic environment of business
- Continuous process
- Maintenance types:
 - Corrective: correct existing defects
 - Perfective: improve
 - Adaptive: to new environment / requirements

Software Process model

- An abstract representation of a software process, presented from a particular perspective; for example, workflow (sequence of activities), data-flow (information flow), or role/action (who does what)
- These process models explain different approaches to software development; for example, Waterfall, Iterative, and Component Based Software Engineering



Software Engineering methods

- Structured approaches to software development, including:
 - Model descriptions: Describes graphical models (i.e. object, data-flow, state machine models, etc)
 - Rules: Constraints applied to system models (i.e. entities must have unique names)
 - Recommendations: Best practices for designing software (i.e. include no more than nine processes in a data flow diagram)
 - Process guidance: what activities to follow (i.e. document object attributes before defining its operations)
- Examples of methods:
 - Functional oriented: DeMarco's Structured Analysis and Jackson's JSD
 - Object oriented: Booch, Rumbaugh, and Boehm's Object Oriented methods, Rational Unified Process

CASE: Computer-Aided Software Engineering

- Computer-aided software engineering (CASE) is software to support software development and evolution processes.
- Activity automation
 - Graphical editors for system model development;
 - Data dictionary to manage design entities;
 - Graphical UI builder for user interface construction;
 - Debuggers to support program fault finding;
 - Automated translators to generate new versions of a program.

Key points [Professional Software Development]

- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production
- ✧ Essential software product attributes are maintainability, dependability and security, efficiency and acceptability
- ✧ The high-level activities of specification, development, validation and evolution are part of all software processes
- ✧ The fundamental notions of software engineering are universally applicable to all types of system development

Key points [Professional Software Development]

- ✧ There are **many different types of system** and each requires appropriate software engineering tools and techniques for their development
- ✧ The **fundamental ideas** of software engineering are **applicable** to all types of software system



Software engineering ethics

- ✧ Software engineering involves wider responsibilities than simply the application of technical skills
- ✧ Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals
- ✧ Ethical behaviour is more than simply upholding the law but involves following a set of principles that are morally correct



Issues of professional responsibility

❖ Confidentiality

- Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed

❖ Competence

- Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence

Issues of professional responsibility

❖ Intellectual property rights

- Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.

❖ Computer misuse

- Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

ACM/IEEE Code of Ethics

- ✧ The professional societies in the US have cooperated to produce a code of ethical practice
- ✧ Members of these organisations sign up to the code of practice when they join
- ✧ **The Code** contains **eight Principles** related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession

The ACM/IEEE Code of Ethics

Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Ethical principles

1. **PUBLIC** - Software engineers shall act consistently with the public interest.
2. **CLIENT AND EMPLOYER** - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. **PRODUCT** - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. **JUDGMENT** - Software engineers shall maintain integrity and independence in their professional judgment.
5. **MANAGEMENT** - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. **PROFESSION** - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. **COLLEAGUES** - Software engineers shall be fair to and supportive of their colleagues.
8. **SELF** - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Ethical principles: Examples

A software engineer is employed by the company that designs and manufactures voting machines. She is employed to develop an archival system for retaining voting information and results.

Which of the following actions would most likely be a violation of the Software Engineering Code of Ethics and Professional Practice? Give the Software Engineering Code of Ethics related to your choice?

- **a. She failed to keep adequate documentation of her work on the project.(3.11)**

- b. She has not previously worked on a voting machine project and did not disclose it to company(2.01)**

- c. She has registered as a voter in the state in which the machines will be used. (4.05)**

- d. She discovered significant number of defects in her design for the system that can cause failure of the project but did not report them. (2.06)**

Ethical principles: Examples

- **Principle 3: 3.11**
- **Principle 2: 2.01; 2.06**
- **Principle 4: 4.05**



Ethical principles: Examples

- You are working on the development of a software and your manager wants to ship the product in one week. You are aware that the securing function still does not work correctly.
- According to professional ethics, what are at least three actions you need to do?
- Answer:
 - at least he convinced them to delay the delivery of the software,
 - not complete the project unless his requirements are satisfied,
 - approve any software that meet specification only.
- The software engineering code of ethics 1.03, which states that a software engineer: “*Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.*”

Ethical dilemmas

- ✧ Disagreement in principle with the policies of senior management
- ✧ Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system
- ✧ Participation in the development of military weapons systems or nuclear systems



Key points

- ✧ Software engineers have **responsibilities** to the engineering profession and society. They should not simply be concerned with technical issues.
- ✧ Professional societies publish **codes of conduct** which set out the **standards of behaviour** expected of their members.



Software Engineering – CSC 342

Chapter 2

Software Processes



Software Engineering

Software Processes

Objectives

- To introduce software process models
- To describe three generic process models and when they may be used
- To describe outline process models for:
 - requirements engineering
 - software development
 - testing and evolution
- To introduce CASE technology to support software process activities

Topics covered

- Software process models
- Process iteration
- Process activities
- Computer-aided software engineering



1. Introduction

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

1. Introduction

- A software process model:
 - is an abstract representation of a process
 - it presents a description of a process from some particular perspective.

- Many organization still rely on ad-hoc processes
 - no use of sw processes methods
 - no use of best practice in sw industry

2. Generic software process models

The waterfall model

- Separate and distinct phases of specification and development:
Requirements, design, implementation, testing, ...
- No evolution process, only development
- Widely used & practical
- Recommended when requirements *are well known and stable at start*

Evolutionary development

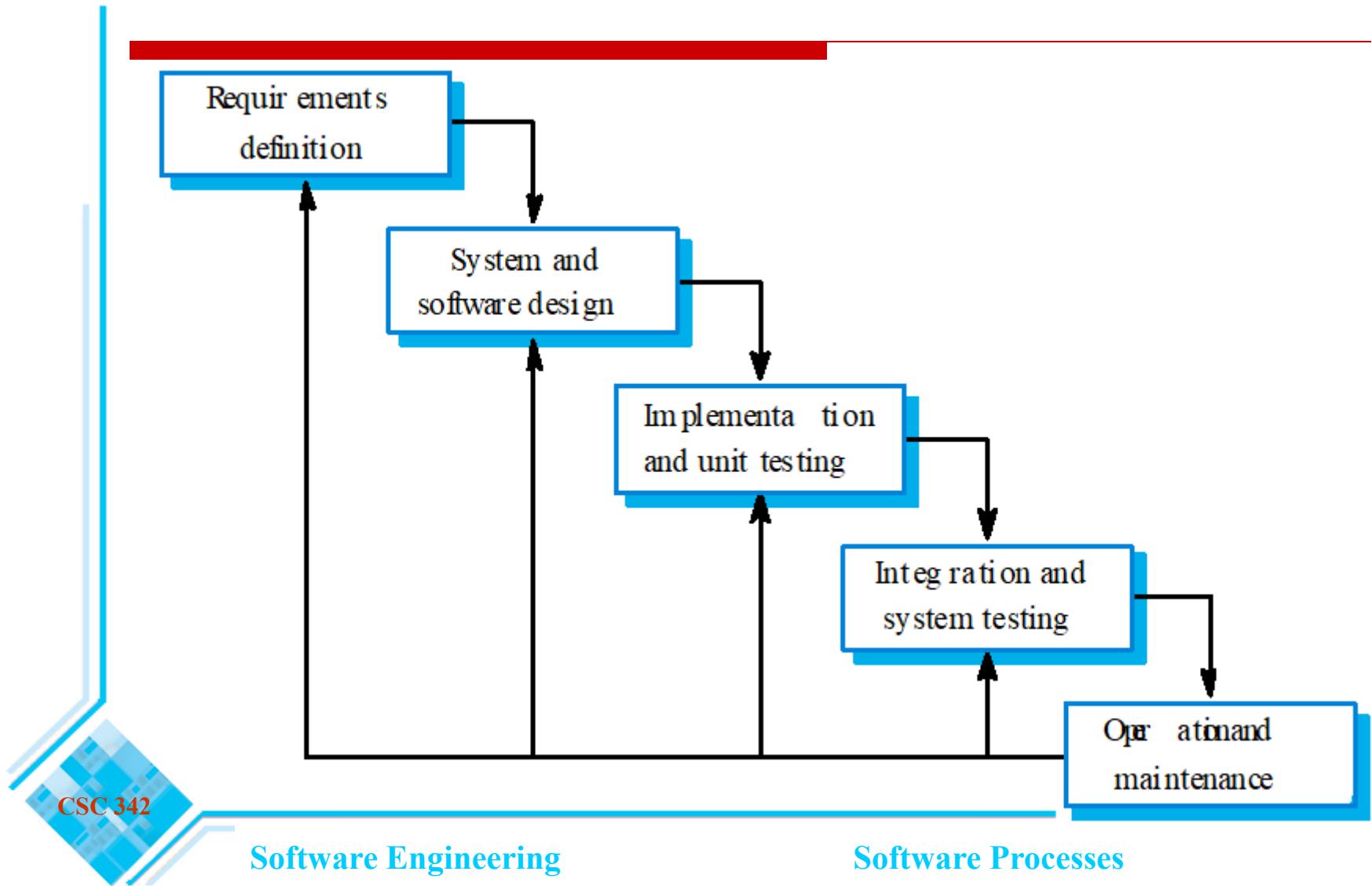
- Specification and development are interleaved
- Develop rapidly & refine with client
- Widely used & practical
- Recommended when requirements *are not well known at start*

6

Generic software process models

- Reuse-based (Component-based) development
 - The system is *assembled* from existing components
 - » Components already developed within the organization
 - » COTS “Commercial of the shelf” components
 - Integrating rather than developing
 - Allows rapid development
 - Gaining more place
 - Future trend

Waterfall model



Waterfall model

- The classic way of looking at S.E. that accounts for the importance of requirements, design and quality assurance.
- The model suggests that software engineers should work in a series of stages.
- Before completing each stage, they should perform quality assurance (verification and validation).
- The waterfall model also recognizes, to a limited extent, that you sometimes have to step back to earlier stages.

Limitations of the waterfall model

- The model implies that you should attempt to complete a given stage before moving on to the next stage
 - Does not account for the fact that requirements constantly change.
 - It also means that customers can not use anything until the entire system is complete.
- The model makes no allowances for prototyping.
- It implies that you can get the requirements right by simply writing them down and reviewing them.
- The model implies that once the product is finished, everything else is maintenance.

Limitations of the waterfall model

- Drawback: the difficulty of accommodating change after the process is underway
- Inflexible partitioning of the project into distinct stages
- Inflexible: to respond to dynamic business environment leading to requirements changes
- Appropriate when the requirements are *well-understood and stable*

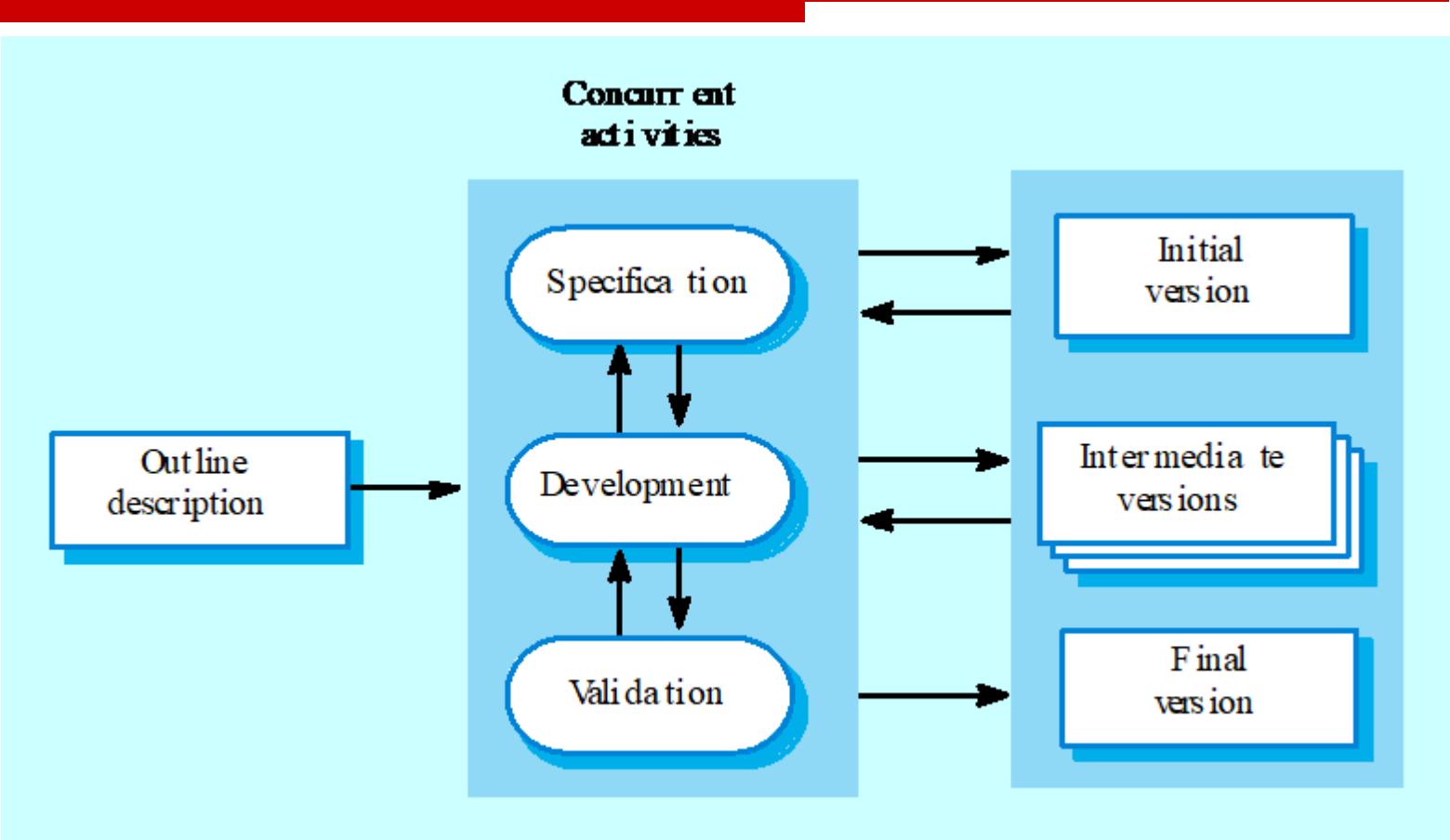
Evolutionary development

- Develop an initial implementation prototype
- Client test drive ... ➔ feed back
- Refine prototype
- 2 types of Evolutionary development

Exploratory development

Throw-away prototyping

Evolutionary development



Evolutionary development

2 types of Evolutionary development

- Exploratory development
 - Objective is to work with customers, explore their requirements and to evolve a final system from an initial outline specification.
 - Should start with *well-understood* requirements and add new features as proposed by the customer.
- Throw-away prototyping
 - Objective is to understand the system requirements and outline a better definition of requirements.
 - Should start with poorly understood requirements to clarify what is really needed.

Evolutionary development

➤ Problems

- Lack of process visibility at client management level (less regular reports/documentation ... the system is changing continuously)
- Systems are often poorly structured
- Special skills (e.g. in languages/tools for rapid prototyping) may be required

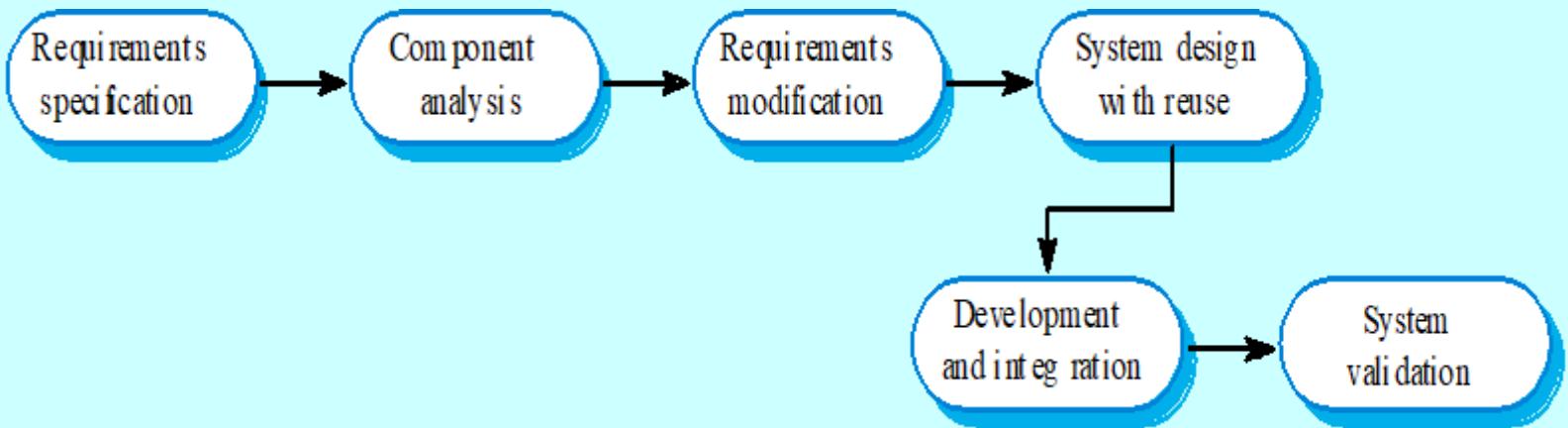
➤ Applicability

- 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

- Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- Process stages
 - Component analysis;
 - Requirements modification;
 - System design with reuse;
 - Development and integration.
- This approach is becoming increasingly used as component standards have emerged.

Reuse-oriented development



3. Process iteration

- Change is inevitable in all large sw projects. As new technologies, designs and implementation change.
- The process activities are regularly repeated as the system is reworked in response to change requests.
- Iteration can be applied to any of the generic process models.
- Iterative process models present the sw as a cycle of activities.
- The advantage of this approach is that it avoids premature commitments to a specification or design.

Process iteration

- Two (related) approaches:
 - Incremental delivery: the software specification, design and implementation are broken into a series of increments that are each developed in turn.
 - Spiral development: the development of the system spirals outwards from an initial outline through to the final developed system.

Incremental delivery

Software process models - Comparison

- **Waterfall model**

Requirements should be well defined **at start and committed to**

- **Evolutionary model**

Requirements & design decisions may be delayed: Poor structure
difficult to maintain

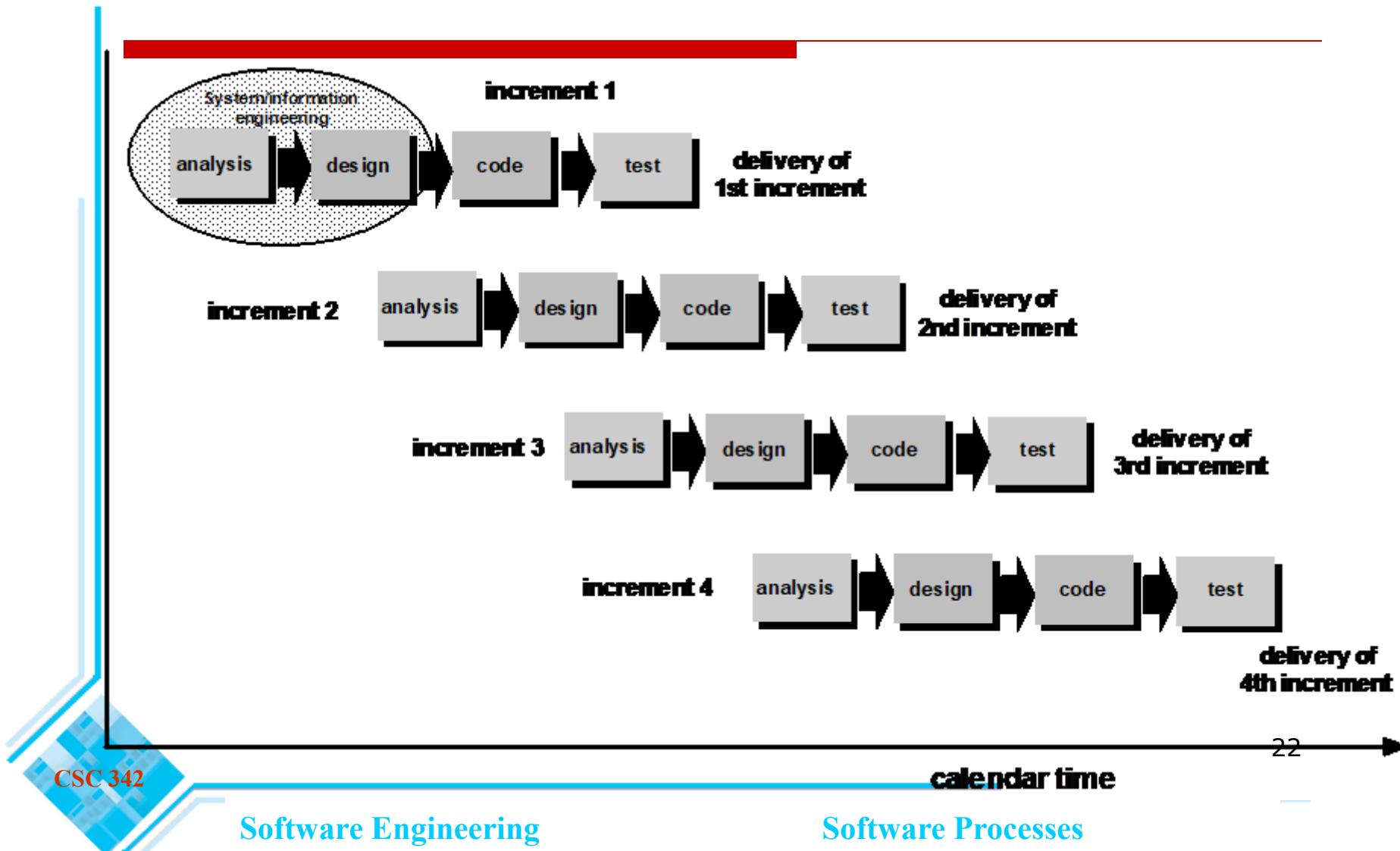
- **Incremental development**

- Is an in-between approach that combines the advantages of these models.
- Incremental ***prioritized delivery of modules***
- ***Hybrid*** of Waterfall and Evolutionary

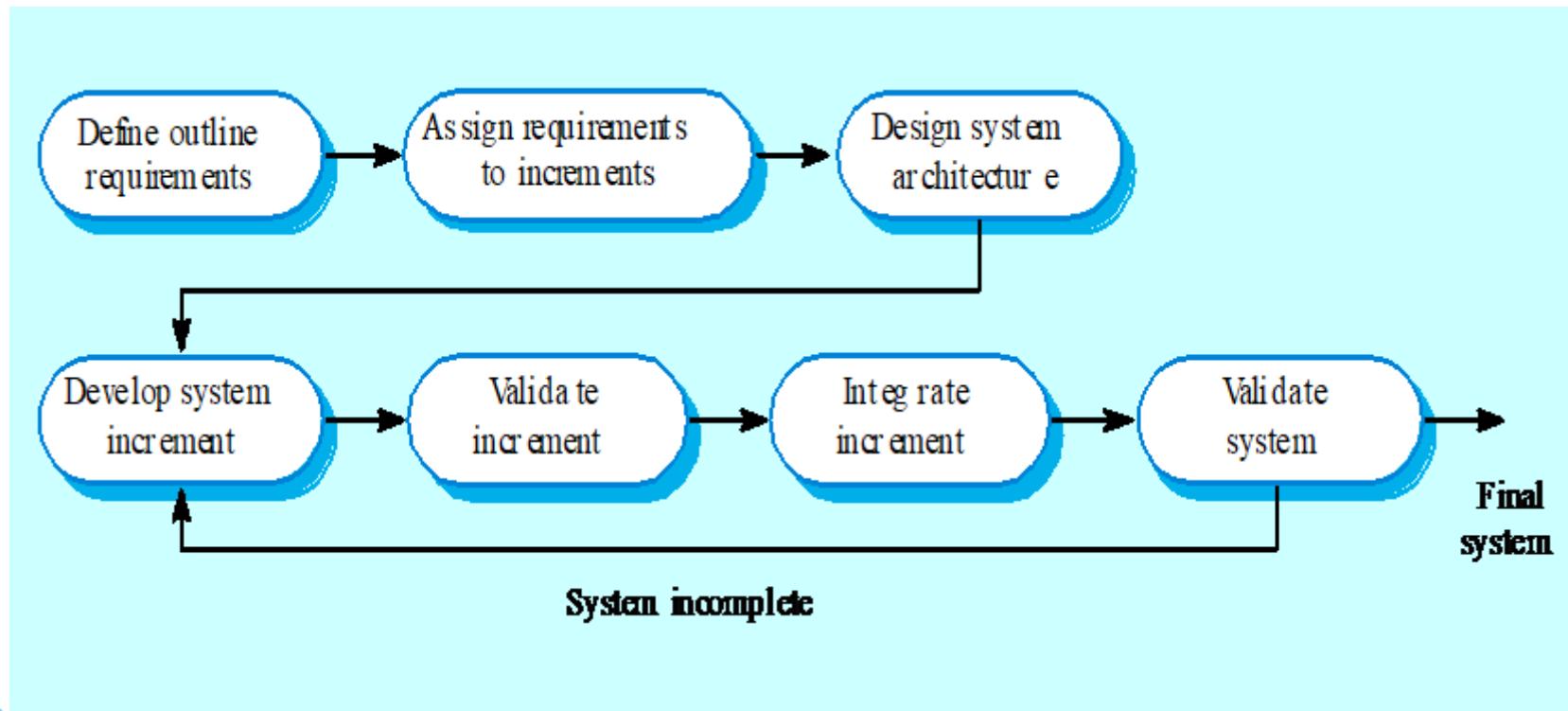
Incremental delivery

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

Incremental delivery



Incremental development



Incremental development advantages

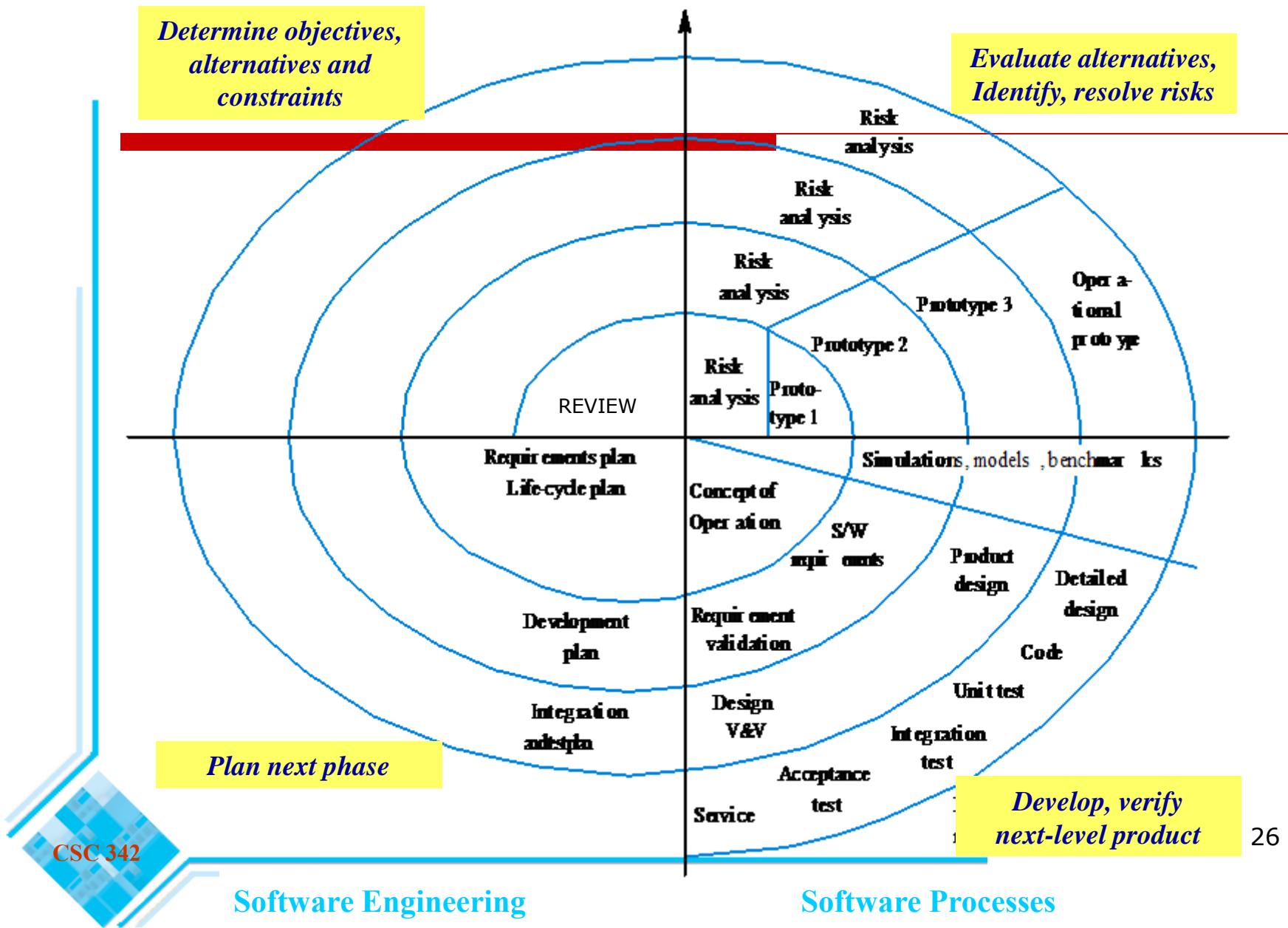
- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.

Spiral development

- Best features of waterfall & prototyping models
 - + Risk Analysis (missed in other models)
- Process is represented as a spiral rather than as a sequence of activities with backtracking.
- Each loop in the spiral represents a phase in the process.
- Risks are explicitly assessed and resolved throughout the process.

Informally, risk simply means something that can go wrong.

Spiral model of the software process



Spiral development

- It explicitly embraces prototyping and an *iterative* approach to software development.
 - Start by developing a small prototype.
 - Followed by a mini-waterfall process, primarily to gather requirements.
 - Then, the first prototype is reviewed.
 - In subsequent loops, the project team performs further requirements, design, implementation and review.
 - The first thing to do before embarking on each new loop is risk analysis.
 - Maintenance is simply a type of on-going development.

Spiral model: 4 sectors

Each loop in the spiral is split into four sectors:

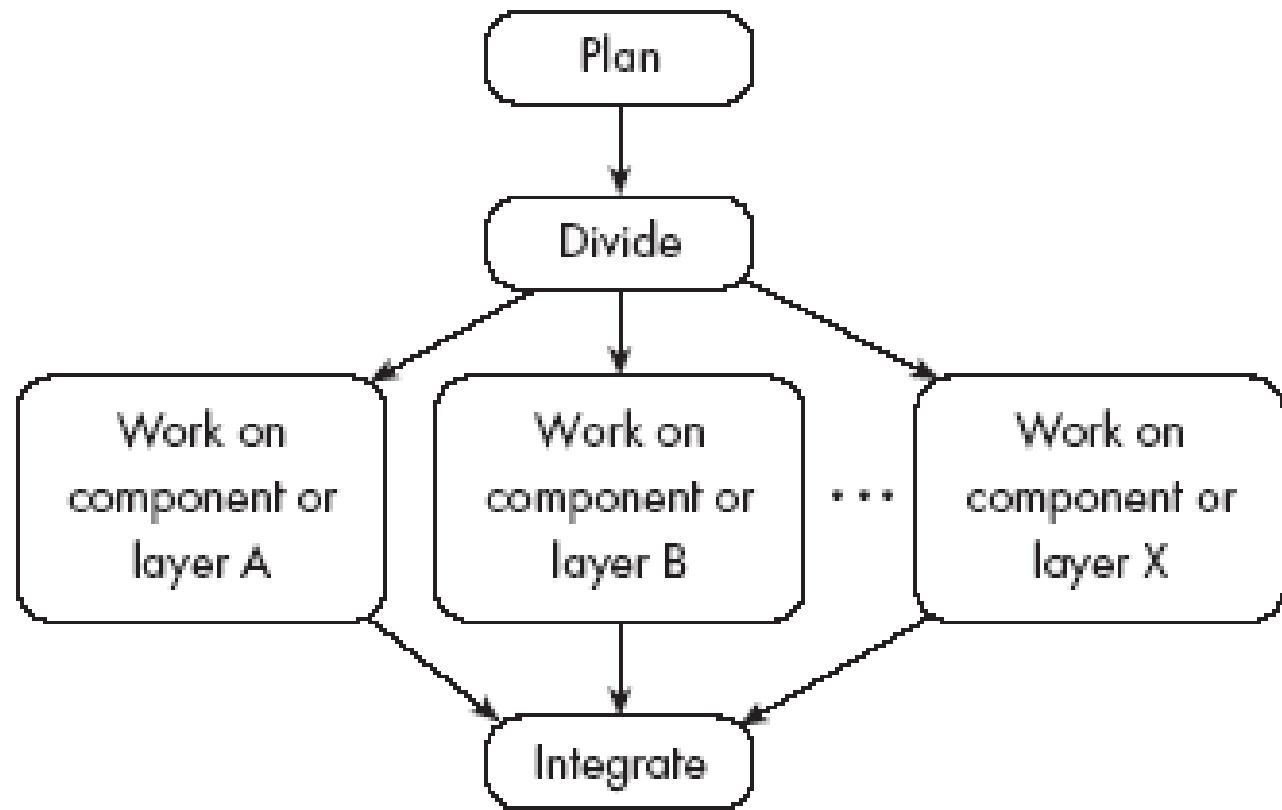
- Objective setting
 - Specific objectives for the phase are identified.
- Risk assessment and reduction
 - Risks are assessed and activities put in place to reduce the key risks. For example if there is a risk that the requirement. are inappropriate, a prototype system may be developed.
- Development and validation
 - A development model for the system is chosen which can be any of the generic models.
- Planning
 - Review with client
 - Plan next phase of the spiral if further loop is needed

Spiral model usage

- Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development.

- In practice, however, the model is rarely used as published for practical software development.

The concurrent engineering model



The concurrent engineering model

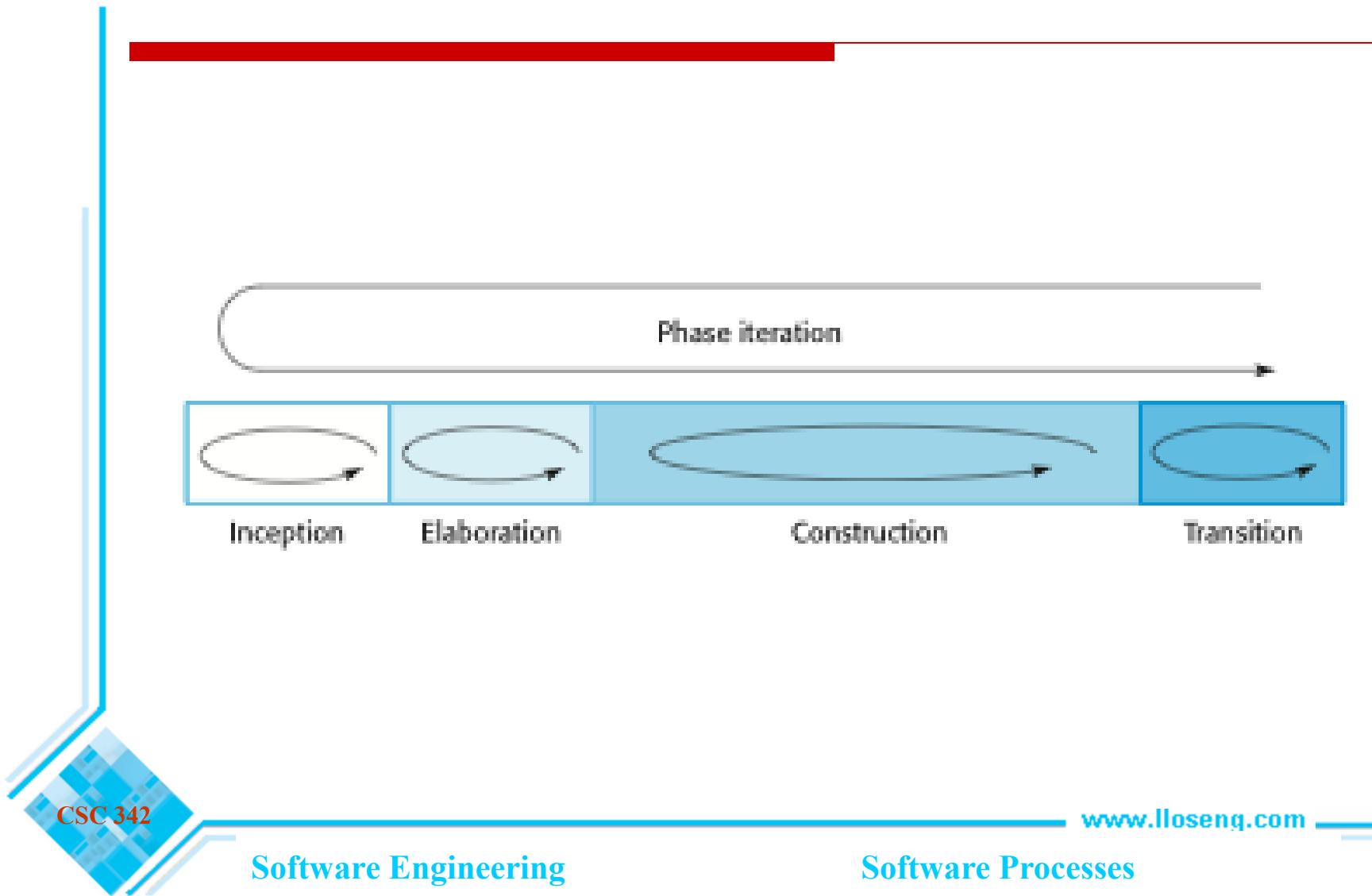
- It explicitly accounts for the divide and conquer principle.
 - Each team works on its own component, typically following a spiral or evolutionary approach.
 - There has to be some initial planning, and periodic integration.



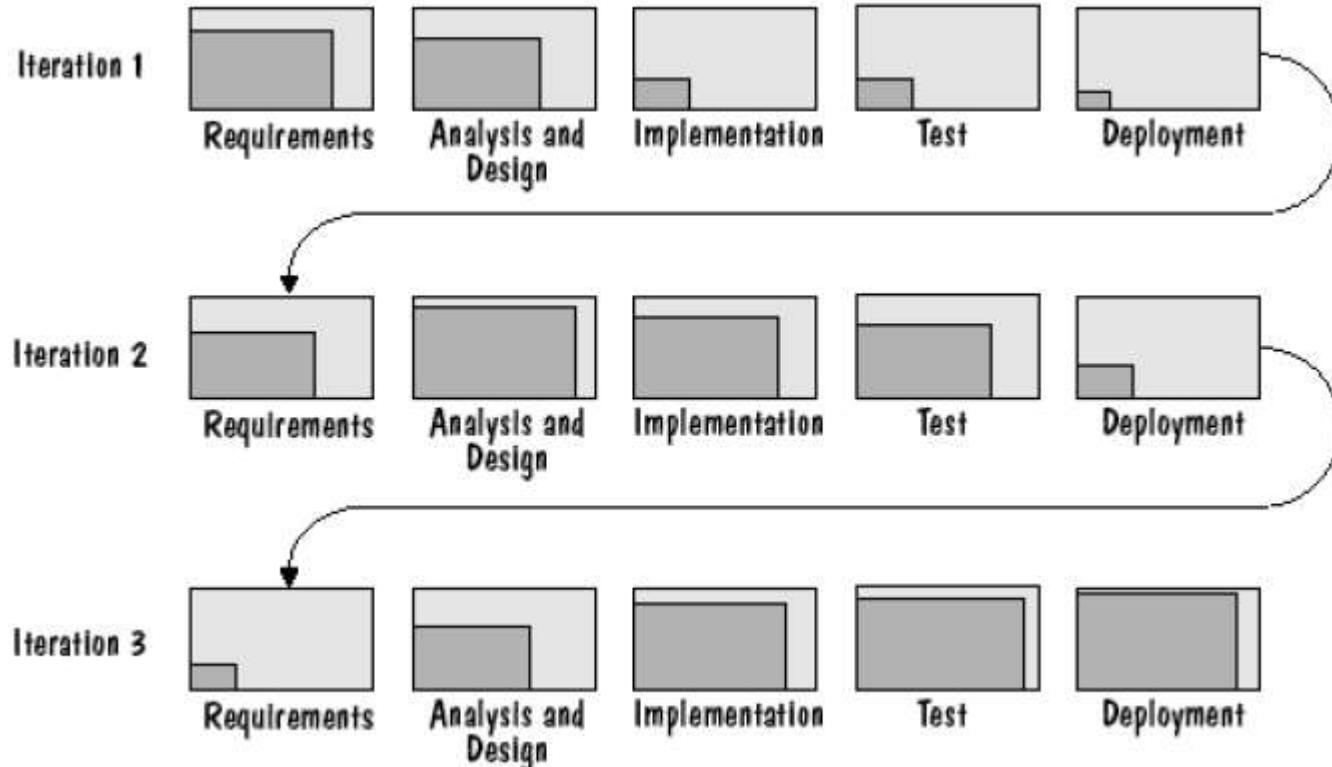
The Rational Unified Process

- A modern generic process derived from the work on the UML and associated process.
- **RUP is like an online mentor that provides guidelines, templates, and examples for all aspects and stages of program development.**
- Brings together aspects of the 3 generic process models discussed previously.
- Normally described from 3 perspectives
 - A dynamic perspective that shows phases over time;
 - A static perspective that shows process activities;
 - A practice perspective that suggests good practice.

Phases in the Rational Unified Process



RUP iteration



RUP phases

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

RUP iteration

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

RUP disciplines

1. Business modeling discipline

- Establish a better understanding of the structure (roles and responsibilities), the dynamics and the goals of the target organization (the client)

2. Requirements discipline

- Elicit requests and transform them into a set of requirements.

RUP disciplines

3. Analysis and design discipline

- How the system will:
 - Perform the tasks and functions specified in the use-case descriptions.
 - Be easy to change when requirements change.

4. Implementation discipline

- Software architecture.
- Implement classes and objects
- Test the developed components as units.
- Integrate components into an executable system.

RUP disciplines

5. Test discipline

- Verify the proper integration of all components of the software.
- Verify that all requirements have been correctly implemented.
- Ensure that all the defects are fixed, retested and closed.

6. Deployment discipline

- External releases of the software
- Packaging the software
- Distributing the software
- Installing the software
- Providing help and assistance to users.

RUP good practice

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

RUP good practice

- Visually model software
 - Use graphical UML models to present static and dynamic views of the software.
- Verify software quality
 - Ensure that the software meet's organizational quality standards.
- Control changes to software
 - Manage software changes using a change management system and configuration management tools.

Key points

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

Choosing a process model

- From the waterfall model:
 - Incorporate the notion of stages.
- From the Incremental delivery model:
 - Incorporate the notion of doing some initial high-level analysis, and then dividing the project into releases.
- From the spiral model:
 - Incorporate prototyping and risk analysis.
- From the evolutionary model:
 - Incorporate the notion of varying amounts of time and work, with overlapping releases.
- From concurrent engineering:
 - Incorporate the notion of breaking the system down into components and developing them in parallel.

4. Process Activities

- Software specification
- Software design and implementation
- Software validation
- Software evolution



Software specification

Requirements engineering process

The process of establishing

- What services are required (Functional requirements) for the system
- Identifying the constraints on the system's operation and development (Non-functional requirements)

Requirements engineering process

1. Feasibility study: *An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies.*

- Alternatives & Quick cost/benefit analysis
- Feasibility: Technical, Financial, Human, Time schedule
- **Deliverables:** Feasibility report

2. Requirements elicitation and analysis: Facts finding

- Interviews, JAD “Joint Application Development”, Questionnaires, Document inspection, Observation
- **Deliverables:** System models (Diagrams)

Software specification

Requirements engineering process

Requirements engineering process

3. Requirements specification: *the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements.*

- User level: abstract specification
- System level: detailed specification
- **Deliverables:** User and system requirements

4. Requirements validation: *this activity checks the requirements for::*

- Completeness
- Consistency
- Realism
- **Deliverables:** Updated requirements

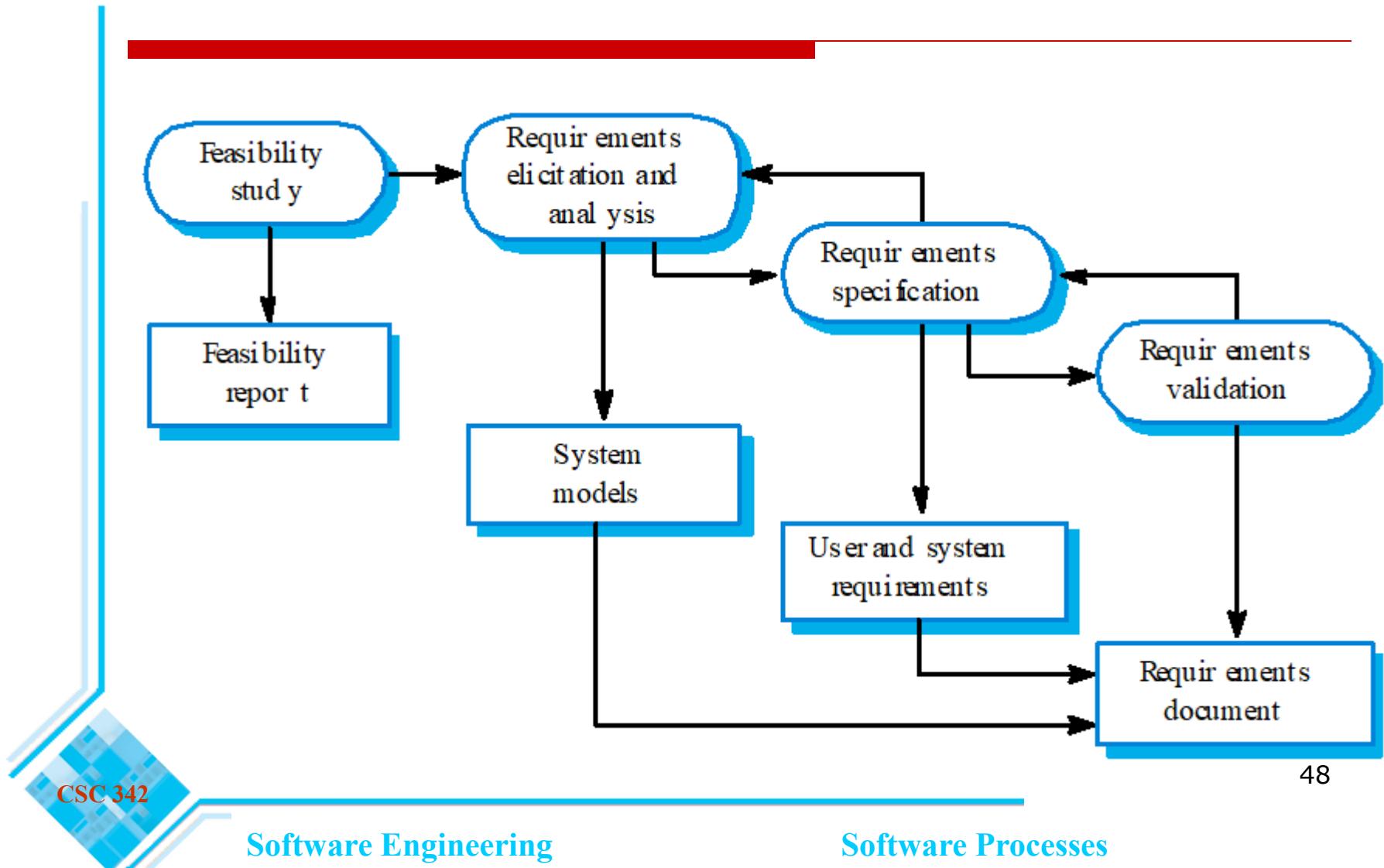
Global Deliverables of the Requirements Engineering Process :
System Requirements Specification document

Completeness

- Let us consider the following requirement statement:
The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security VDU and deposited in the login file when an operator logs into the system.
- This is an example of ambiguous, incomplete requirement as it is not clear what is meant by “it” in the second sentence and what should be displayed on the VDU. Does it refer to the operator identity as a whole, his name, or his password?

Software specification

Requirements engineering process



Software design and implementation

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

Design process activities

- **Architectural design**
- **Abstract specification**
- **Interface design**
- **Component design**
- **Data structure design**
- **Algorithm design**

Design Process Activities

1. Architectural design

- Subsystems/relationships, block diagram
- Deliverables: System architecture

2. Abstract specification for each subsystem

- Deliverables: For each sub-system, an abstract specification of its services and constraints under which it must operate is produced

3. System/subsystems Interface design

- With other subsystems of the sys
- With external systems (Bank, GOSI, ...) *General Organization for Social Insurance*
- Deliverables: Interface specs for each subsystem in relation to other subsystems or external systems

Design Process Activities

4. Component design

- Services are allocated to components
- Components interfaces are designed
 - » Interfaces with other components of the system
 - » Interfaces with external systems
 - » GUI
 - » Input
 - » Output
- Deliverables: Component specs

Design Process Activities

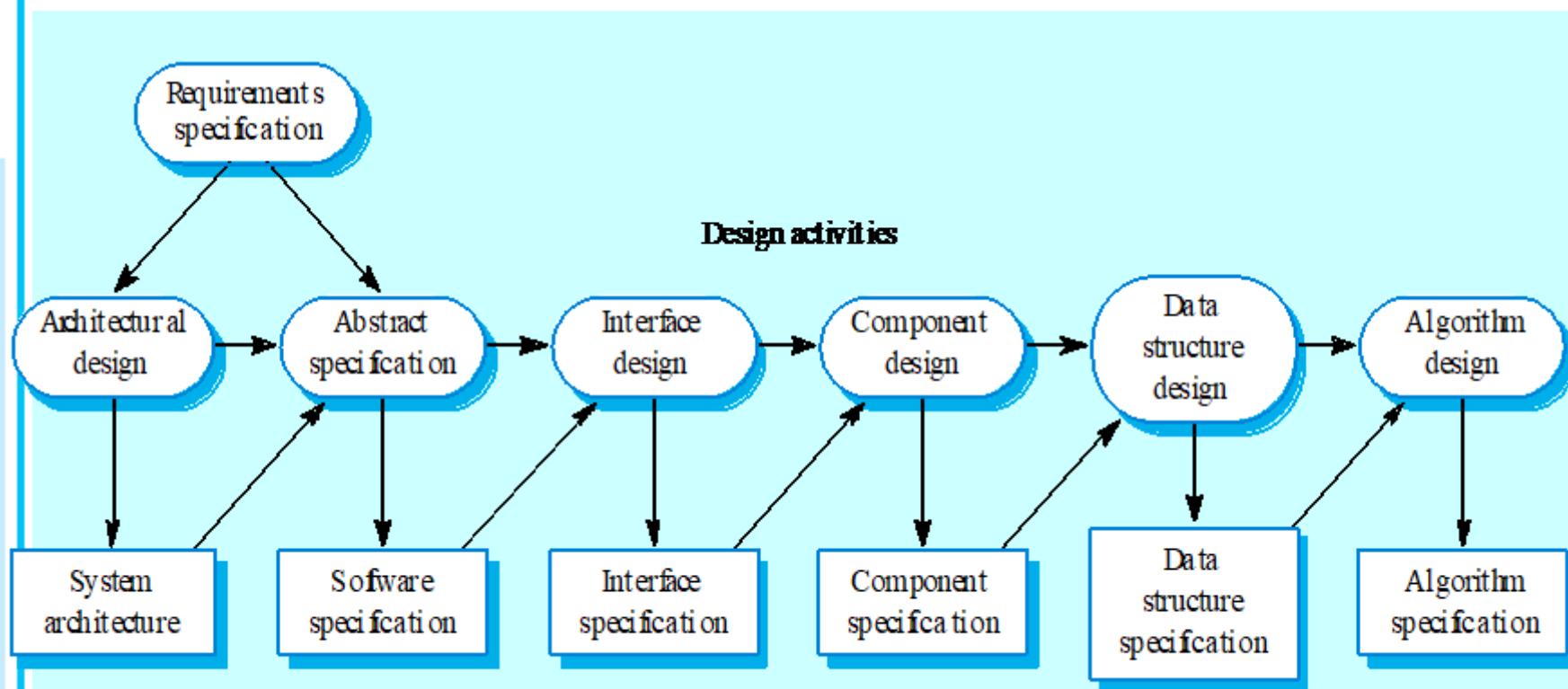
5. Data structure (Database) design

- Detailed design of data structure to be implemented (design **or** implementation activity)
- **Deliverables:** Data structure specs

6. Algorithm design

- Detailed design of algorithm for services to be implemented (design **or** implementation activity)
- **Deliverables:** Algorithm specs

The software design process



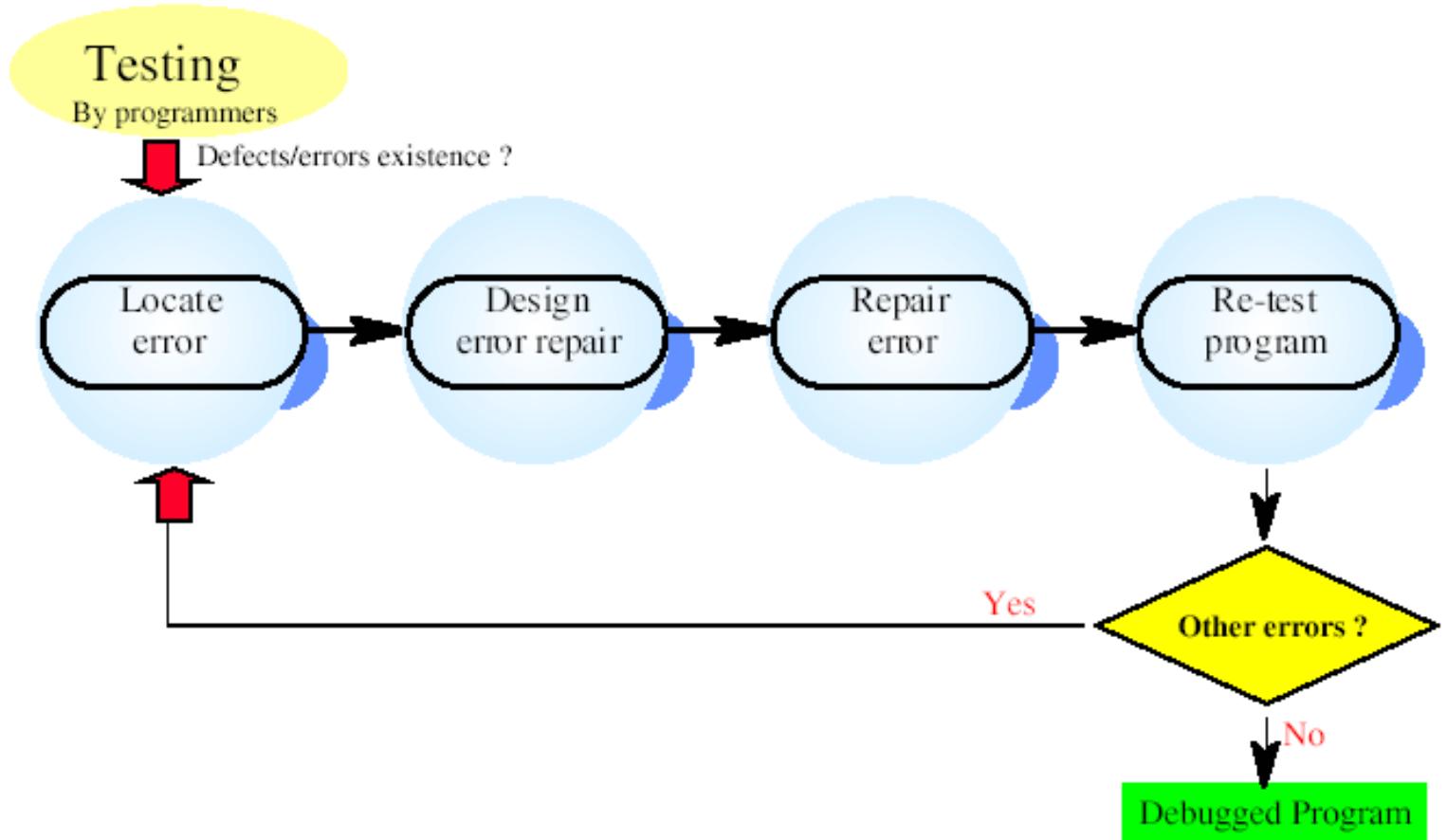
Structured methods

- Systematic approaches to developing a software design.
- Structured methods: Set of notations & guidelines for s/w design
 - Graphical methods
 - CASE tools to generate a skeleton program from a design.
- The design is usually documented as a set of graphical models.
- Possible models
 - Object model that shows the object classes used in the system and their dependencies.
 - Sequence model that shows how objects in the system interact when the system is executing.
 - State transition model that shows system states.
 - Structural model where the system components and their aggregations are documented.
 - Data-flow model.

Programming and debugging

- Translating a design into a program and removing errors from that program.
- Programming is a personal activity - there is no generic programming process.
- Programmers carry out some program testing to discover faults in the program and remove these faults in the debugging process.
- Debugging process is part of both sw development (as above by programmers) but sw testing is done by testers

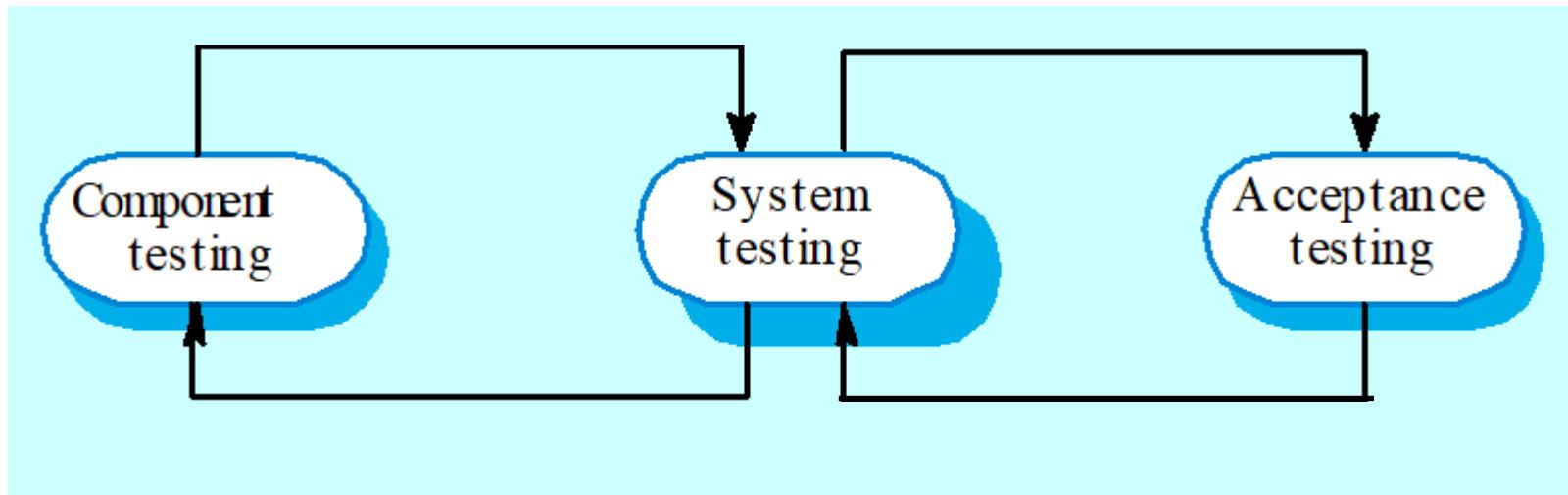
The debugging process



Software validation/verification

- Validation: Are we building the right product (satisfying client requirements)
- Verification: Are we building the product right (standards of the development process. Does SW conform to its specification)
- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking and review processes and system testing.
- System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.

The testing process



The testing process

- Testing = Verification + Validation
 - Verification: **Document-based testing**, Static Testing (no run)
 - Validation: Dynamic Testing (Run code)

Verification: (standards of the development process)

- IEEE/ANSI: Determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- Verification: **document-based testing, static testing**
- Evaluating, reviewing, inspecting, and doing desk checks of the work produced during development such as:
 - Requirements specifications: Do we have high quality requirements “clear, complete, measurable,..)
 - Design specifications
 - Code: Static analysis of code (code review) not dynamic execution of the code

Validation

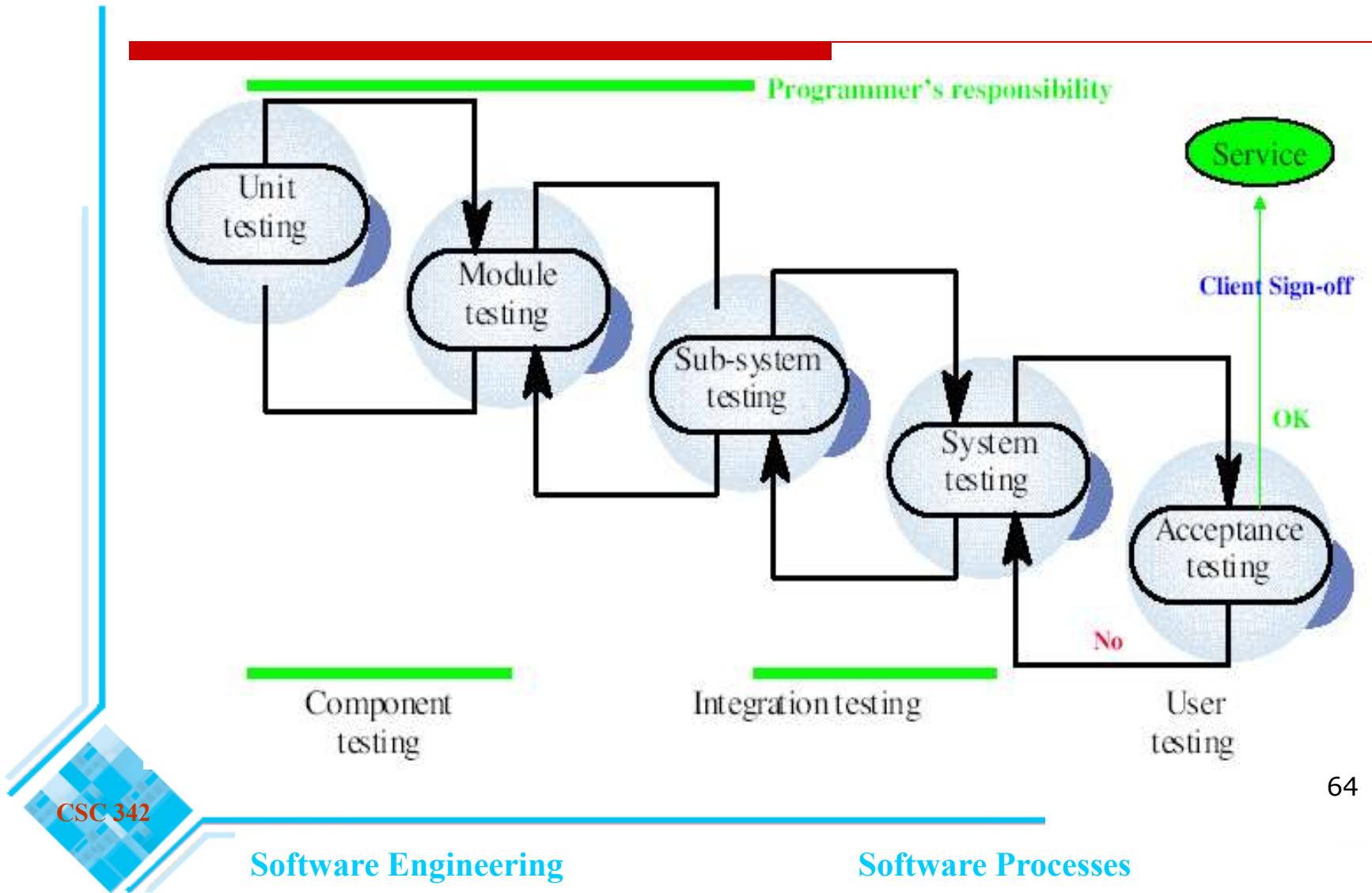
- IEEE/ANSI: System evaluation during or at the end of the development process to determine whether it satisfies the specified requirements.

- Validation:
 - **Run** actual software on a computer.
 - Computer-based testing, Dynamic testing

Validation/Verification

- V&V are complementary
- Each provides filters to catch different kinds of problems

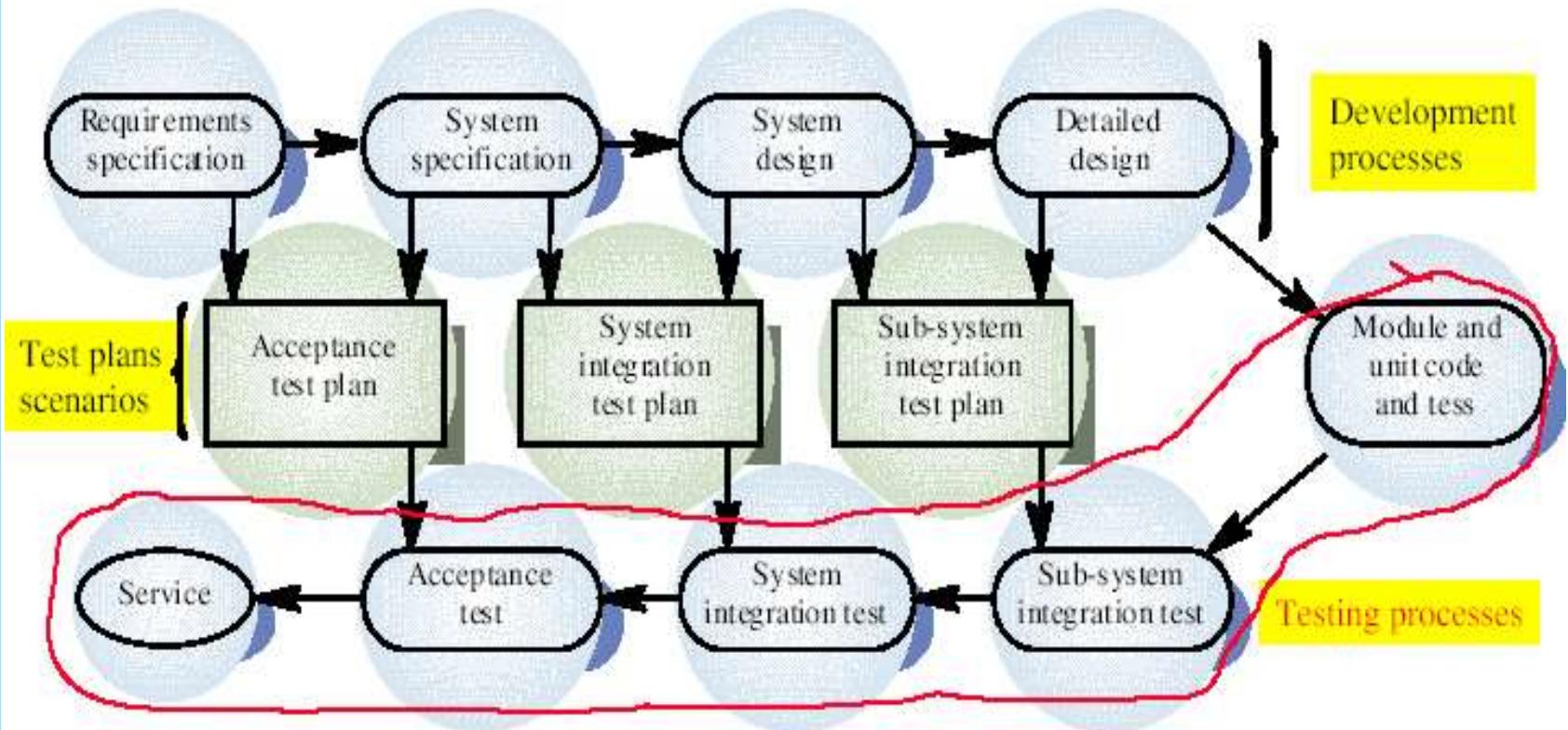
The System Testing Process



Testing stages

- Unit testing
 - Individual components are tested
- Module testing
 - Related collections of dependent components are tested
- Sub-system testing
 - Modules are integrated into sub-systems and tested. The focus here should be on **interface testing**
- System testing/Integration Testing
 - Testing functionality of the integrated system as a whole
 - Testing of **emergent properties**
- Acceptance testing
 - Testing with customer data (real, not simulated data)to check that the is acceptable

Testing phases

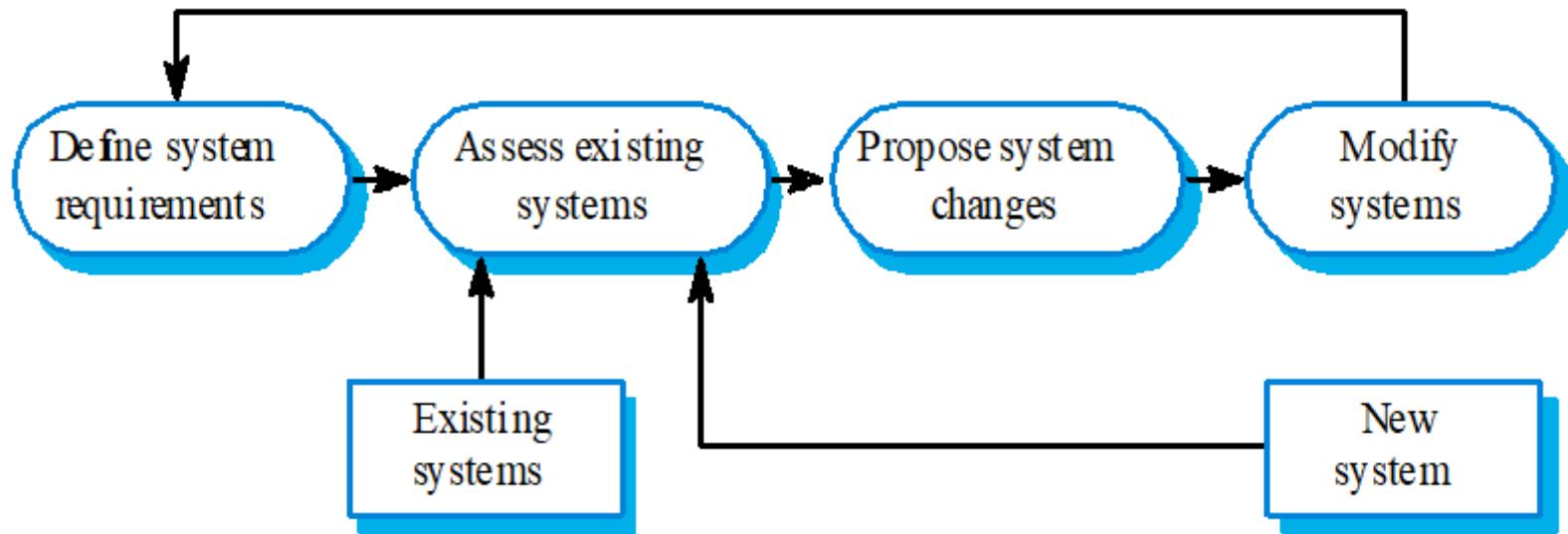


Test plans (scenarios) are the link between development & testing

Software Evolution

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

System Evolution



5. CASE: Computer-Aided Software Engineering

- Computer-aided software engineering (CASE) is software to support software development and evolution processes.
- Activity automation
 - Graphical editors for system model development;
 - Data dictionary to manage design entities;
 - Graphical UI builder for user interface construction;
 - Debuggers to support program fault finding;
 - Automated translators to generate new versions of a program.

CASE technology

- Case technology has led to significant improvements in the software process. However, these are not the order of magnitude improvements that were once predicted
 - Software engineering requires creative thought - this is not readily automated;
 - Software engineering is a team activity and, for large projects, much time is spent in team interactions. CASE technology does not really support these.

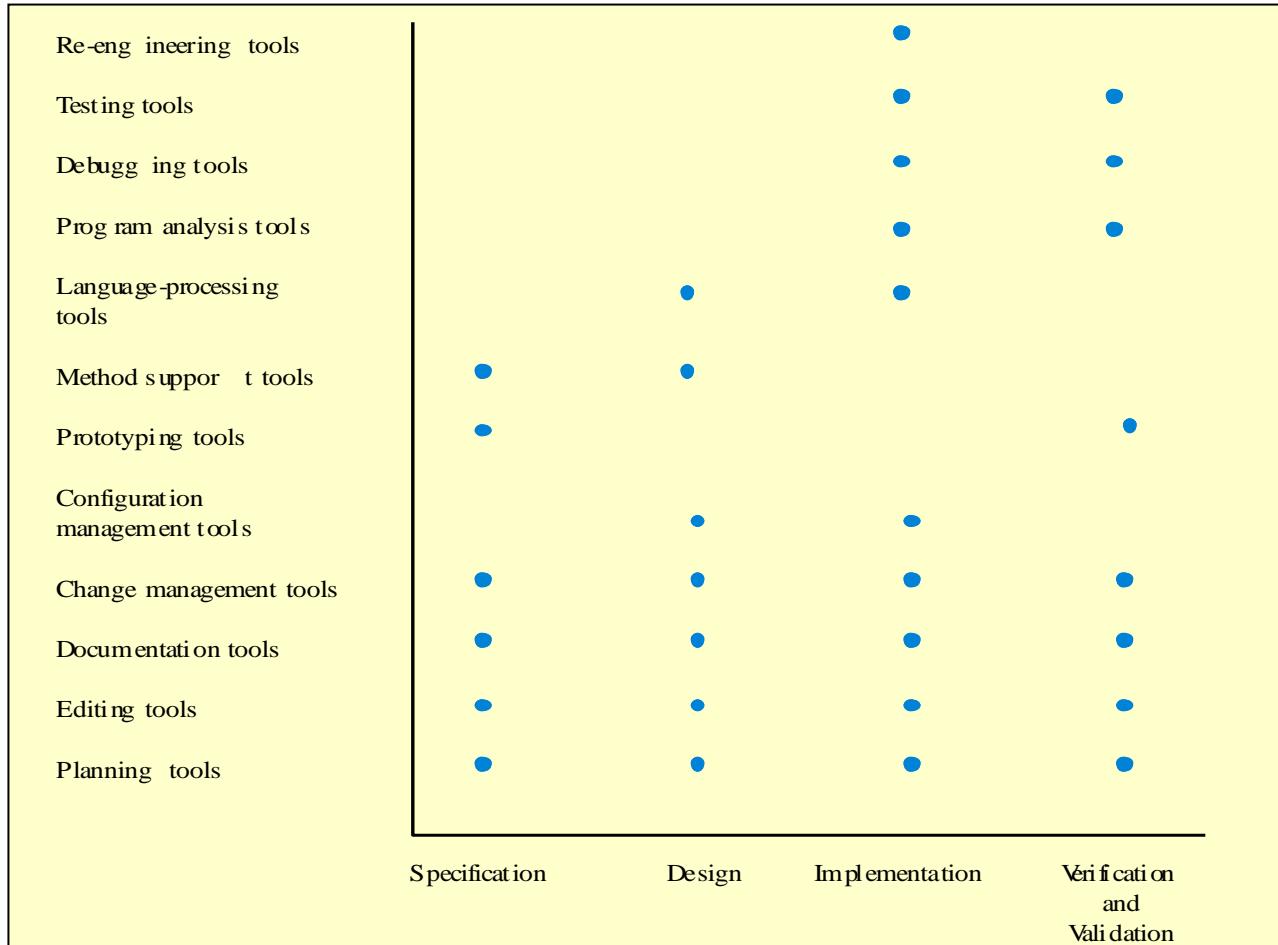
CASE classification

- Classification helps us understand the different types of CASE tools and their support for process activities.
- Functional perspective
 - ✓ Tools are classified according to their specific function.
- Process perspective
 - ✓ Tools are classified according to process activities that are supported.
- Integration perspective
 - ✓ Tools are classified according to their organisation into integrated units.

Functional tool classification

Tool type	Examples
Planning tools	PERT tools, estimation tools, spreadsheets
Editing tools	Text editors, diagram editors, word processors
Change management tools	Requirements traceability tools, change control systems
Configuration management tools	Version management systems, system building tools
Prototyping tools	Very high-level languages, user interface generators
Method-support tools	Design editors, data dictionaries, code generators
Language-processing tools	Compilers, interpreters
Program analysis tools	Cross reference generators, static analysers, dynamic analysers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Re-engineering tools	Cross-reference systems, program re-structuring systems

Activity-based tool classification



CASE integration

□ Tools

- Support individual process tasks such as design consistency checking, text editing, etc.

□ Workbenches

- Support a process phase such as specification or design,
Normally include a number of integrated tools.

□ Environments

- Support all or a substantial part of an entire software process.
Normally include several integrated workbenches.

Key points

- Software processes are the activities involved in producing and evolving a software system.
- Software process models are abstract representations of these processes.
- General activities are specification, design and implementation, validation and evolution.
- Generic process models describe the organisation of software processes. Examples include the waterfall model, evolutionary development and component-based software engineering.
- Iterative process models describe the software process as a cycle of activities.

75

Key points

- Requirements engineering is the process of developing a software specification.
- Design and implementation processes transform the specification to an executable program.
- Validation involves checking that the system meets to its specification and user needs.
- Evolution is concerned with modifying the system after it is in use.
- The Rational Unified Process is a generic process model that separates activities from phases.
- CASE technology supports software process activities.

Software Engineering – CSC 342

Chapter 3

Software Requirements

King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI



Objectives

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

Outcomes

When you have read the chapter, you will

- Understand the concepts of user requirements
- Understand the concepts of system requirements
- Understand why these requirements should be written in different ways
- Understand the differences between functional and non-functional software requirements
- Understand how requirements may be organized in a software requirements document.

Requirements engineering

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

What is a requirement?

Requirements Analysis and Definition

- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- The **requirements analysis** and definition establish the system's services, constraints and goals by consultation with users. They are then **defined** in a manner that is understandable by both users and development staff.

Requirements define the function of the system FROM THE CLIENT'S VIEWPOINT.

Types of requirement

- **User requirements**

- Statements in natural language plus diagrams of what services the system is expected to provide and its operational constraints. Written for customers.

- **System requirements**

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

Types of requirement

User Requirement:

Let us assume that we have a word-processing system that does not have a spell checker. In order to be able to sell the product, it is determined that it must have a spell checker. Hence the business requirement could be stated as:

- *user will be able to correct spelling errors in a document efficiently.*
- Hence, the Spell checker will be included as a feature in the product.

Types of requirement

System Requirement:

- After documenting the user's perspective in the form of user requirements, the system's perspective: what is the functionality provided by the system and how will it help the user to accomplish these tasks. Viewed from this angle, the functional requirement for the same user requirement could be written as follows:
 - *The spell checker will find and highlight misspelled words. Right clicking a misspelled word, will then display a dialog box with suggested replacements. The user will be allowed to select from the list of suggested replacements. Upon selection it will replace the misspelled word with the selected word. It will also allow the user to make global replacements.*

Definitions and specifications

User requirement definition

Library System (LIBSYS) shall keep track of all data required by copyright licensing agencies.

System requirements specification

- On making a request for a document from LIBSYS, the requestor shall be presented with a form that records details of the user and the request made.
- LIBSYS request forms shall be stored on the system for five years from the date of the request.
- All LIBSYS request forms must be indexed by user, by the name of the material requested and by the supplier of the request.
- LIBSYS shall maintain a log of all requests that have been made to the system.
- For material where authors' lending rights apply, loan details shall be sent monthly to copyright agencies that have registered with LIBSYS

Functional and non-functional requirements

Software system requirements are often classified as functional requirements, non-functional requirements:

- Functional requirements
 - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- Non-functional requirements (Quality Requirements)
 - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

Functional requirements

- Describe functionality or system services.
- Depend on the type of software, expected users of the software and the general approach taken by the organisation when writing requirements.
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

Example: The LIBSYS system

Functional requirements may be expressed in a number of way.

Example: LIBSYS used by students and faculty to order books and documents from other libraries.

- A library system that 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.
- 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.

Non-functional requirements

- These define system properties and constraints e.g. reliability, response time and storage requirements.
- They may define constraints on the system such as the capabilities of I/O devices and the data representations used in system interfaces.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional classifications

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

User requirements

- Should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge.
- User requirements are defined using natural language, tables and diagrams as these can be understood by all users.

Problems with natural language

Various problems can arise when requirements are written in natural language sentences in a text document:

- Lack of clarity
 - Precision is difficult without making the document difficult to read.
- Requirements confusion
 - Functional and non-functional requirements tend to be mixed-up.
- Requirements amalgamation
 - Several different requirements may be expressed together as a single requirement.

Guidelines for writing requirements

To minimise misunderstandings when writing user requirements, It is recommended that you follow some simple guidelines:

- Invent a standard format and use it for all requirements.
- Use language in a consistent way. You should always distinguish between mandatory and desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.

System requirements

- More detailed specifications of system functions, services and constraints than user requirements.
- They are intended to be a basis for designing the system.
- They add detail and explain how the user requirements should be provided by the system.
- The system requirements should simply describe the external behaviour of the system and its operational constraints.
- They should not be concerned with how the system should be designed or implemented.



Requirements and design

- In principle, requirements should state **what** the system should do and the design should describe **how** it does this.
- In practice, requirements and design are inseparable
 - A system architecture may be designed to structure the requirements;
 - In most cases, systems must interoperate with other existing systems. These constrain the design, and these constraints impose requirements on the new system;

Problems with Natural Language (NL) specification

- Ambiguity
 - The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult.
- 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.

Because of these problems, requirements specification written in natural language are prone to misunderstandings.

Alternatives to NL specification

Notation	Description
Structured natural language	This approach depends on defining standard forms or templates to express the requirements specification.
Design description languages	This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system. This approach is not now widely used although it can be useful for interface specifications.
Graphical notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. An early example of such a graphical language was <u>SADT</u> . Now, <u>use-case descriptions</u> and <u>sequence diagrams</u> are commonly used.
Mathematical specifications	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

Structured language specifications

- The freedom of the requirements writer is limited by a predefined template for requirements.
- All requirements are written in a standard way.
- The terminology used in the description may be limited.
- The advantage is that the most of the expressiveness of natural language is maintained but a degree of uniformity is imposed on the specification.
- Structured language notations limit the terminology that can be used and use templates to specify system requirements.

Form-based specifications

- Definition of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Indication of other entities required.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

Graphical models

- Graphical models are most useful when you need to show how state changes or where you need to describe a sequence of actions.
- Different graphical models are explained in next chapters.

The requirements document

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

Key points

- 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. User requirements should be written using natural language, tables and diagrams.

Key points

- 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.



Software Project Management

King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

Objectives

The main objectives of this chapter are:

- ❑ To explain the main tasks undertaken by project managers
- ❑ To introduce software project management and to describe its distinctive characteristics
- ❑ To discuss project planning and the planning process
- ❑ To explain the responsibilities of software managers
- ❑ To introduce the different types of Project
 - Plans Management activities
 - Project planning
 - Project scheduling

What is Software Engineering?

Developing software having:

- High **quality**
- Within **budget**
- On **schedule** (time)
- Satisfying client's **requirements**

Project Attributes

A project:

- ❑ Has a unique purpose.
- ❑ Is temporary.
- ❑ Is developed using progressive elaboration.
- ❑ Requires resources, often from various areas.
- ❑ Should have a primary customer or sponsor.
 - ❑ The **project sponsor** usually provides the direction and funding for the project.
- ❑ Involves uncertainty.

What is a Project Management ?

Project management encompasses all the activities needed to plan and execute a project:

- Deciding what needs to be done
- Estimating costs
- Ensuring there are suitable people to undertake the project
- Defining responsibilities
- Scheduling
- Making arrangements for the work

What is a Project Manager ?

- Directing
- Being a technical leader
- Reviewing and approving decisions made by others
- Building morale and supporting staff
- Monitoring and controlling
- Co-ordinating the work with managers of other projects
- Reporting
- Continually striving to improve the process

Failure Statistics of SW Projects

Success

- ✓ On-time,
- ✓ On-Budget,
- ✓ And scope-coverage (with Most of the Features & Functions)

Failed

- ✓ Over-budget,
- ✓ Over-time,
- ✓ And/or with less scope (Fewer Features & Functions)

Why Projects Fail?

- an unrealistic **deadline** is established
- changing customer **requirements**
- an honest underestimate of **effort**
- predictable and/or unpredictable **risks**
- **Technical difficulties**
- **Miscommunication** among project staff
- failure in project **management**.

Software project management

S/W PM is an essential part of SE.

Why S/W Project Management ?

- Because software development is always subject to
 - **Budget** and
 - **Schedule** constraints
 - **Quality** constraints
 - Satisfying all **requirements** that are set by the organization developing the software
 - Minimize **risk** of failure

Software project management

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

S/W Management Activities

- ❖ S/W manager responsibilities include:

- ❑ **Proposal writing:** Objectives, methodology, deliverables, cost & schedule estimates
- ❑ **Project planning and scheduling:** Goals, activities, resources, milestones
- ❑ **Project costing:** Resources needed for activities
- ❑ **Project monitoring and reviews:** Track actual versus planned cost and time
- ❑ **Personnel selection and evaluation**
- ❑ **Report writing and presentations**

Project Management Concerns

- product quality?
- risk assessment?
- software measurement?
- cost estimation?
- project scheduling?
- customer communication?
- staffing?
- other resources?
- project monitoring?

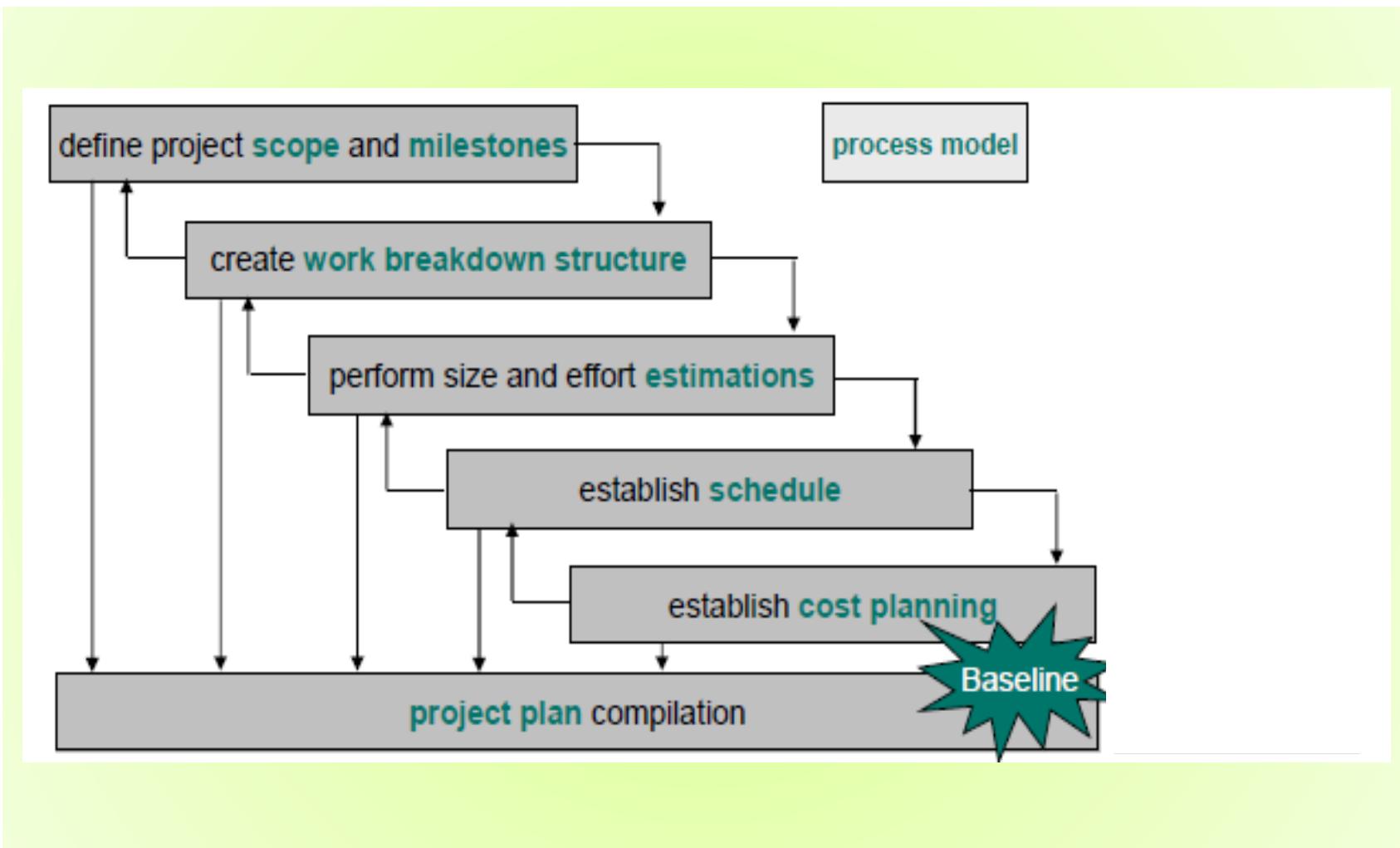
Project Planning

- ❑ Main software project plan that is concerned with **schedule and budget**

- ❑ 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 the **main software project plan** that is concerned with schedule and budget.

Project Planning Process



The project plan

The project plan sets out:

- The work breakdown activities/tasks (**What**);
- The resources available to the project (**Who**);
- A schedule for the work (**When**).

The project plan Structure

1. Introduction

- + Project objectives –constraints (budget, time, etc.)

2. Project organization

- + People involved, roles

3. Risk analysis

- + Projects risks, Risk reduction strategies

4. Resource requirements: Hardware and software

5. Work breakdown

- + Activities, milestones, deliverables

6. Project schedule (3W: What activity, when, who)

- + Activities dependencies, activities time, allocate people to activities

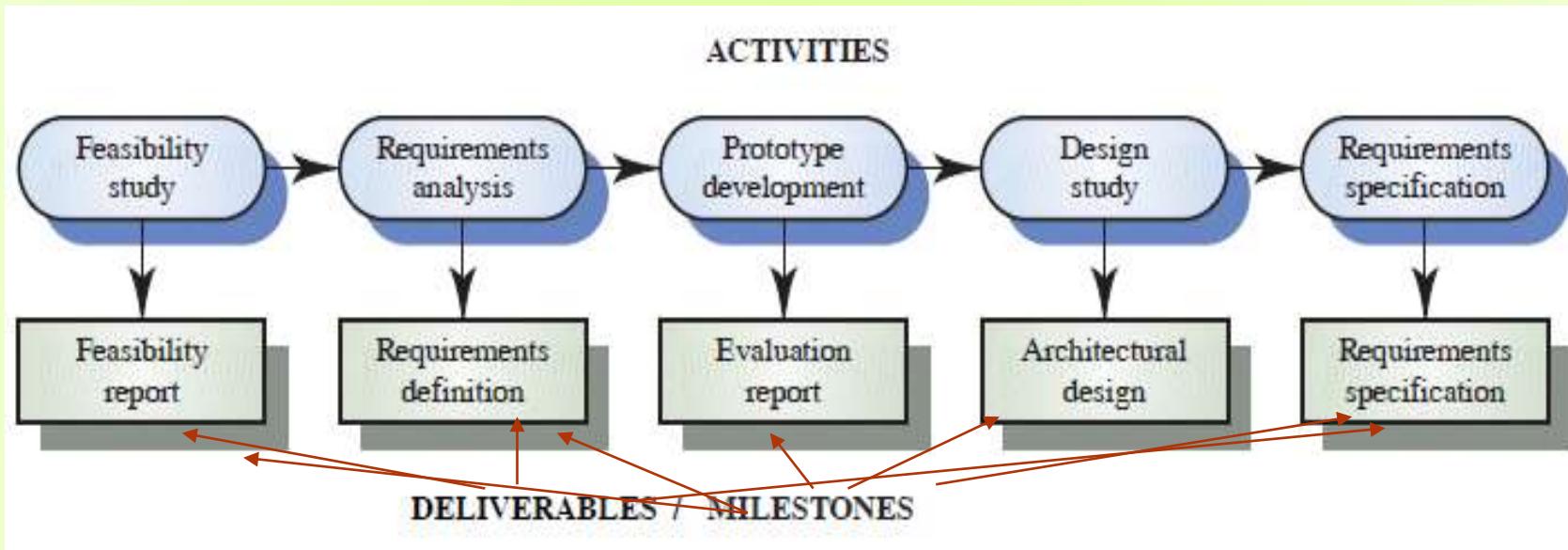
7. Monitoring and reporting mechanisms

- + What management reports and when
- + Monitoring mechanism used
- + Revise plan, update schedule

The project plan Structure

- Activities in a project should be organized to produce **tangible outputs** for management to judge progress
- **Milestones**
 - ✚ Check point based on :
 - Time
 - Budget
 - Deliverable
 - ✚ End-point of logical stage (activity) in the project
 - ✚ **At each milestone there should be a formal output** (report) presented to management
 - Management needs documentation & information to judge project progress
- **Deliverables**
 - ✚ Are project results delivered to customers
 - ✚ Deliverables are usually milestones but milestones need not be deliverables

Milestones Example: Requirements Engineering process (prototyping)

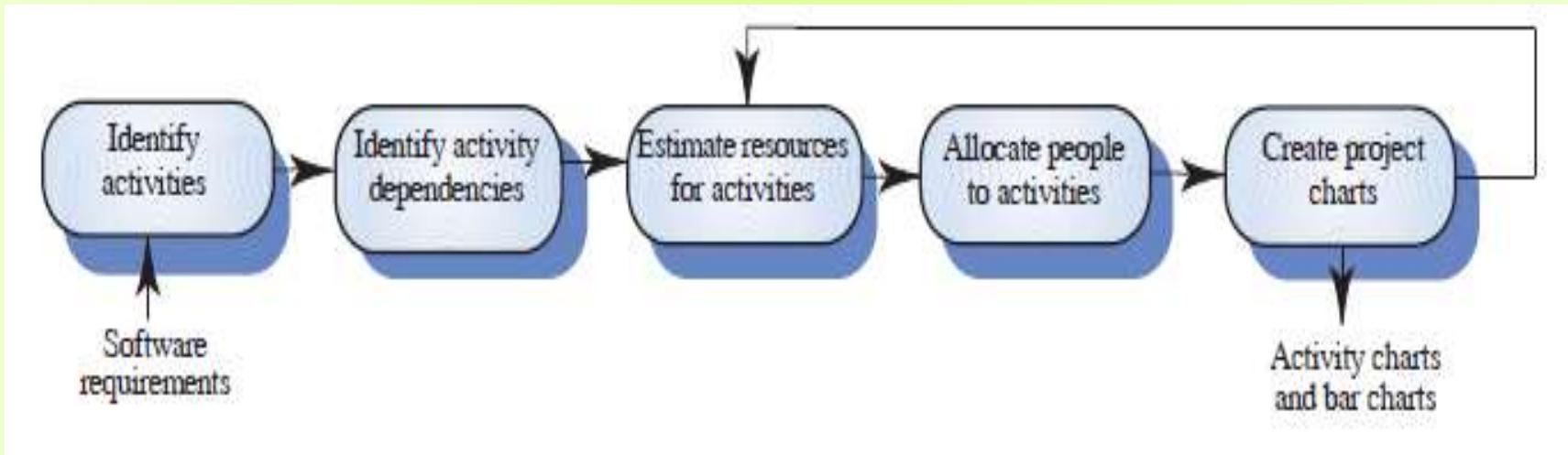


Deliverables are usually milestones

Project scheduling

- **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.

The project scheduling process



Scheduling problems

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

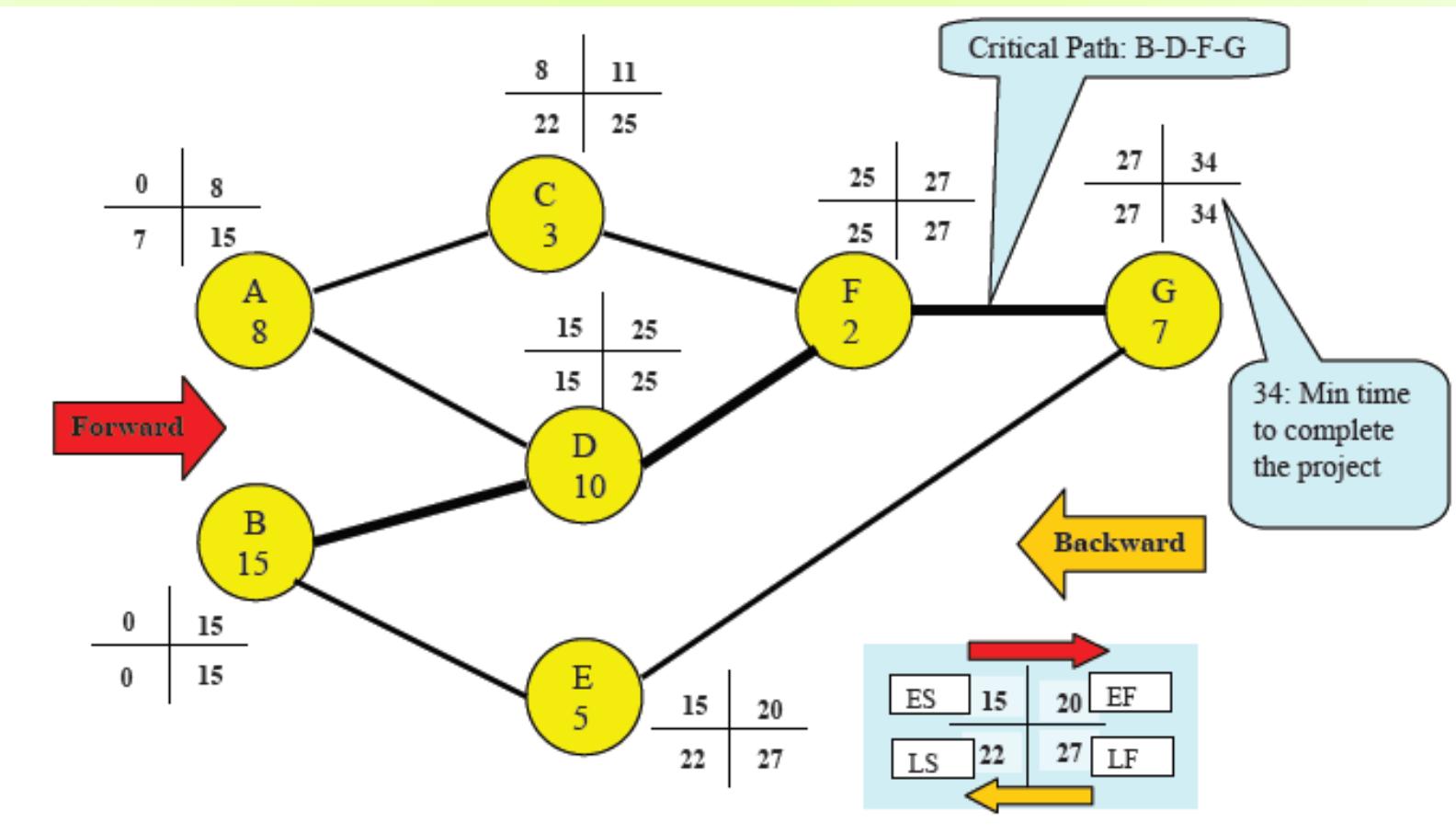
Bar charts and activity networks

- Graphical notations used to illustrate the project schedule.
- Show project breakdown into tasks. Tasks should not be too small. They should take about a week or two.
- Activity charts show task dependencies and the critical path.
- Bar charts show schedule against calendar time.

Project Precedence Table

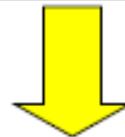
Task	Duration (Weeks)	Precedence
A	8	-
B	15	-
C	3	A
D	10	A, B
E	5	B
F	2	C, D
G	7	E, F

Activity network – Critical Path



Project Precedence Table

Task	Duration (Weeks)	Precedence	Earliest start	Earliest finish	Latest start	Latest finish	Slack
A	8	-	0	8	7	15	7
B	15	-	0	15	0	15	0
C	3	A	8	11	22	25	14
D	10	A, B	15	25	15	25	0
E	5	B	15	20	22	27	7
F	2	C, D	25	27	25	27	0
G	7	E, F	27	34	27	34	0

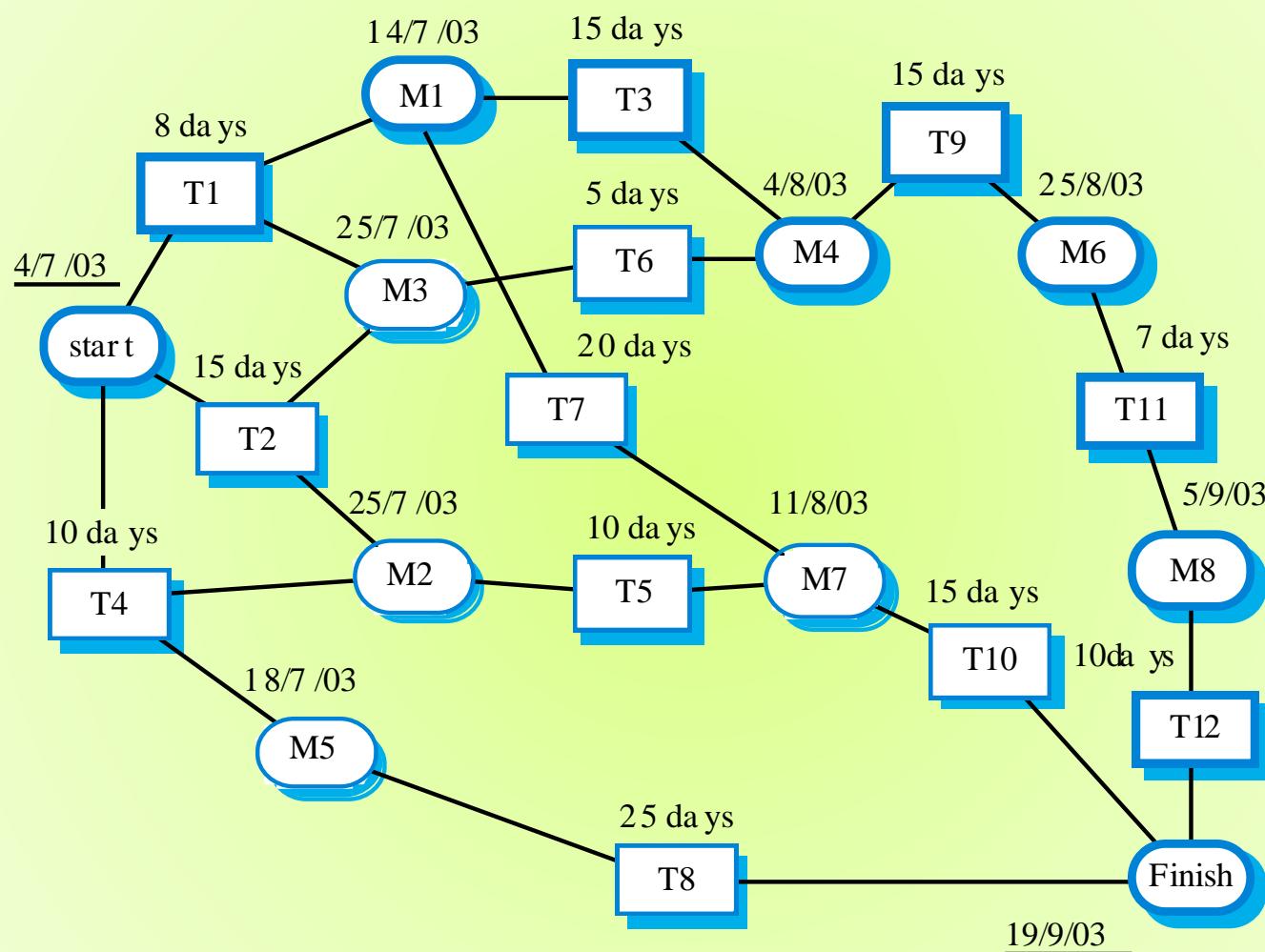


Critical task

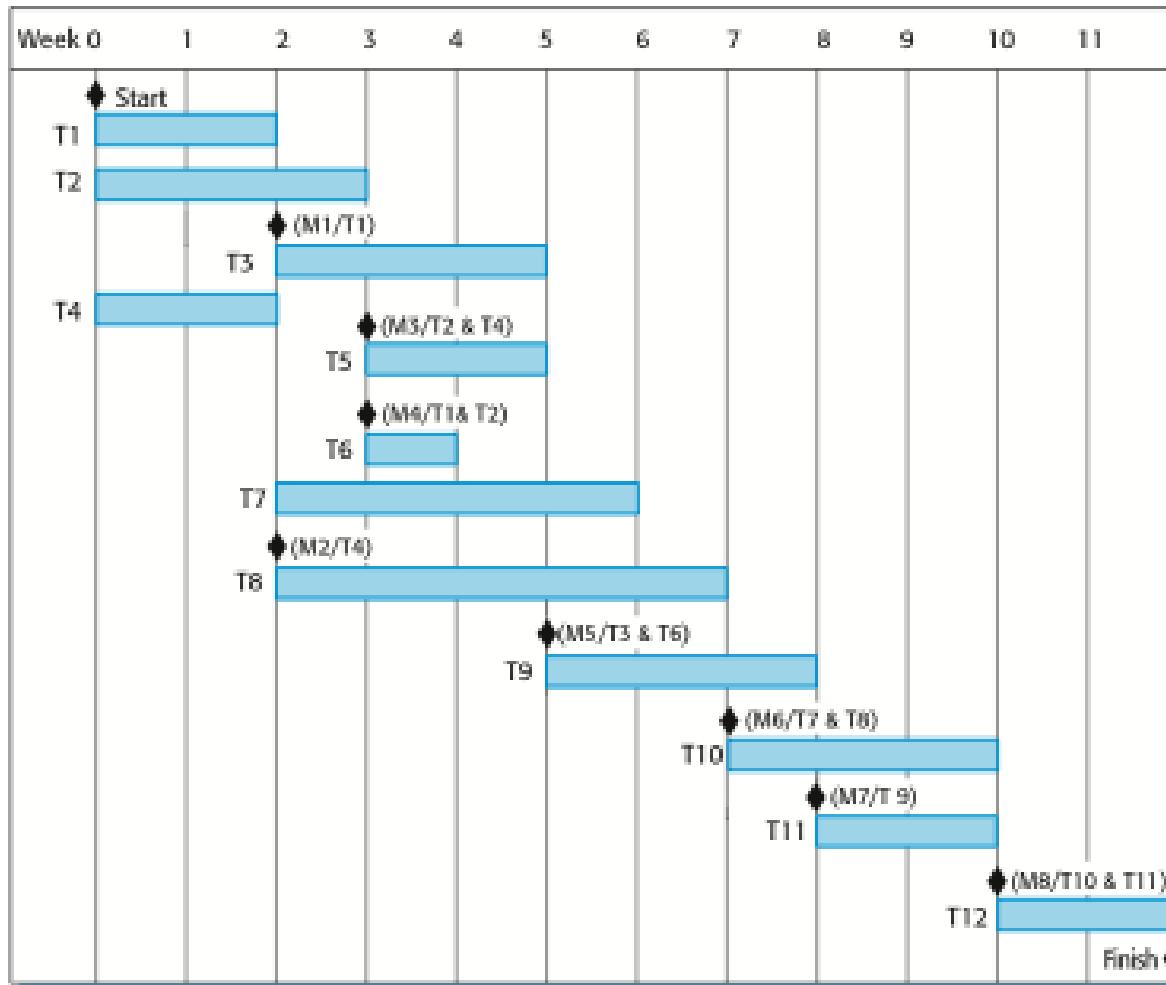
Task durations and dependencies

Activity	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

Activity network – (Task dependency)



Bar Chart



Object Oriented Analysis and Design Using the UML

Object Oriented Analysis

Object Oriented Analysis

Process of Object Modeling

- 1. Modeling the functions of the system.**
- 2. Finding and identifying the business objects.**
- 3. Organizing the objects and identifying their relationships.**

Object Oriented Analysis

Process of Object Modeling

- ◆ **Identifying objects:**
 - Using concepts, CRC cards, stereotypes, etc.
- ◆ **Organising the objects:**
 - classifying the objects identified, so similar objects can later be defined in the same class.
- ◆ **Identifying relationships between objects:**
 - this helps to determine inputs and outputs of an object.
- ◆ **Defining operations of the objects:**
 - the way of processing data within an object.
- ◆ **Defining objects internally:**
 - information held within the objects.

Goals of OO analysis

- ▶ What are the two main goals of OO analysis?
 - 1) Understand the customer's requirements
 - 2) Describe problem domain as a set of classes and relationships
- ▶ What techniques have we studied for the 1st goal?
 - Develop a requirements specification
 - Describe scenarios of use in user's language as use cases
- ▶ What techniques have we studied for the 2nd goal?
 - CRC cards discover classes and run simulations
 - UML class diagrams represent classes & relationships
 - Sequence diagrams model dynamic behavior of a system

Construction the Analysis Use-Case Model

System analysis use case – a use case that documents the interaction between the system user and the system. It is highly detailed in describing what is required but is free of most implementation details and constraints.

1. Identify, define, and document new actors.
2. Identify, define, and document new use cases.
3. Identify any reuse possibilities.
4. Refine the use-case model diagram (if necessary).
5. Document system analysis use-case narratives.

Use case diagrams

Use Case Diagrams describe the functionality of a system and users of the system.

Describe the functional behavior of the system as seen by the user.

These diagrams contain the following elements:

- **Actors**, which represent users of a system, including human users and other systems.
- **Use Cases**, which represent functionality or services provided by a system to users.

Use case diagrams

Use case diagrams are considered for high level requirement analysis of a system. So when the requirements of a system are analyzed the functionalities are captured in use cases.

To draw an use case diagram we should have the following items identified. Functionalities to be represented as an use case
Actors Relationships among the use cases and actors.

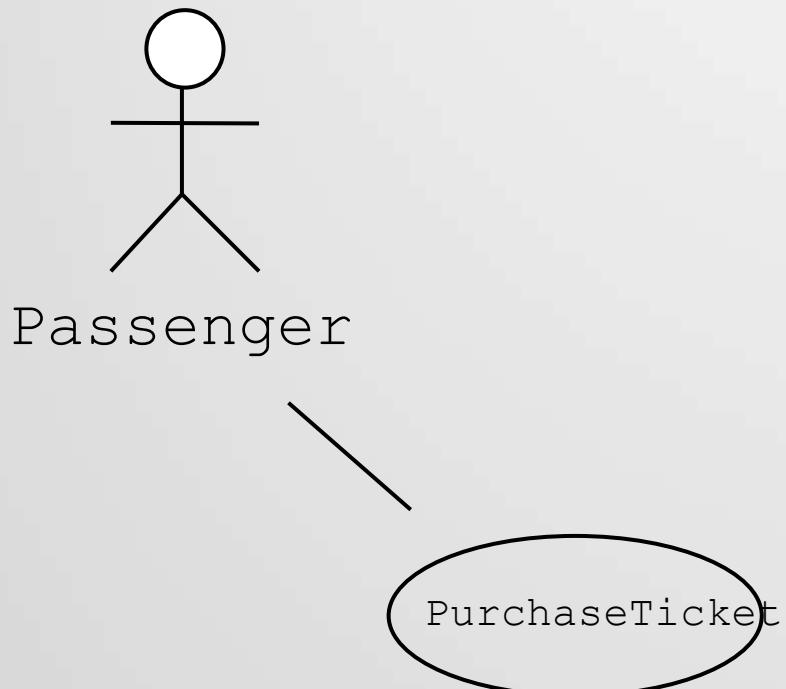
Use case diagrams

Use case diagrams are drawn to capture the functional requirements of a system. So after identifying the above items we have to follow the following guidelines to draw an efficient use case diagram.

The name of a use case is very important. So the name should be chosen in such a way so that it can identify the functionalities performed.

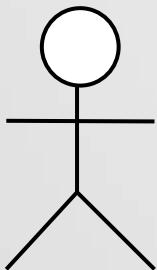
- ❑ Give a suitable name for actors. Show relationships and dependencies clearly in the diagram.
- ❑ Do not try to include all types of relationships.
- ❑ Because the main purpose of the diagram is to identify requirements.
- ❑ Use note when ever required to clarify some important points.

Use Case Diagrams



- ▶ Used during requirements elicitation to represent external behavior
- ▶ *Actors* represent roles, that is, a type of user of the system
- ▶ *Use cases* represent a sequence of interaction for a type of functionality
- ▶ The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

Actors



Passenger

- ▶ An actor models an external entity which communicates with the system:
 - User
 - External system
 - Physical environment
- ▶ An actor has a unique name and an optional description.
- ▶ Examples:
 - Passenger: A person in the train
 - GPS satellite: Provides the system with GPS coordinates

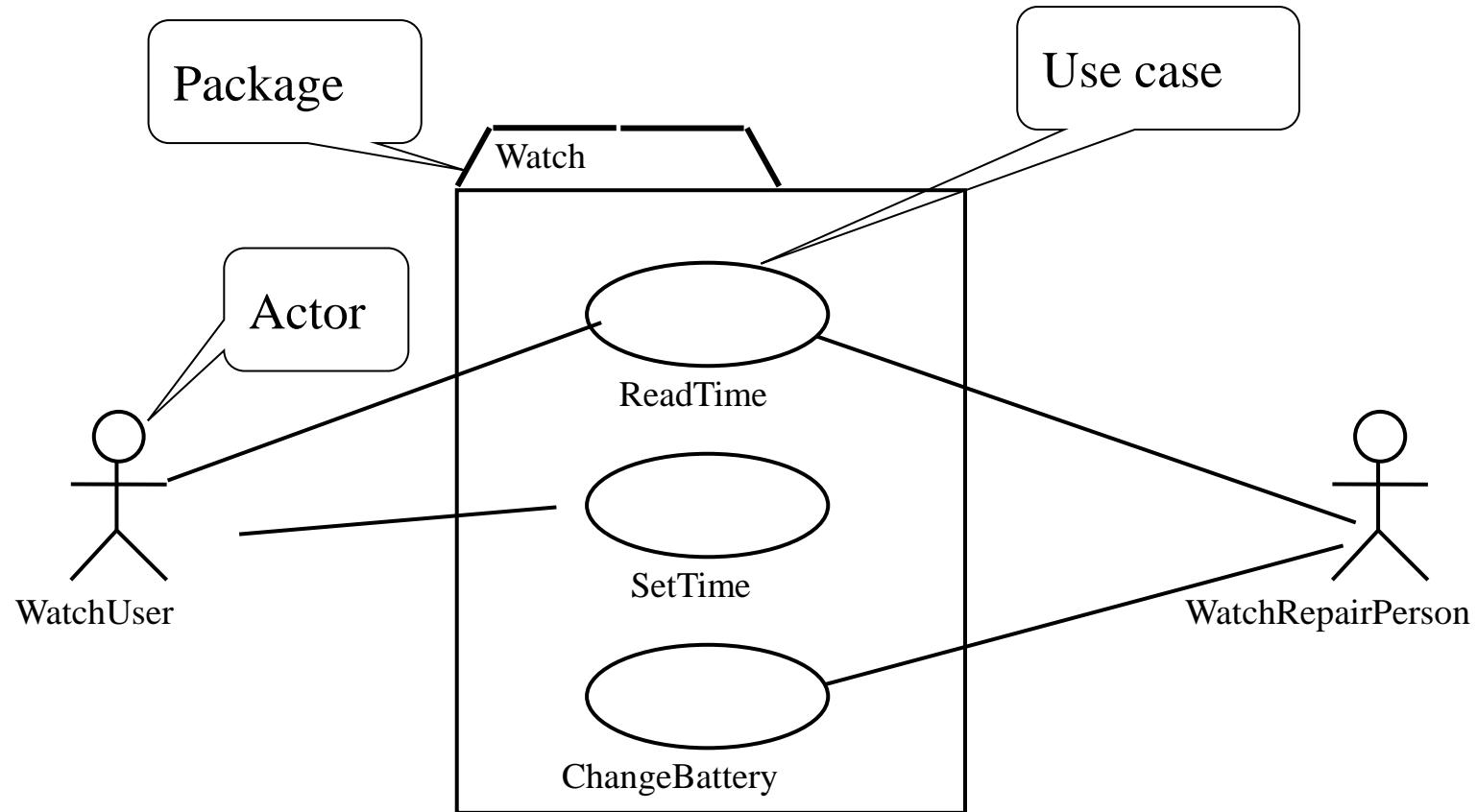
Use Case

A use case represents a class of functionality provided by the system as an event flow.

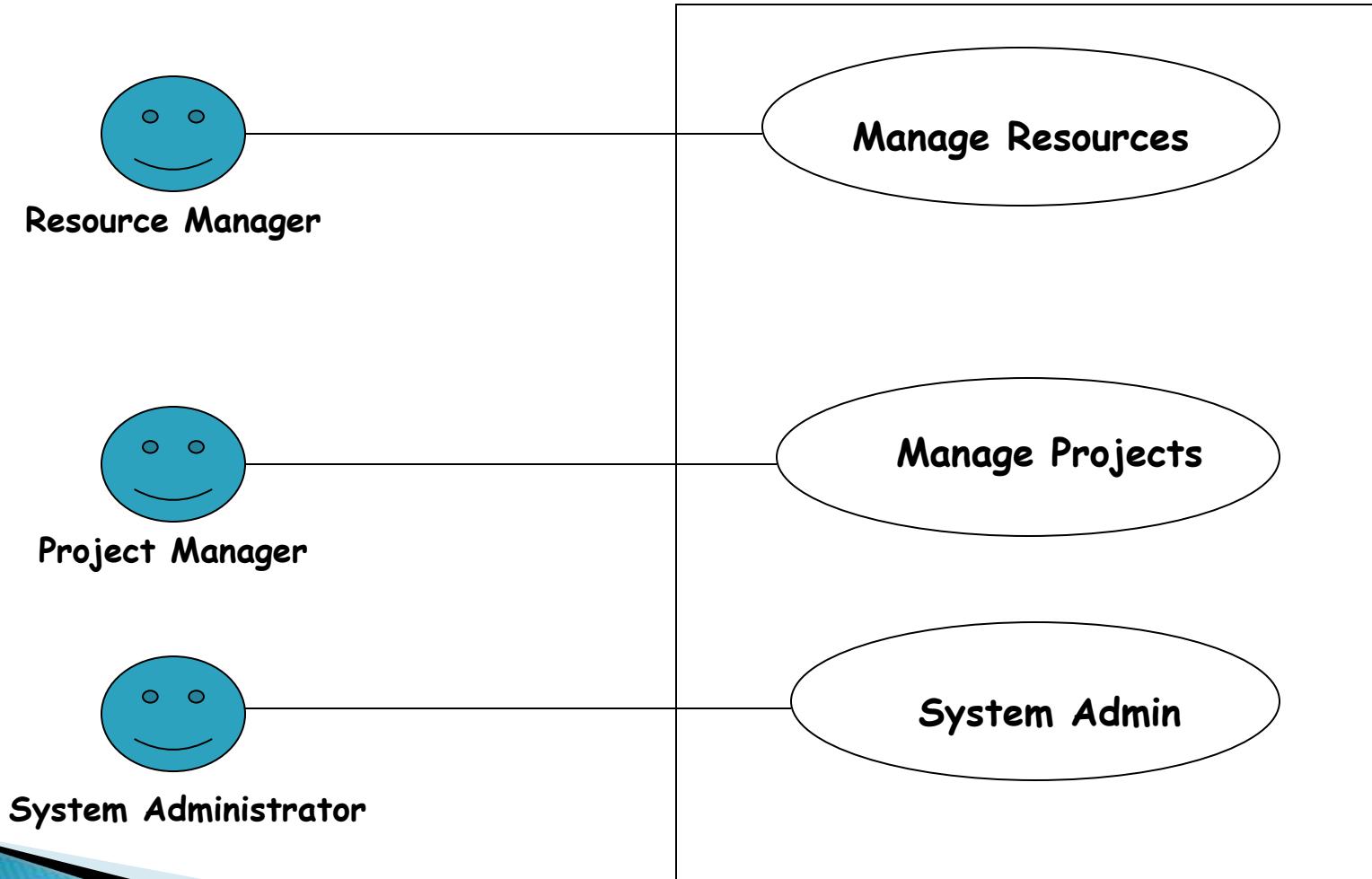


- A use case consists of:
- ▶ Unique name
 - ▶ Participating actors
 - ▶ Entry conditions
 - ▶ Flow of events
 - ▶ Exit conditions
 - ▶ Special requirements

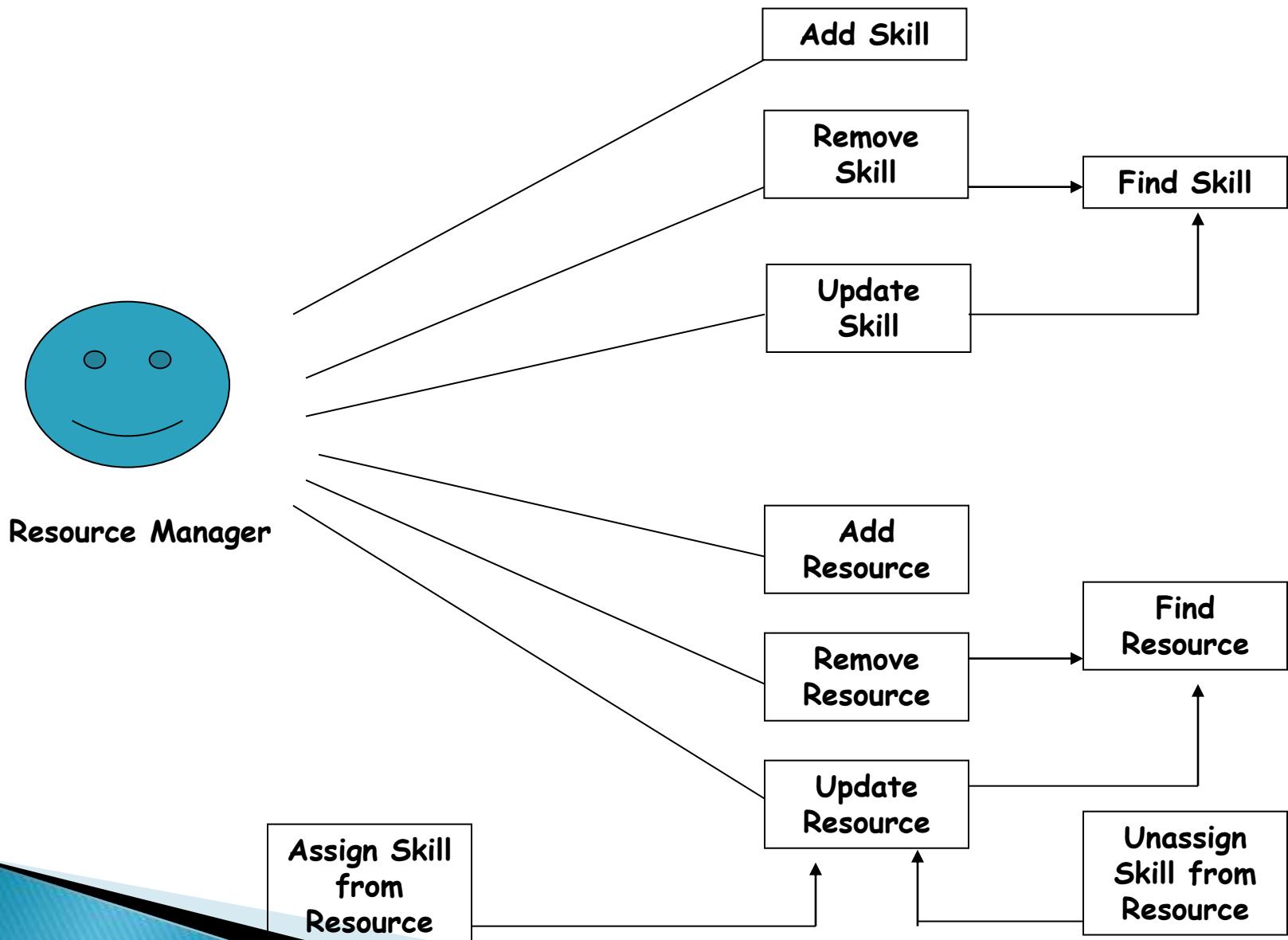
Use case diagrams



Example: High Level Use Case Diagram



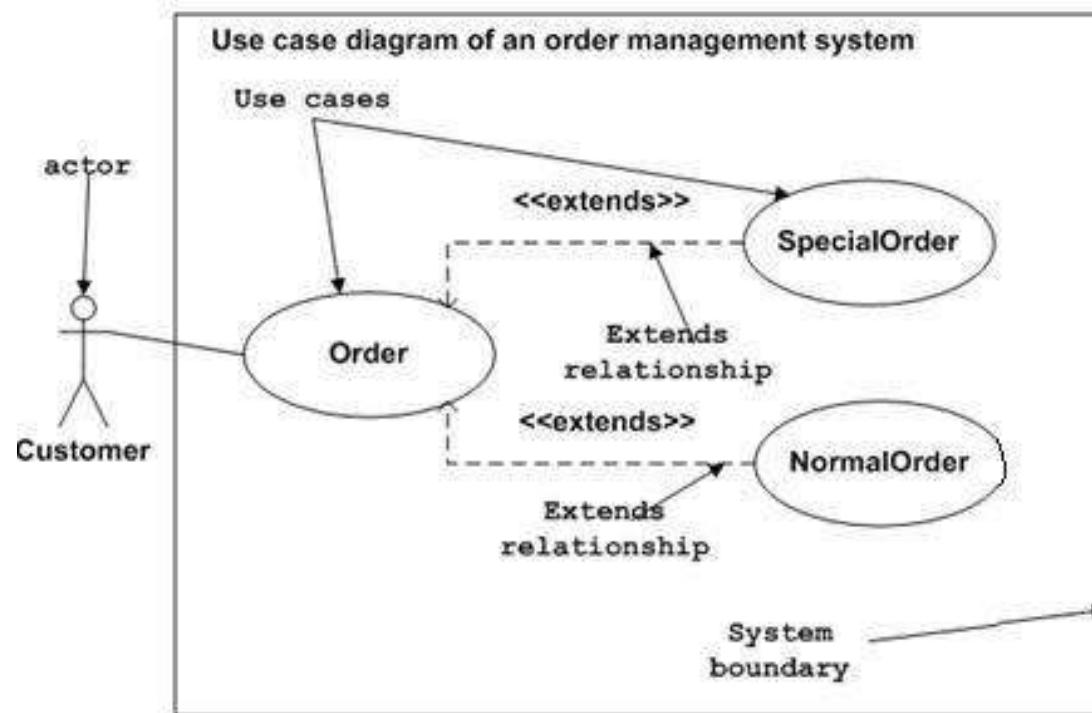
Example: Managing Resources Use Case Diagram



Example: Order management system

The following is a sample use case diagram representing the order management system. So if we look into the diagram then we will find three use cases (Order, SpecialOrder and NormalOrder) and one actor which is customer.

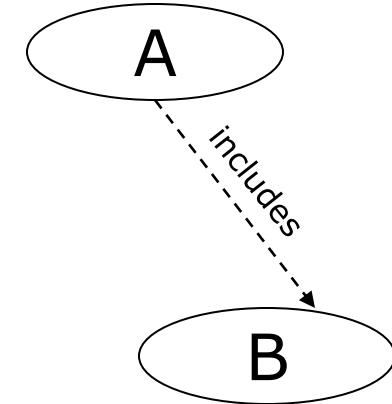
The SpecialOrder and NormalOrder use cases are extended from Order use case. So they have extends relationship.



Dependences

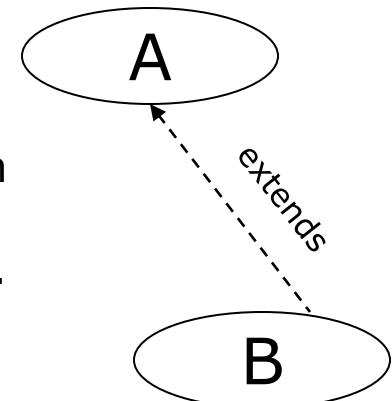
Include Dependencies

An include dependency from one use case (called the base use case) to another use case (called the inclusion use case) indicates that the base use case will include or call the inclusion use case. A use case may include multiple use cases, and it may be included in multiple use cases.



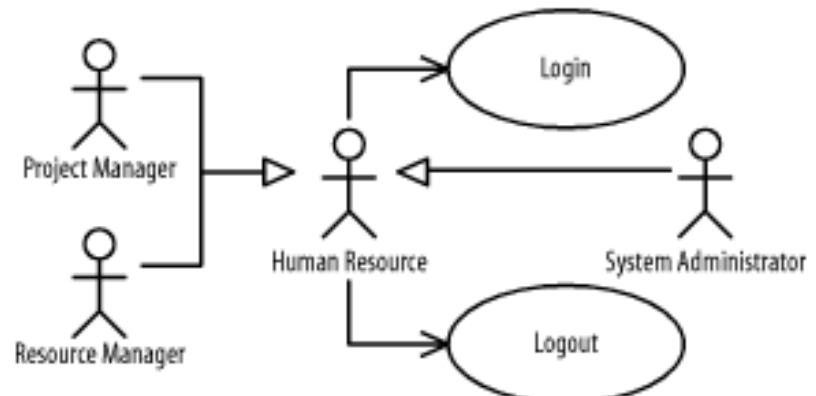
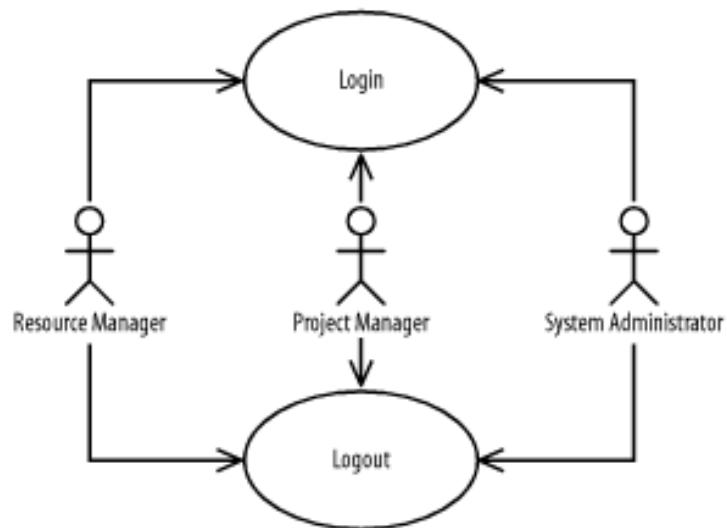
Extend Dependencies

An extend dependency from one use case (called the extension use case) to another use case (called the base use case) indicates that the extension use case will extend (or be inserted into) and augment the base use case. A use case may extend multiple use cases, and a use case may be extended by multiple use cases.



Generalizations

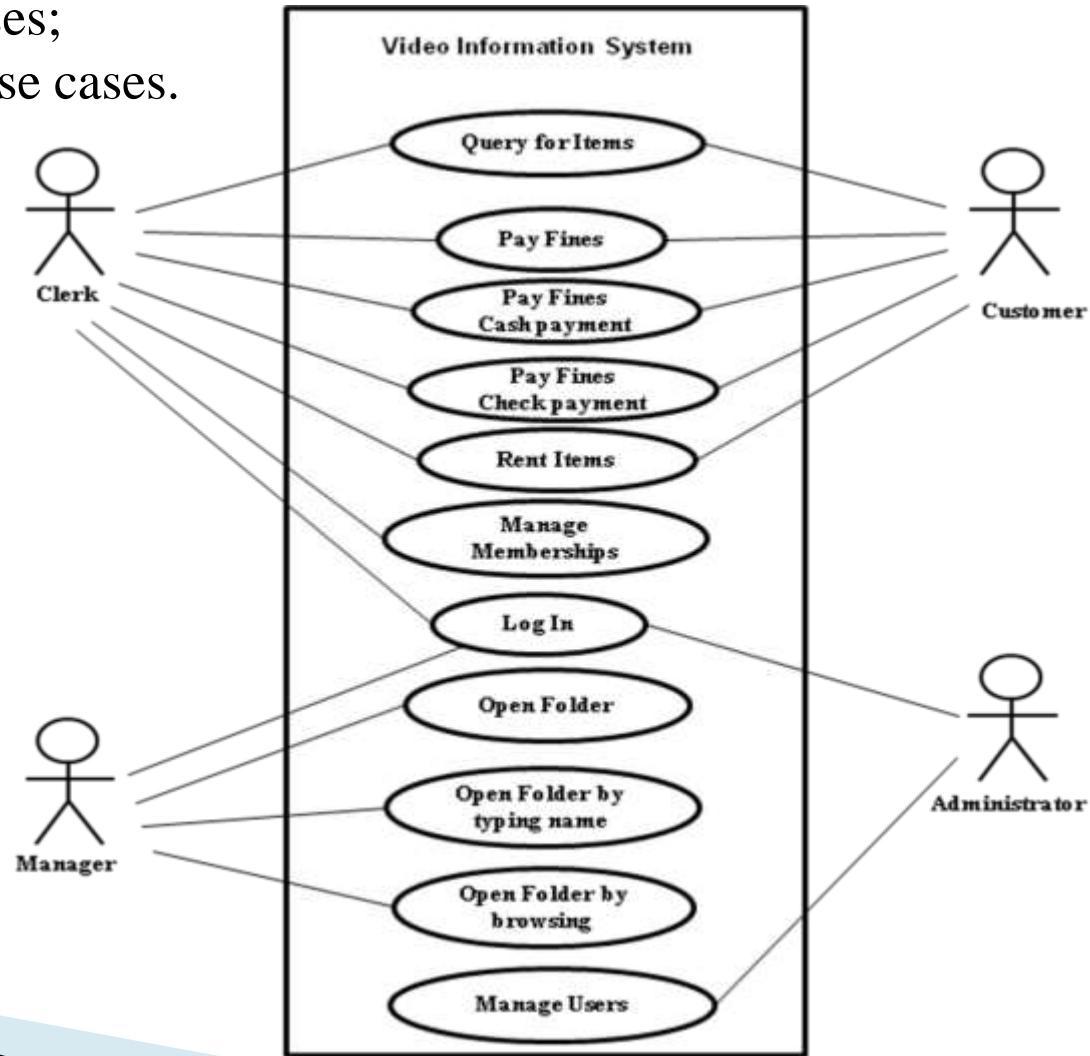
- ❑ Actors may be similar in how they use a system; for example, project managers, resource managers, and system administrators may log in and out of our project management system.
- ❑ Use cases may be similar in the functionality provided to users; for example, a project manager may publish a project's status in two ways: by generating a report to a printer or by generating a web site on a project web server.

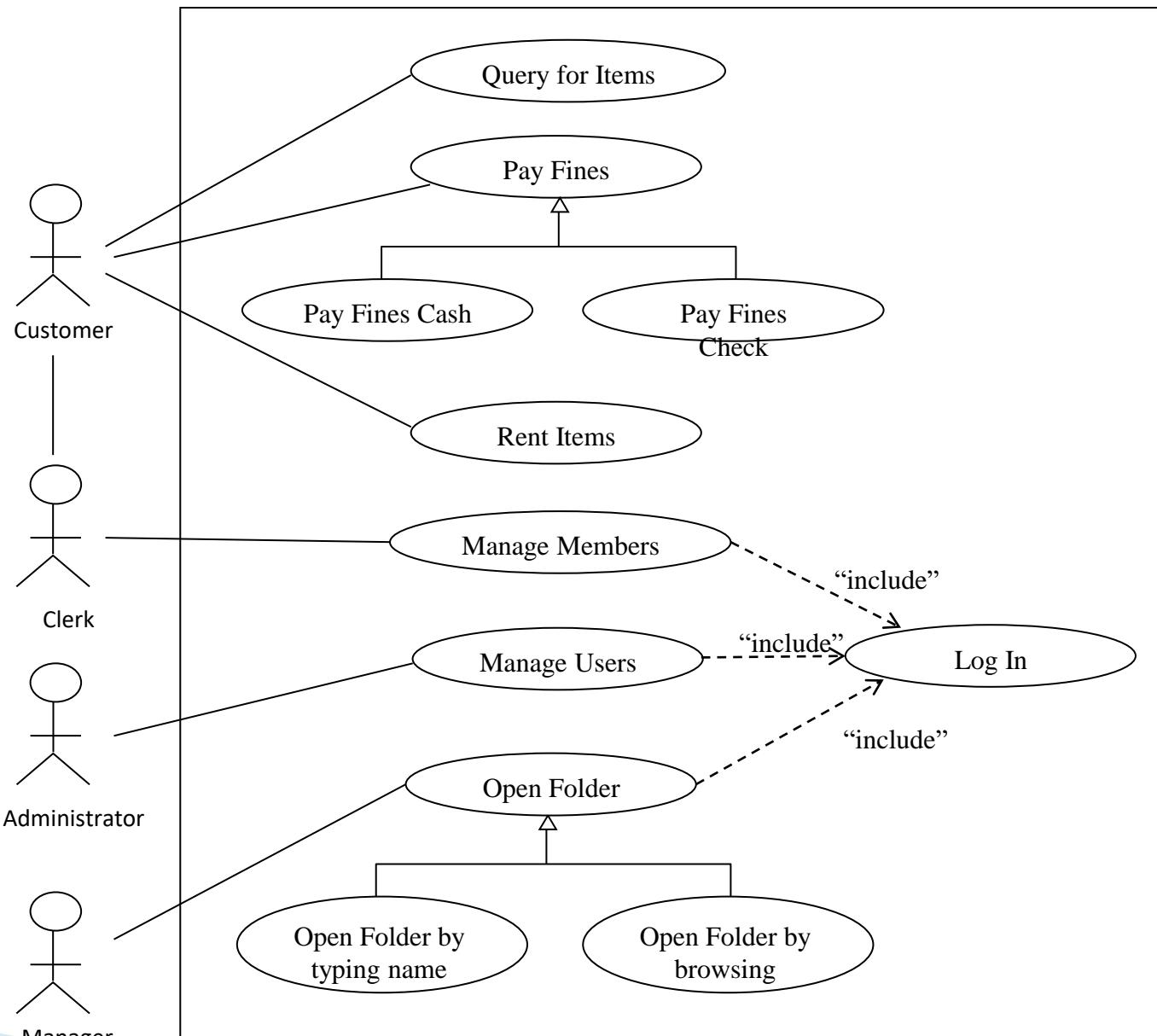


Example

Redraw the given use case diagram after applying:

- generalization between actors
- generalization between use cases;
- include relationship between use cases.





Use Case Description

Use case text provides the detailed description of a particular use case

Use Case ID:	Give each use case a unique integer sequence number identifier.	
Use Case Name:	Start with a verb.	
Created By:	Last Updated By:	
Date Created:		Date Last Updated:

Actors:	Calls on the system to deliver its services.
Description:	"user-goal" or "sub-function"
Stakeholders and Interests:	Who cares about this use case, and what do they want?
Trigger:	Identify the event that initiates the use case.
Pre-conditions:	What must be true on start, and worth telling the reader?
Post-conditions:	Describe the state of the system at the conclusion of the use case execution.
Normal Flow:	A typical, unconditional happy path scenario of success.
Alternative Flows (Extensions):	Alternative scenarios of success or failure.
Priority:	Indicate the relative priority of implementing the functionality required to allow this use case to be executed.
Technology and Data Variations List	Varying I/O methods and data formats.
Special Requirements:	Related non-functional requirements.
Notes and Issues:	Such as open issues.

Use Case Description

- ▶ **Use Case Identification**
 - **Use Case ID**

Give each use case a unique integer sequence number identifier.
 - **Use Case Name**

State a concise, results-oriented name for the use case. These reflect the tasks the user needs to be able to accomplish using the system. Include an action verb and a noun.
 - **Use Case History**
 - Created By
 - Date Created
 - Last Updated By
 - Date Last Updated
 - **Actors**

An actor is a person or other entity external to the software system being specified who interacts with the system and performs use cases to accomplish tasks. Name the actor that will be initiating this use case and any other actors who will participate in completing the use case.
 - **Description**

Provide a brief description of the reason for and outcome of this use case.
 - **Stakeholders and Interests**

Who cares about this use case, and what do they want?
 - **Trigger**

Identify the event that initiates the use case. This could be an external business event or system event that causes the use case to begin, or it could be the first step in the normal flow.

Use Case Description

Use Case Definition

- **Pre-conditions**

List any activities that must take place, or any conditions that must be true, before the use case can be started. Number each precondition. Examples:

- User's identity has been authenticated.
- User's computer has sufficient free memory available to launch task.

- **Post-conditions**

Describe the state of the system at the conclusion of the use case execution. Number each post-condition. Examples:

- Price of item in database has been updated with new value.

- **Normal (basic) Flow of events – Happy path – Successful path – Main Success Scenario**

Provide a detailed description of the user actions and system responses that will take place during execution of the use case under normal, expected conditions. This dialog sequence will ultimately lead to accomplishing the goal stated in the use case name and description.

- **Alternative Flows (Extensions): Alternate scenarios of success or failure**

Document other, legitimate usage scenarios that can take place within this use case separately in this section. State the alternative flow, and describe any differences in the sequence of steps that take place. Number each alternative flow in the form "X.Y", where "X" is the Use Case ID and Y is a sequence number for the alternative flow. For example, "5.3" would indicate the third alternative flow for use case number 5.

Use Case Description

- **Priority**

Indicate the relative priority of implementing the functionality required to allow this use case to be executed. The priority scheme used must be the same as that used in the software requirements specification.

- **Technology and Data Variations List**

Varying I/O methods and data formats.

- **Special Requirements**

Identify any additional requirements, such as nonfunctional requirements, for the use case that may need to be addressed during design or implementation. These may include performance requirements or other quality attributes.

- **Notes and Issues**

List any additional comments about this use case or any remaining open issues or TBDs (To Be Determined) that must be resolved. Identify who will resolve each issue, the due date, and what the resolution ultimately is.

Text and Diagrams

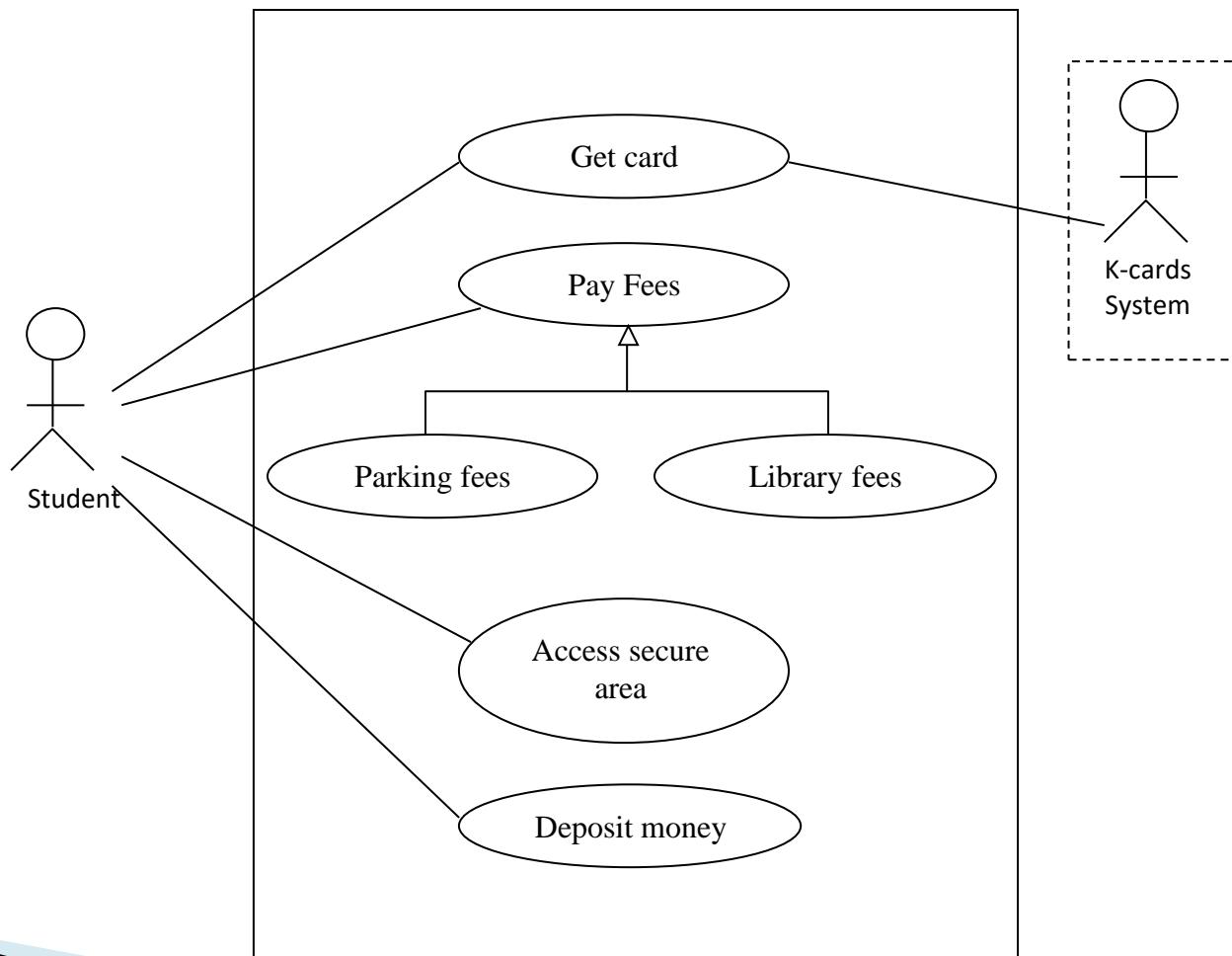
- ▶ Use case *text* provides the detailed description of a particular use case
- ▶ Use case *diagram* provides an overview of interactions between actors and use cases

Use case diagram for Club Sport

The ClubRiyadh Sport has decided to implement an electronic card system for its subscriber, so that subscribers can use their K-cards to access secure areas, and also as a debit card, linked to an account into which subscribers can deposit money to be used to pay club fees. For the initial release of the system, this will be limited to a few club usages: equipment rental at the sports centre, beverage fees, and library fees at club libraries. The system will keep a usage record for each K-card.

Identify use cases by providing the actors, use case names. Draw the use case diagram.

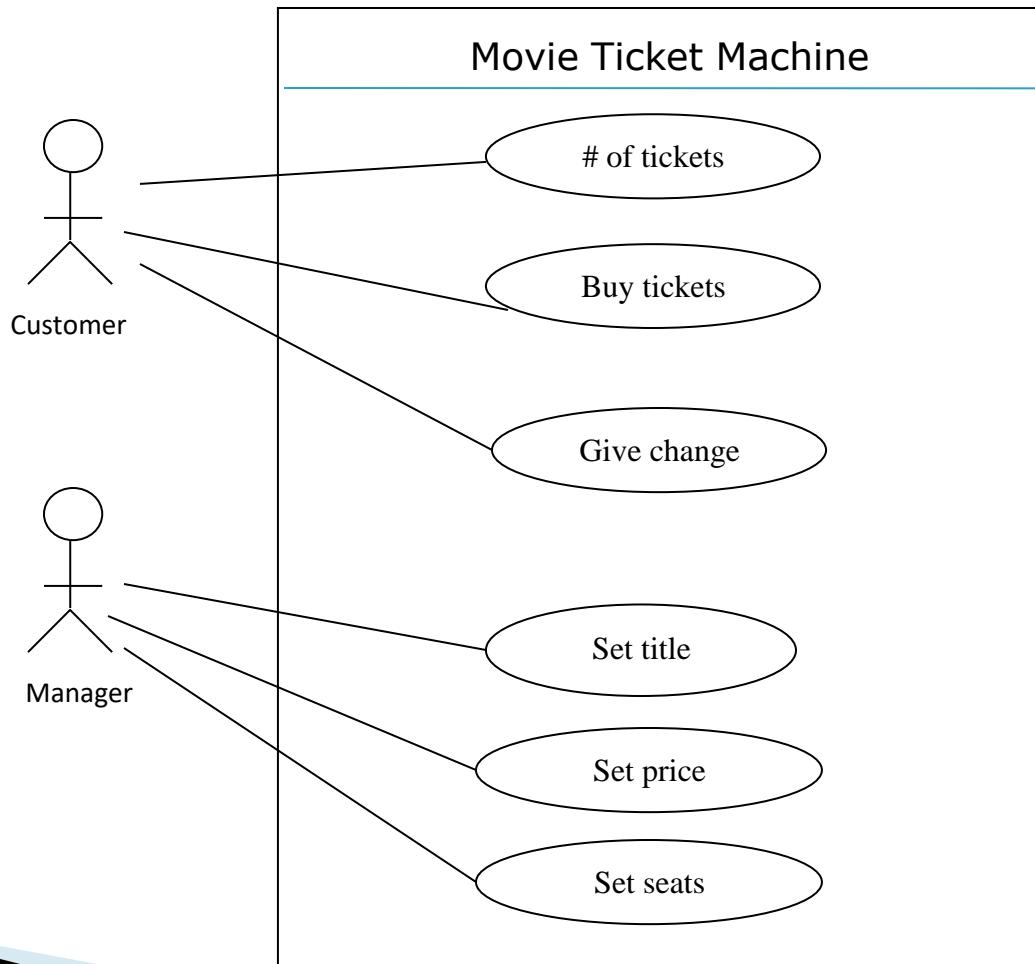
Use case diagram for Club Sport



Use case diagram for Movie Ticket Machine

- ▶ I am the manager of a theatre.
- ▶ I want to create an automated movie ticket machine.
- ▶ You are analysts who need to describe what the customer wants as a set of use cases
- ▶ Simplifying assumptions:
 - One movie showing at a time
 - Movie time is same every day, only one time, same price
 - Only manager can change/add movie
 - Customer can only buy tickets
- ▶ **Who or what are the actors?**
- ▶ **What are the use cases (goals of actors)?**

Use case diagram for Movie Ticket Machine



Identification of Use Cases

Use cases for Manager

- ▶ Use case: Set title
 - ▶ Actors: Manager, Machine
 - ▶ 1. Manager requests a change of movie title
 - ▶ 2. Machine asks manager for new movie title
 - ▶ 3. Manager enters movie title
- ▶ Use case: Set price
 - ▶ Actors: Manager, Machine
 - ▶ 1. Manager requests a change of ticket price
 - ▶ 2. Machine asks manager for new price for movie title
 - ▶ 3. Manager enters ticket price
 - ▶ Alternatives: Invalid price
 - ▶ If manager enters price below SR5 or greater than SR50
 - ▶ 3a. Machine asks manager to reenter price
- Use case: Set seats
 - Actors: Manager, Machine
 - 1. Manager requests a change in number of seats
 - 2. Machine asks manager for number of seats in theatre
 - 3. Manager enters number of seats
 - Alternatives: Invalid number of seats
 - If manager enters number less than 20 or greater than 999
 - 3a. Machine asks manager to reenter number of seats

Identification of Use Cases

Use cases for Customer

- ▶ Use case: # of tickets
 - ▶ Actors: Customer, Machine
 - ▶ 1. Customer enters number of tickets
 - ▶ 2. Machine displays total balance due
 - ▶ Alternative: Customer wants zero tickets
 - ▶ At step 1, customer enters zero tickets
 - ▶ 1a. Display thank you message
 - ▶ 1b. Set balance to \$0.0

- ▶ Use case: Return change to customer
 - ▶ Actors: Customer, Machine
 - ▶ 1. Customer requests change
 - ▶ 2. Machine dispenses money
 - ▶ 3. Machine updates customer balance

- Use case: Buy tickets
 - Actors: Customer, Machine
 - 1. Customer requests tickets
 - 2. Machine tells customer to put balance due in money slot
 - 3. Customer enters money in money slot
 - 4. Machine updates customer balance
 - 5. Customer requests tickets
 - 6. Machine prints tickets
 - 7. Machine updates number of seats
 - Alternative: Insufficient seats
 - At step 1, if number of tickets requested is less than available seats,
 - 1a. Display message and end use case
 - Alternative: Insufficient funds
 - At step 5, if money entered < total cost,
 - 5a. Display insufficient amount entered
 - 5b. Go to step 3

OO domain modeling with UML class diagrams and CRC cards

What is a Domain Model?

- ❑ Illustrates meaningful conceptual classes in problem domain
- ❑ Represents real-world concepts, not software components
- ❑ A diagram (or set of diagrams) which represents real world *domain objects*
 - '*conceptual classes*'
- ❑ *Not a set of diagrams describing software classes, or software objects with responsibilities*

Why make a Domain Model?

- ▶ to understand what concepts need to be modelled by our system, and how those concepts relate
- ▶ a springboard for designing software objects

What Domain Model should it show?

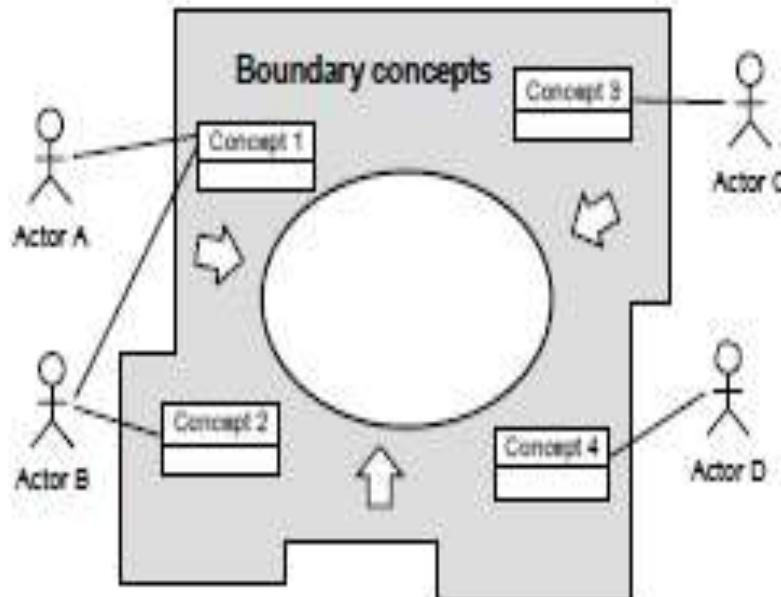
- ▶ conceptual classes
- ▶ associations between conceptual classes
- ▶ attributes of conceptual classes

Building the Domain Model

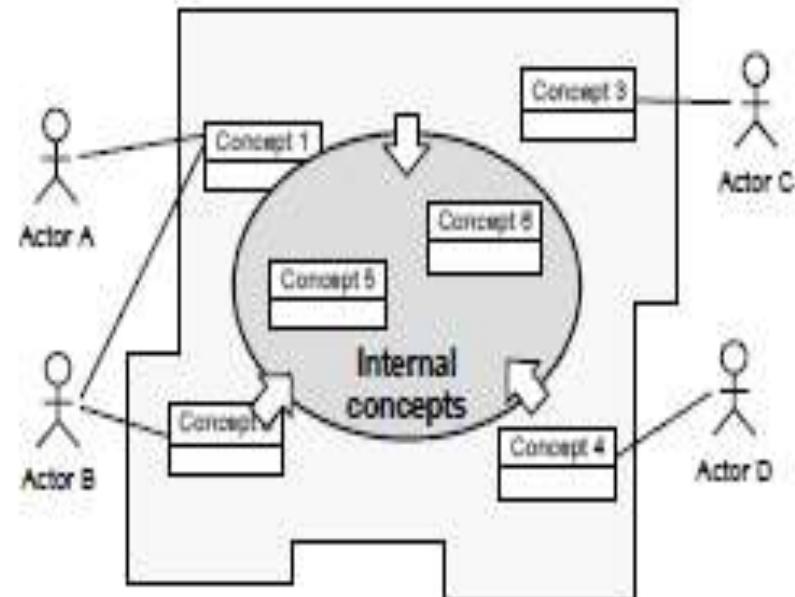
A useful strategy for building a domain model is to start with:

- the “boundary” concepts that interact directly with the actors
- and then identify the internal concepts

Step 1: Identifying the boundary concepts



Step 2: Identifying the internal concepts



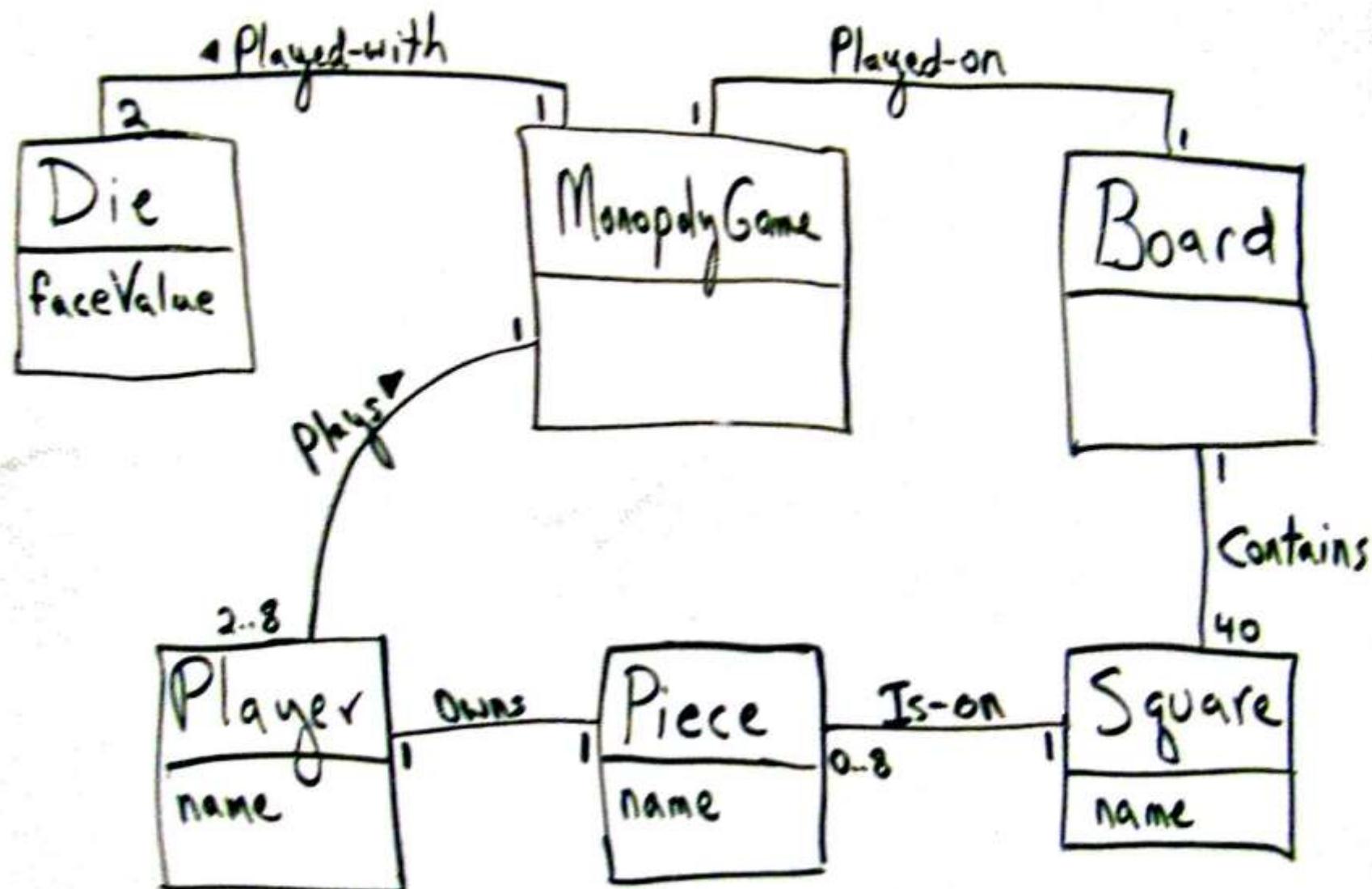
Steps to create a Domain Model

- ❑ Identify Candidate Conceptual classes
- ❑ Draw them in a Domain Model
- ❑ Add associations necessary to record the relationships that must be retained
- ❑ Add attributes necessary for information to be preserved

Identify conceptual classes

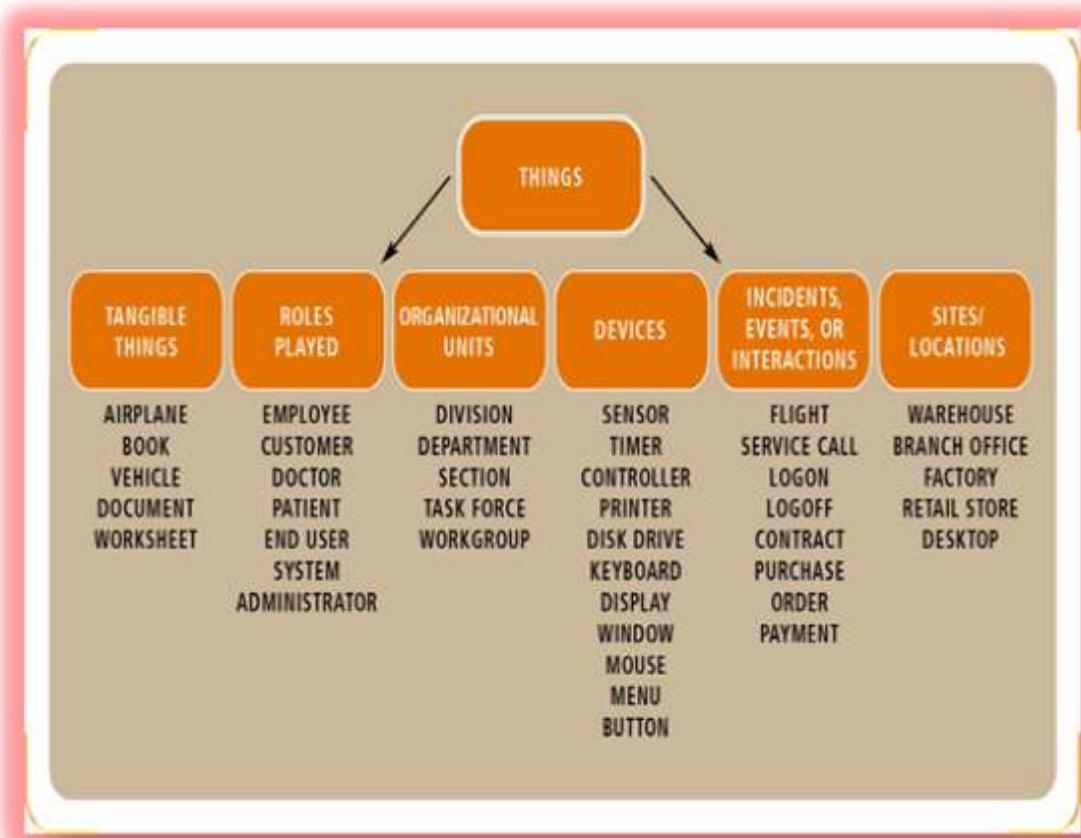
- Three strategies to find conceptual classes
 - Reuse or modify existing models
 - There are published, well-crafted domain models and data models for common domains: inventory, finance, health..
 - Books: *Analysis patterns* by Martin Fowler, *Data Model Patterns* by David Hay, *Data Model Resource Book* by Len Silverston
 - Use a category list
 - Identify noun phrases

Example of Domain Model



Use a category list

- Finding concepts using the concept category list :
 - **Physical objects**: register, airplane, blood pressure monitor
 - **Places**: airport, hospital
 - **Catalogs**: Product Catalog
 - **Transactions**: Sale, Payment, reservation



Identify conceptual classes from noun phrases

- Finding concepts using **Noun Phrase** identification in the textual description of the domain :
 - Noun Phrase Identification [Abbot 83]
 - Analyze ***textual description*** of the domain
 - Identify nouns and noun phrases
(indicate candidate classes or attributes)
 - Caveats:
 - Automatic mapping isn't possible
 - Textual descriptions are ambiguous!
(different words may refer to the same class)
- Noun phrases may also be attributes or parameters rather than classes:
 - If it stores state information or it has multiple behaviors, then it's a class
 - If it's just a number or a string, then it's probably an attribute

Identifying objects

- Look for **nouns** in the SRS (System Requirements Specifications) document
- Look for **NOUNS** in use cases descriptions
- A **NOUN** may be
 - Object
 - Attribute of an object

Identifying Operations ‘methods’

- Look for verbs in the SRS (System Requirements Specifications) document
- Look for **VERBS** in use cases descriptions
- A **VERB** may be
 - translated to an **operation** or set of operations
 - A method is the code implementation of an operation.

Example: Identify conceptual classes from noun phrases

Consider the following problem description, analyzed for Subjects, Verbs, Objects:

The ATM verifies whether the customer's card number and PIN are correct.

S C V R O O A O A

If it is, then the customer can check the account balance, deposit cash, and withdraw cash.

S R V O A V O A V O A

Checking the balance simply displays the account balance.

S M O A V O A

Depositing asks the customer to enter the amount, then updates the account balance.

M S V O R V O A V O A

Withdraw cash asks the customer for the amount to withdraw; if the account has enough cash,

S M A O V O R O A V S C V O A

the account balance is updated. The ATM prints the customer's account balance on a receipt.

O A V C S V O A O

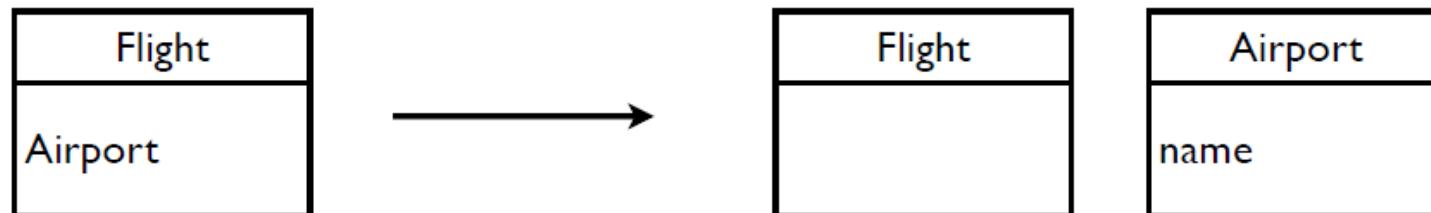
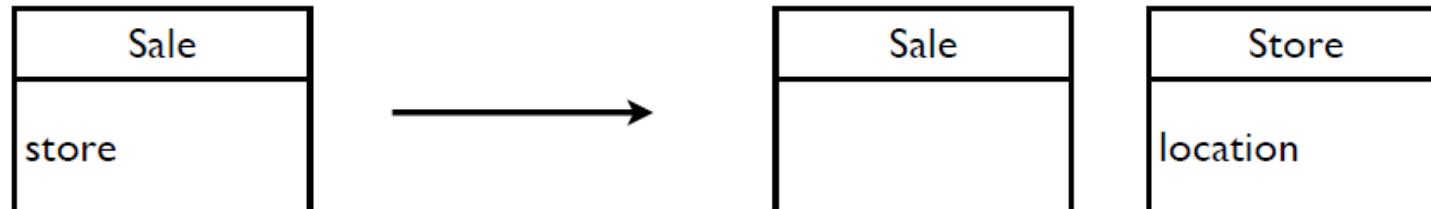
Analyze each **subject** and **object** as follows:

- Does it represent a person performing an action? Then it's an actor, 'R'.
- Is it also a verb (such as 'deposit')? Then it may be a method, 'M'.
- Is it a simple value, such as 'color' (string) or 'money' (number)?
Then it is probably an attribute, 'A'.
- Which NPs are unmarked? Make it 'C' for class.

Verbs can also be classes, for example:

- Deposit is a class if it retains state information

Example



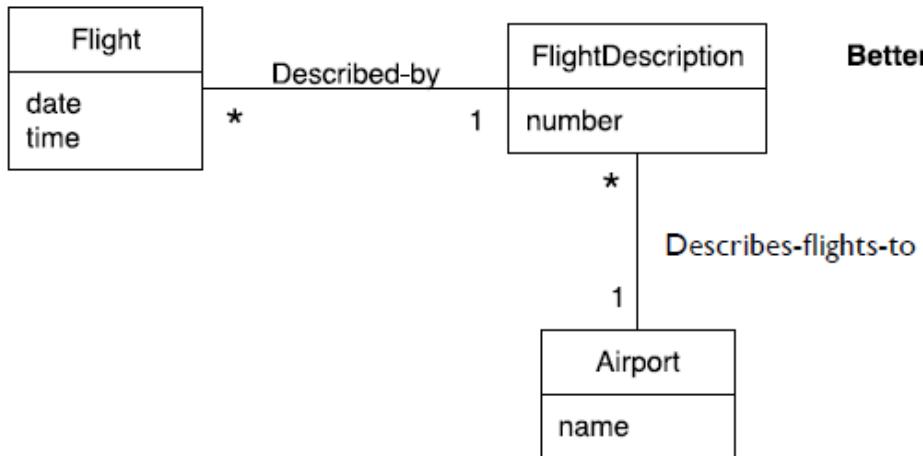
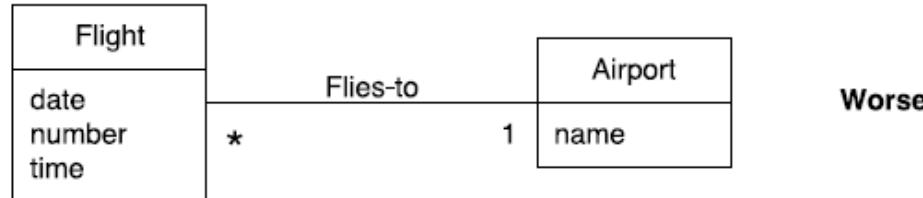
from Ch 9 Applying UML & Patterns (Larman 2004)

both Store and Airport represent concepts of interest in their own right, so we model them as conceptual classes rather than attributes

Adding Specification or Description Conceptual Classes

- ▶ sometimes we need to separate out the description of a concept from the concept itself, so that even if no instances of the concept exist at a given point in time, its description will still be present in the system.
- ▶ doing this can reduce unnecessary redundancy in recording information when we move into design and build.

Example



from Ch 9 Applying UML & Patterns (Larman 2004)

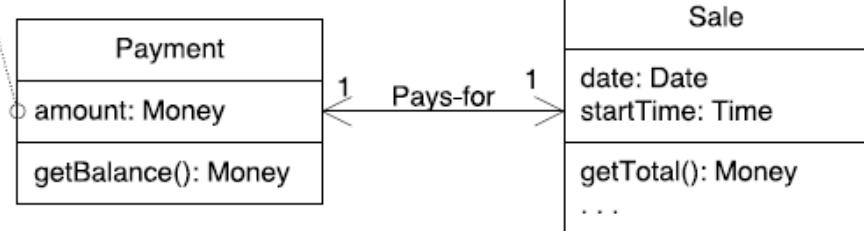
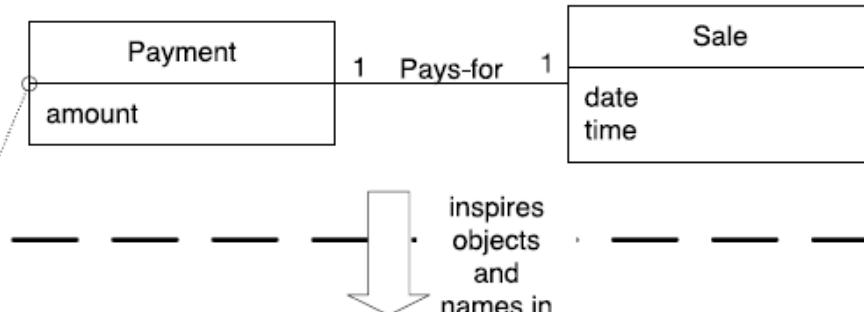
Domain versus Design Models

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

UP Domain Model
Stakeholder's view of the noteworthy concepts in the domain.



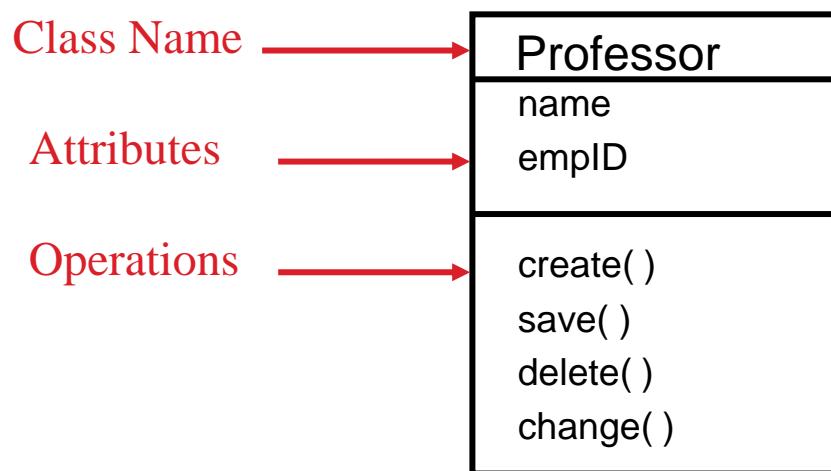
UP Design Model
The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

from Ch 9 Applying UML & Patterns (Larman 2004)

Class Compartments

- ▶ A class is comprised of three sections
 - The first section contains the class name
 - The second section shows the structure (attributes)
 - The third section shows the behavior (operations)



Basic Concepts of Object Orientation

- ▶ Object
- ▶ Class
- ▶ Attribute
- ▶ Operation
- ▶ Interface (Polymorphism)
- ▶ Relationships



Basic Concepts of Object Orientation

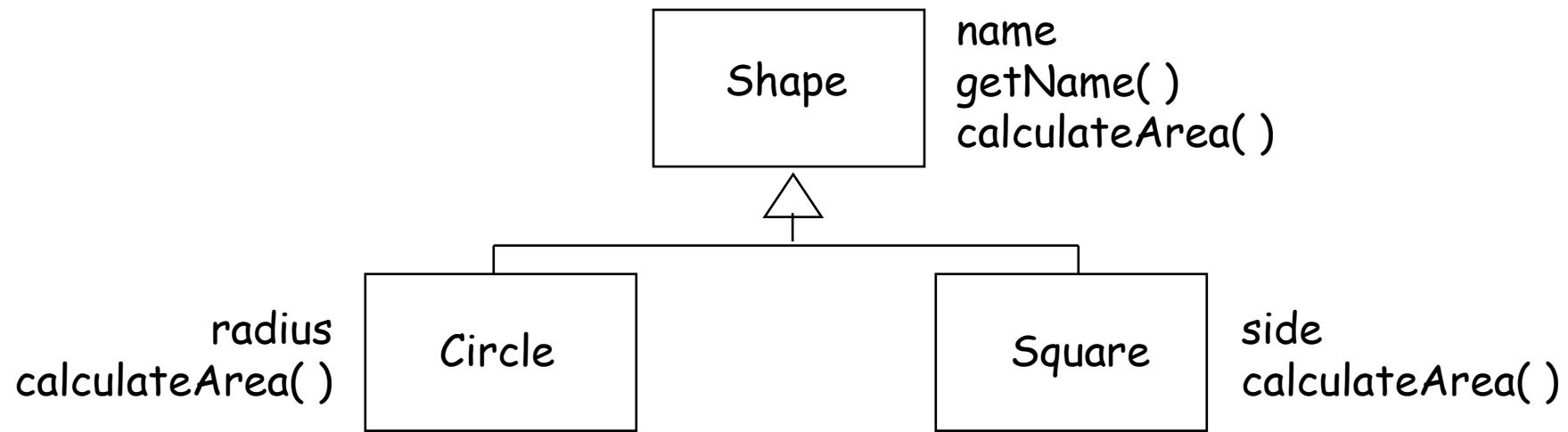
- ▶ Object
- ▶ Class
- ▶ Attribute
- ▶ Operation
- ▶ Interface (Polymorphism)
- ▶ Relationships



Basic Concepts of Object Orientation

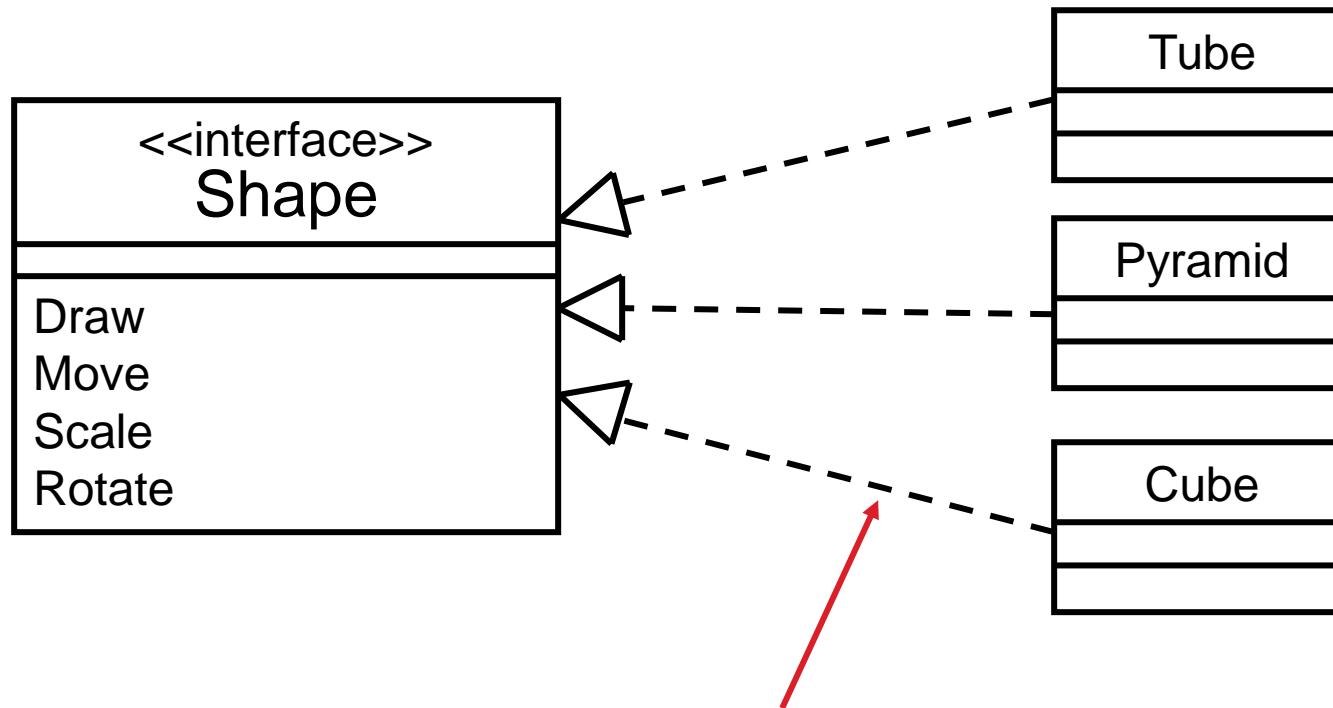
- ▶ Object
- ▶ Class
- ▶ Attribute
- ▶ Operation
- ★ ▶ Interface (Polymorphism)
- ▶ Relationships

What is Polymorphism?



What is an Interface?

- ▶ Interfaces formalize polymorphism



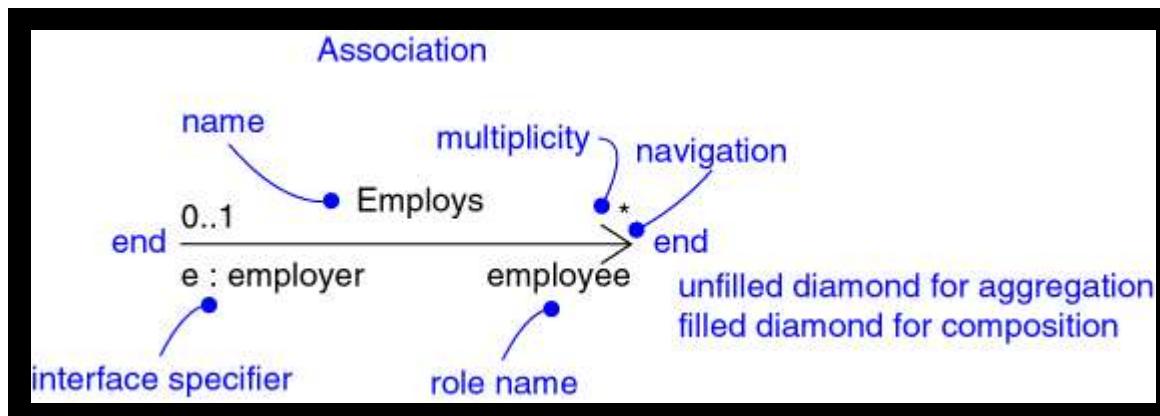
(stay tuned for realization relationships)

Basic Concepts of Object Orientation

- ▶ Object
 - ▶ Class
 - ▶ Attribute
 - ▶ Operation
 - ▶ Interface (Polymorphism)
- ★ ▶ Relationships

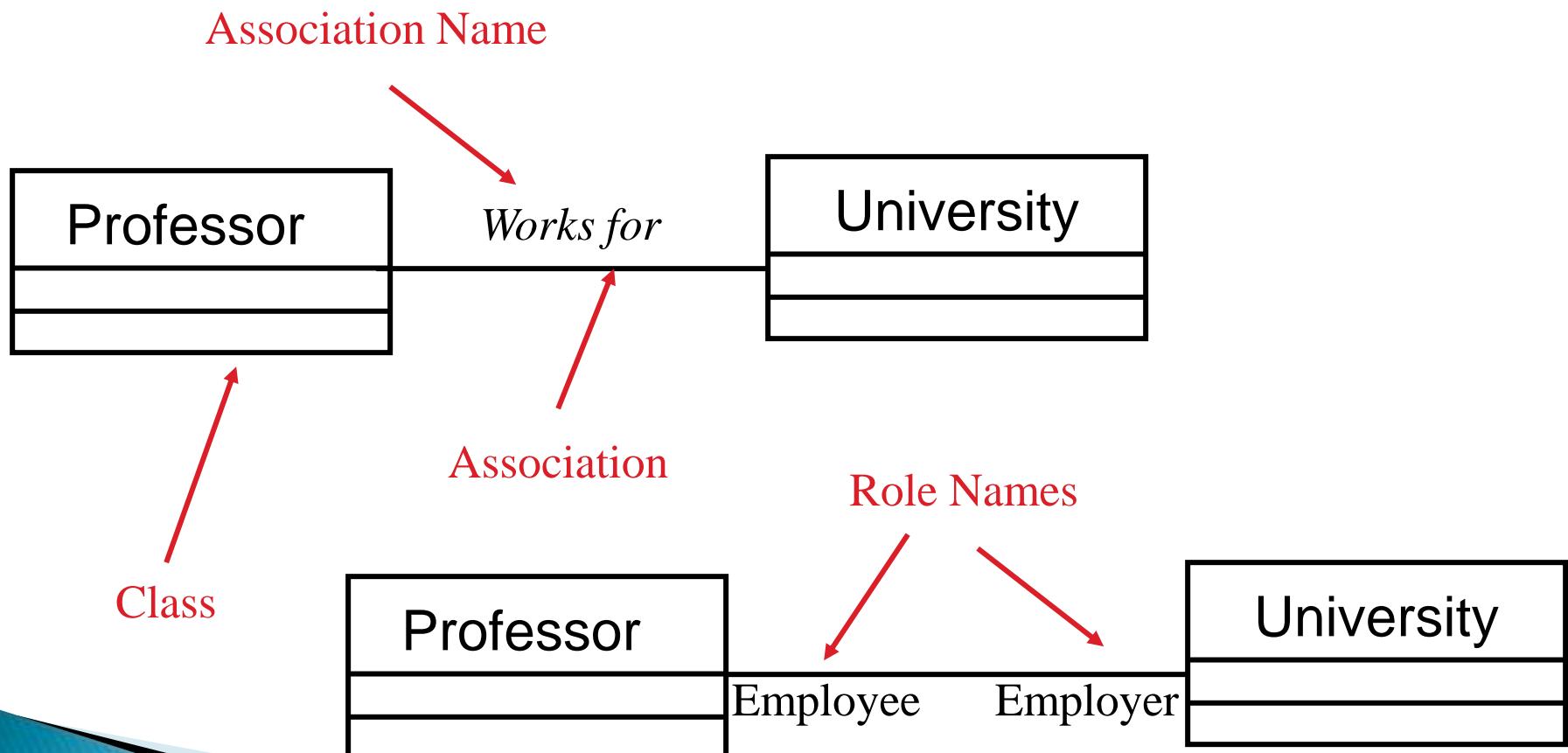
Relationships

- ▶ Association
 - Aggregation
 - Composition
- ▶ Dependency
- ▶ Generalization
- ▶ Realization



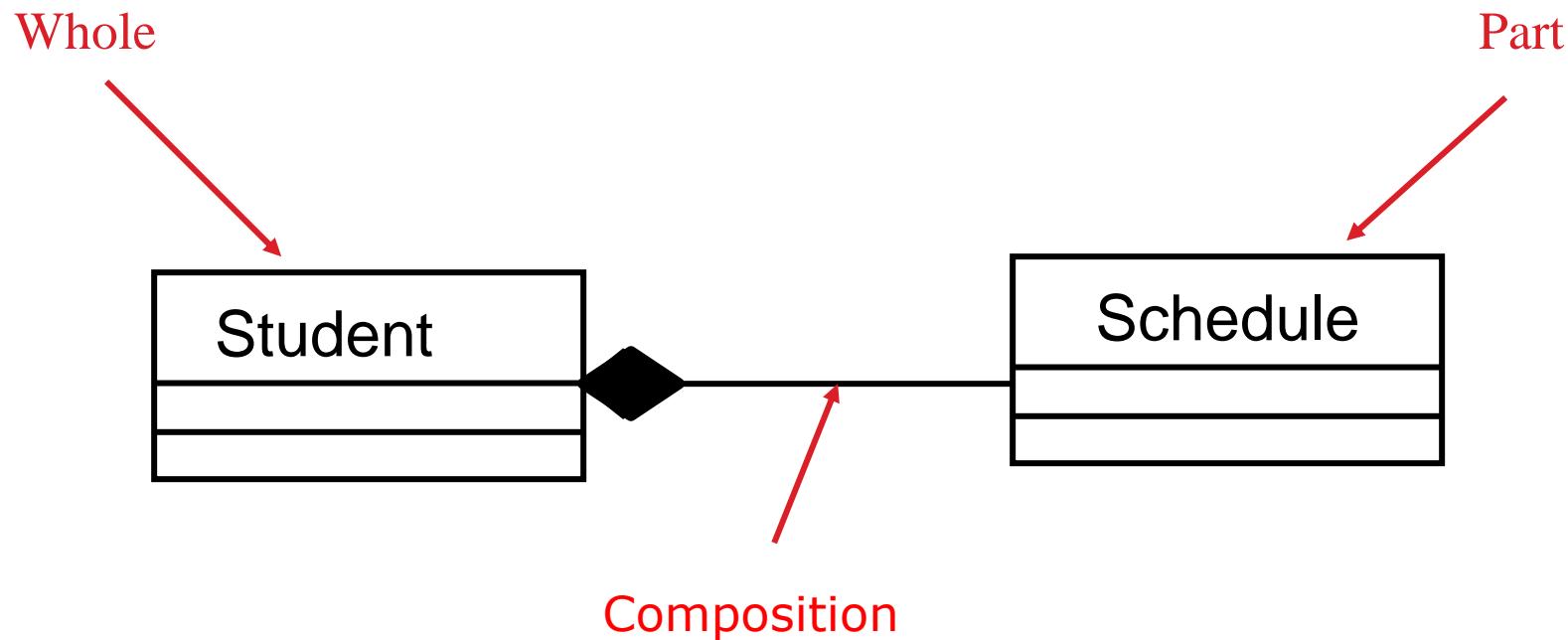
Relationships: Association

- Models a semantic connection among classes



Relationships: Composition

- ▶ A special form of association that models a whole–part relationship between an aggregate (the whole) and its parts



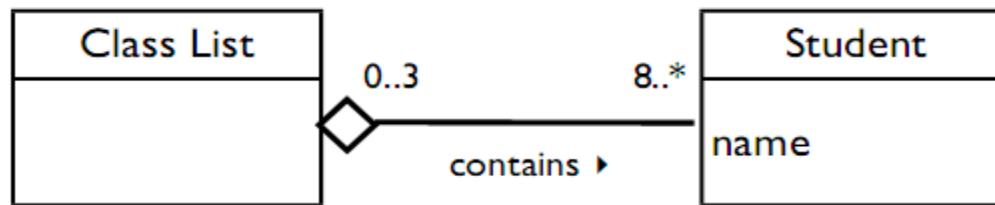
Relationships: Composition

Composition:



without the chess board, the square wouldn't exist...

Aggregation:



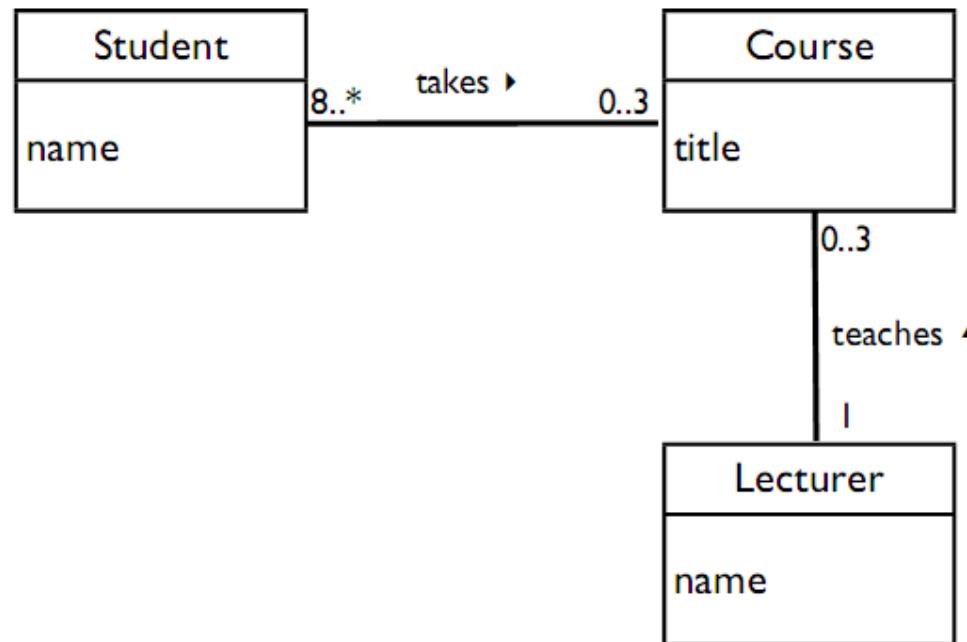
...but without the class list the student would

Association: Multiplicity and Navigation

- ▶ Multiplicity defines how many objects participate in a relationships
 - The number of instances of one class related to ONE instance of the other class
 - Specified for each end of the association
- ▶ Associations and aggregations are bi-directional by default, but it is often desirable to restrict navigation to one direction
 - If navigation is restricted, an arrowhead is added to indicate the direction of the navigation

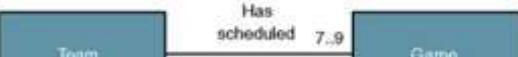
Association: Multiplicity

- how many instances of class A can be associated with a single class B *at a particular point in time*



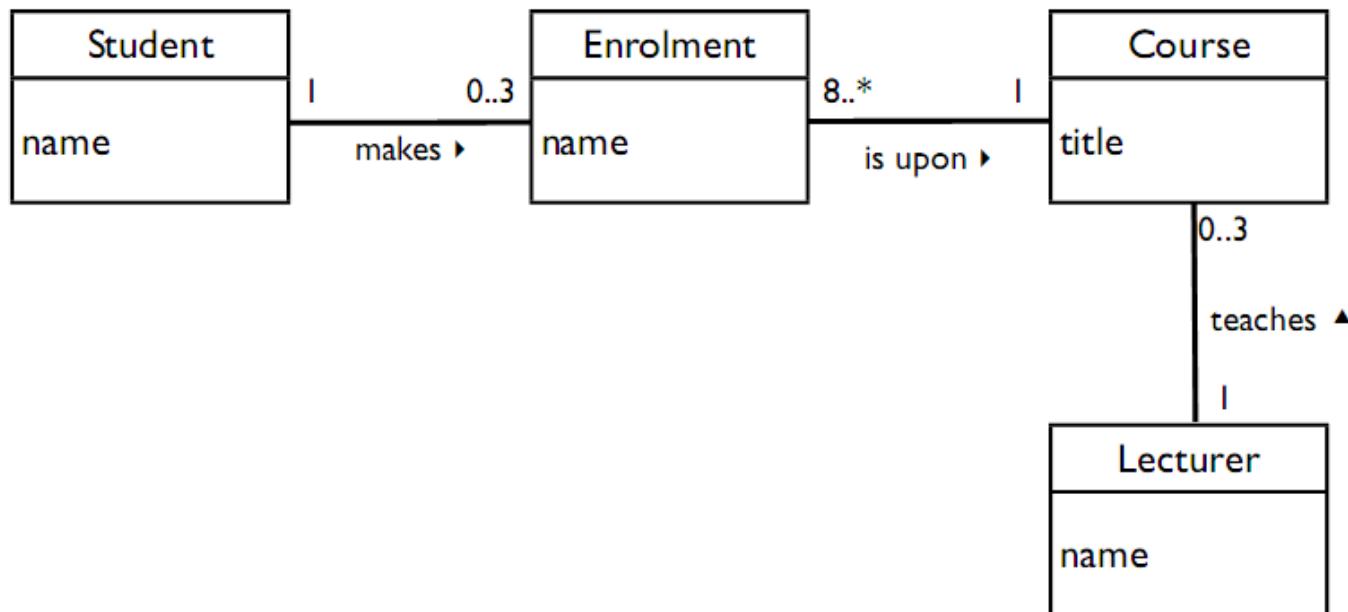
Type of Multiplicity

Multiplicity – the minimum and maximum number of occurrences of one object/class for a single occurrence of the related object/class.

Multiplicity	UML Multiplicity Notation	Association with Multiplicity	Association Meaning
Exactly 1	1 or <i>Leave Blank</i>	 	An employee works for one and only one department.
Zero or 1	0..1		An employee has either one or no spouse.
Zero or more	0..* or *	 	A customer can make no payment up to many payments.
1 or more	1..*		A university offers at least 1 course up to many courses.
Specific range	7..9		A team has either 7, 8, or 9 games scheduled

Example: Multiplicity and Navigation

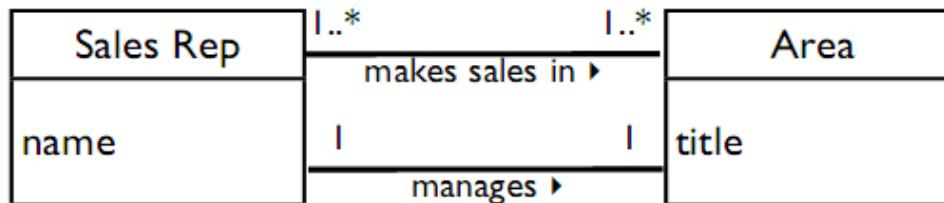
- where possible try to make all associations 1-to-n (or 0-to-n), since this will help to identify deeper concepts in the domain



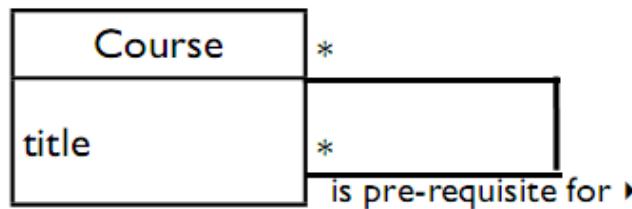
Multiple & Reflexive Associations

- can two conceptual classes have multiple associations with each other, and can a class associate with itself?

yes,



and yes



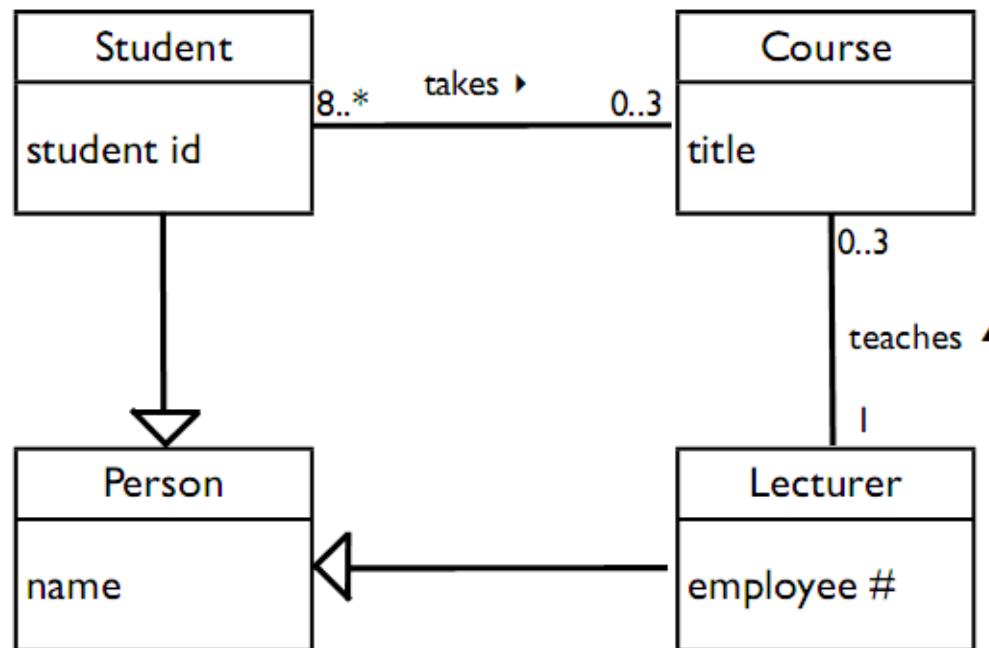
but in each case it might be better to use generalisation and/or to add further conceptual classes to the model

Relationships: Generalization

- ▶ A relationship among classes where one class shares the structure and/or behavior of one or more classes
- ▶ Defines a hierarchy of abstractions in which a subclass inherits from one or more superclasses
 - Single inheritance
 - Multiple inheritance
- ▶ Generalization is an “is-a-kind of” relationship

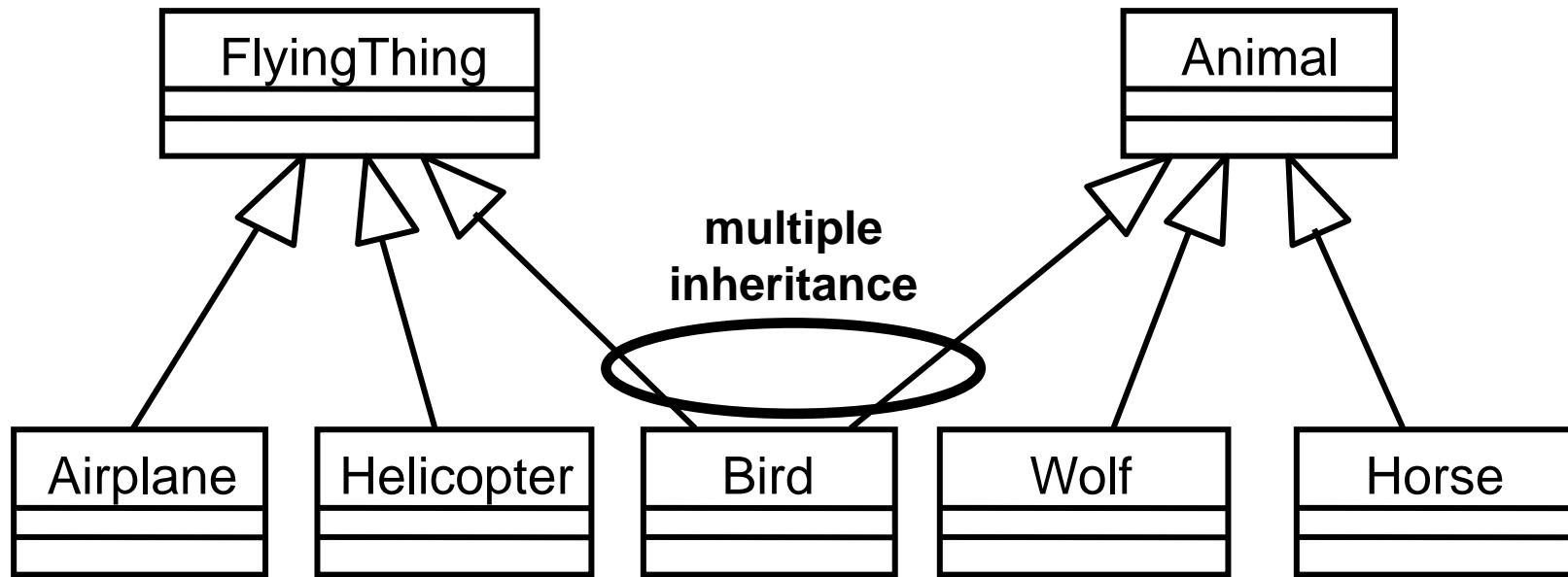
Example: Single Inheritance

- sometimes conceptual classes are (sub) types of another class:



Example: Multiple Inheritance

A class can inherit from several other classes ➤



Use multiple inheritance only when needed, and always with caution !

What Gets Inherited?

- ▶ A subclass inherits its parent's attributes, operations, and relationships
- ▶ A subclass may:
 - Add additional attributes, operations, relationships
 - Redefine inherited operations (use caution!)
- ▶ Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy

Inheritance leverages the similarities among classes

Associations

- ▶ we should model the important relationships between conceptual classes
 - not *every relationship, that would create clutter and confusion*
 - not too few, we want a useful model
- ▶ for each association, provide:
 - a short yet meaningful text label
 - the *multiplicity*

Associations

- Shows relationship between classes
- A class diagram may show:

Relationship	
Generalization (inheritance)	 "is a" "is a kind of"
Association (dependency)	 "Who does What" "uses"
Aggregation	 "has"
Composition: Strong aggregation	 "composed of"

Example: Library System

- ▶ Consider the world of libraries. A library has books, videos, and CDs that it loans to its users. All library material has a id# and a title. In addition, books have one or more authors, videos have one producer and one or more actors, while CDs have one or more entertainers. The library maintains one or more copies of each library item (book, video or CD). Copies of all library material can be loaned to users. Reference-only material is loaned for 2hrs and can't be removed from the library. Other material can be loaned for 2 weeks. For every loan, the library records the user, the loan date and time, and the return date and time. For users, the library maintains their name, address and phone number.

- ▶ Define the two main actors.
- ▶ Identify use cases by providing the actors, use case names. Draw the use case diagram.
- ▶ Create the conceptual class diagram.

Example: Digital Music players

Draw a UML Class Diagram representing the following elements from the problem domain for digital music players: An artist is either a band or a musician, where a band consists of two or more musicians. Each song has an artist who wrote it, and an artist who performed it, and a title. Therefore, each song is performed by exactly one artist, and written by exactly one artist. An album is composed of a number of tracks, each of which contains exactly one song. A song can be used in any number of tracks, because it could appear on more than one album (or even more than once on the same album!). A track has bitrate and duration. Because the order of the tracks on an album is important, the system will need to know, for any given track, what the next track is, and what the previous track is.

Draw a class diagram for this information, and be sure to label all the associations (relationships) with appropriate multiplicities.

References & Further Reading

- ▶ *Applying UML & Patterns (Larman 2007), Chapters 6, 9.*
- ▶ *Object-Oriented Systems Analysis and Design (Bennett et al, Third Edition, 2006), Chapter 6, 7.*

Object Oriented Analysis and Design Using the UML

Analysis: Dynamic Modeling

Dynamic Modeling with UML

- 1) represent how objects behave when you put them to work using the structure already defined in structural diagrams
- 2) model how the objects communicate in order to accomplish tasks within the operation of the system
- 3) describe how the system:
 - a) responds to actions from the users
 - b) maintains internal integrity
 - c) moves data
 - d) creates and manipulates objects, etc.
- 4) describe discrete pieces of the system, such as individual **scenarios** or operations

Note: not all system behaviour have to be specified - simple behaviours may not need a visual explanation of the communication required to accomplish them.

Dynamic Modeling with UML

▶ Diagrams for dynamic modeling

- *Interaction diagrams* describe the dynamic behavior between objects
- *Statecharts* describe the dynamic behavior of a single object

▶ Interaction diagrams

- Sequence Diagram:
 - Dynamic behavior of a set of objects arranged in time sequence.
 - Good for real-time specifications and complex scenarios
- Collaboration Diagram :
 - Shows the relationship among objects. Does not show time

▶ State Chart Diagram:

- A state machine that describes the response of an object of a given class to the receipt of outside stimuli (Events).
- *Activity Diagram*: A special type of statechart diagram, where all states are action states (Moore Automaton)

Sequence Diagram

- ▶ From the flow of events in the use case or scenario proceed to the sequence diagram
- ▶ A sequence diagram is a graphical description of objects participating in a use case or scenario using a DAG (direct acyclic graph) notation
- ▶ Relation to object identification:
 - Objects/classes have already been identified during object modeling
 - Objects are identified as a result of dynamic modeling
- ▶ Heuristic:
 - An event always has a sender and a receiver.
 - The representation of the event is sometimes called a message
 - Find them for each event => These are the objects participating in the use case

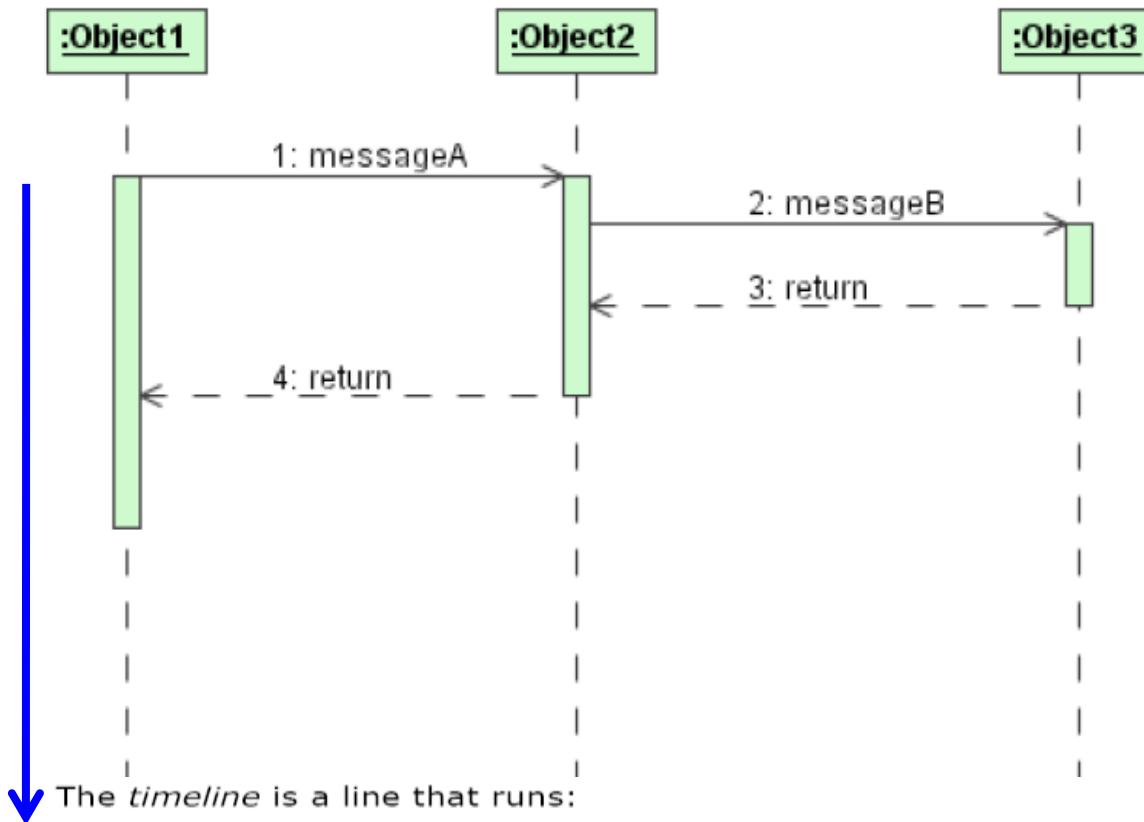
Sequence Diagram

A sequence diagram shows interactions between objects.

Components of sequence diagrams:

- 1) object lifelines
 - a) object
 - b) timeline
- 2) messages
 - a) message, stimulus
 - b) signal, exception
 - c) operations, returns
 - d) identification
- 3) message syntax

Sequence Diagram Example



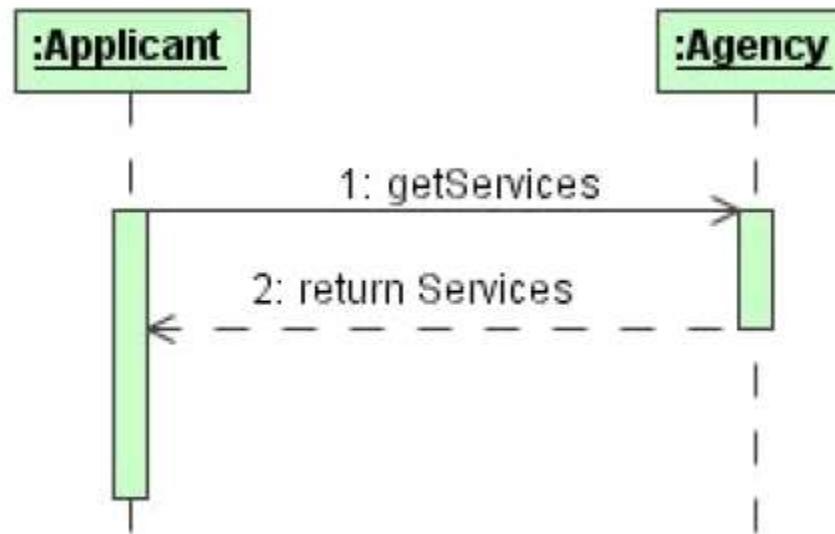
1. from the beginning of a scenario at the top of the diagram
2. to the end of the scenario at the bottom of the diagram.

Layout:

- 1st column:** Should correspond to the actor who initiated the use case
- 2nd column:** Should be a boundary object
- 3rd column:** Should be the control object that manages the rest of the use case

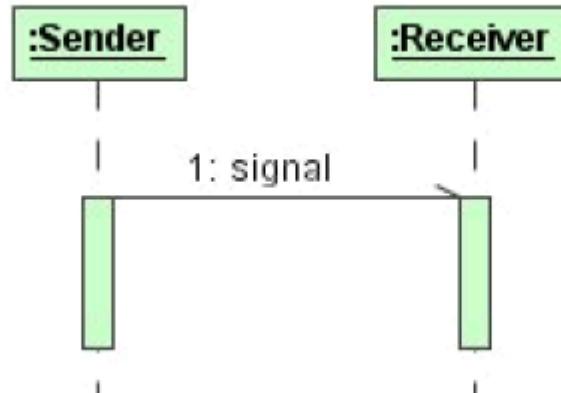
Operations and Returns Example

Example: The *Applicant* object sends a message to the *Agency* object to get the list of services it provides. The *Agency* object returns this information.



Signal

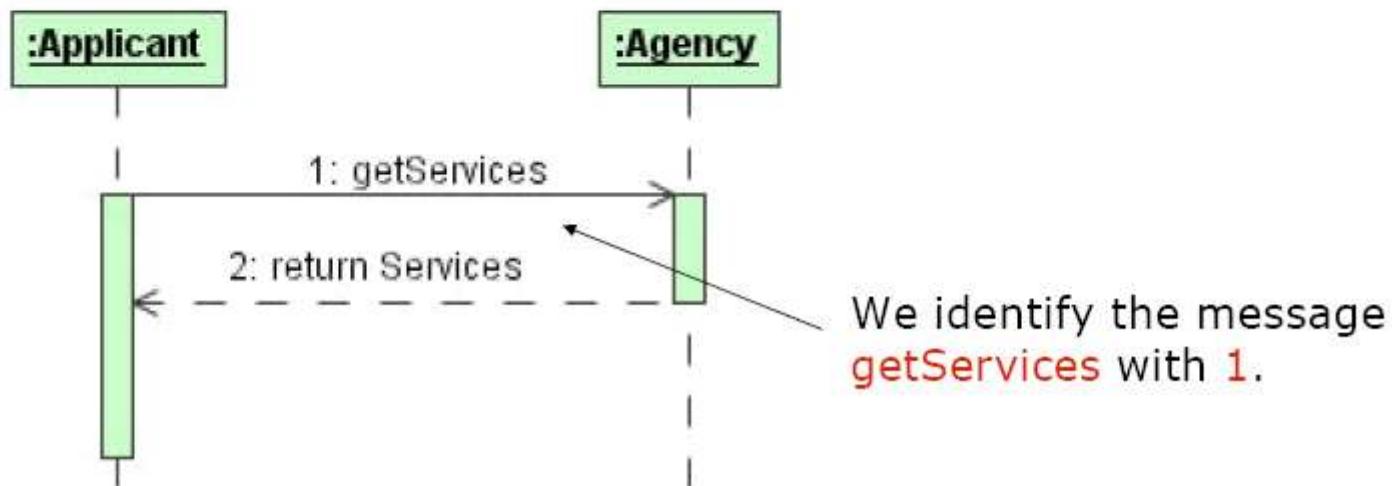
- 1) an object may raise a signal through a message
- 2) a **signal** is a special type of a class associated with an event that can trigger a procedure within the receiving object
- 3) a signal does not require a return from the receiving object



Identification of Messages

A message number or name is used to identify messages.

Example:



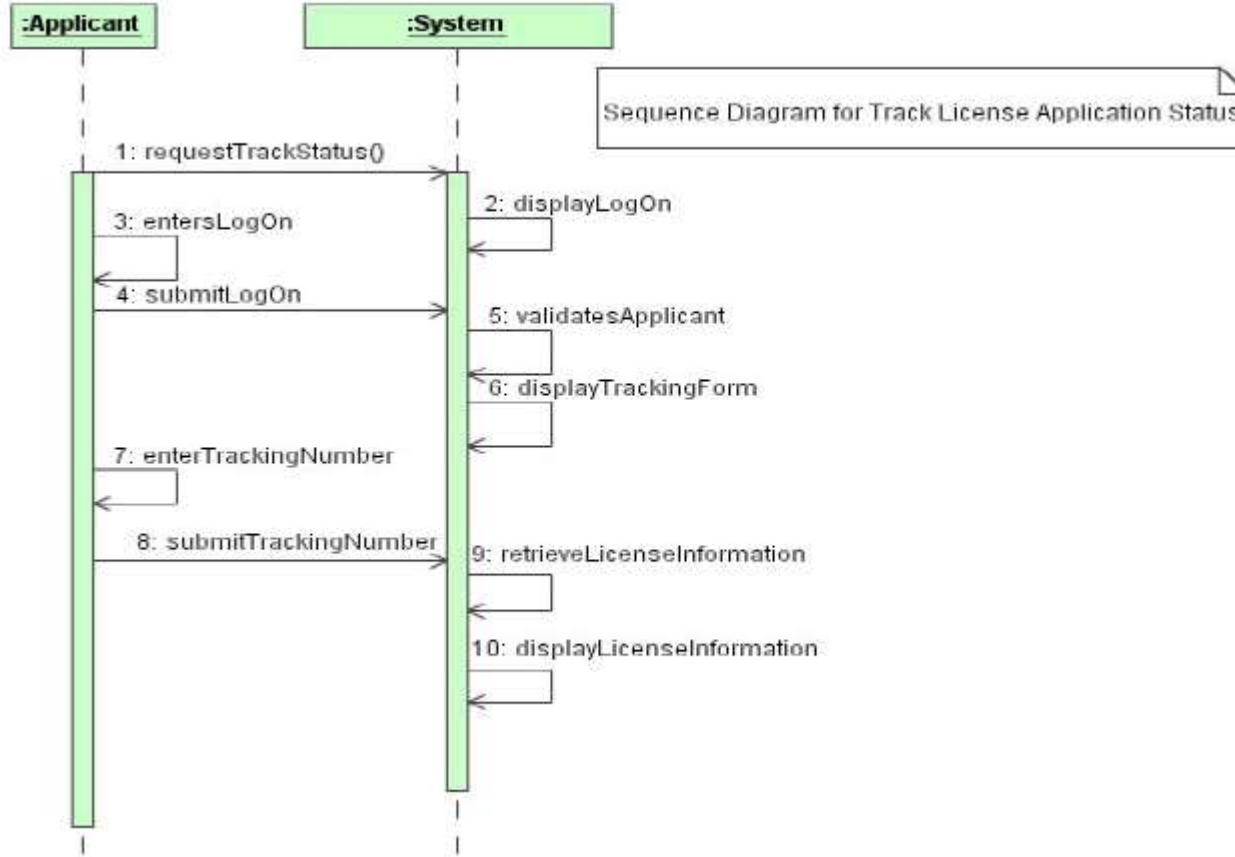
Example Scenario

Recall the scenario: an applicant tracks the status of a license application and the system displays the license information.

Procedure:

1. Applicant requests to track the status of a license application
2. System displays the logon form
3. Applicant enters the logon information
4. Applicant submits the logon information
5. System validates the applicant
6. System displays the form to enter the tracking number
7. Applicant enters the tracking number
8. Applicant submits the tracking number
9. System retrieves the license information
10. System displays the license information

Example Sequence Diagram

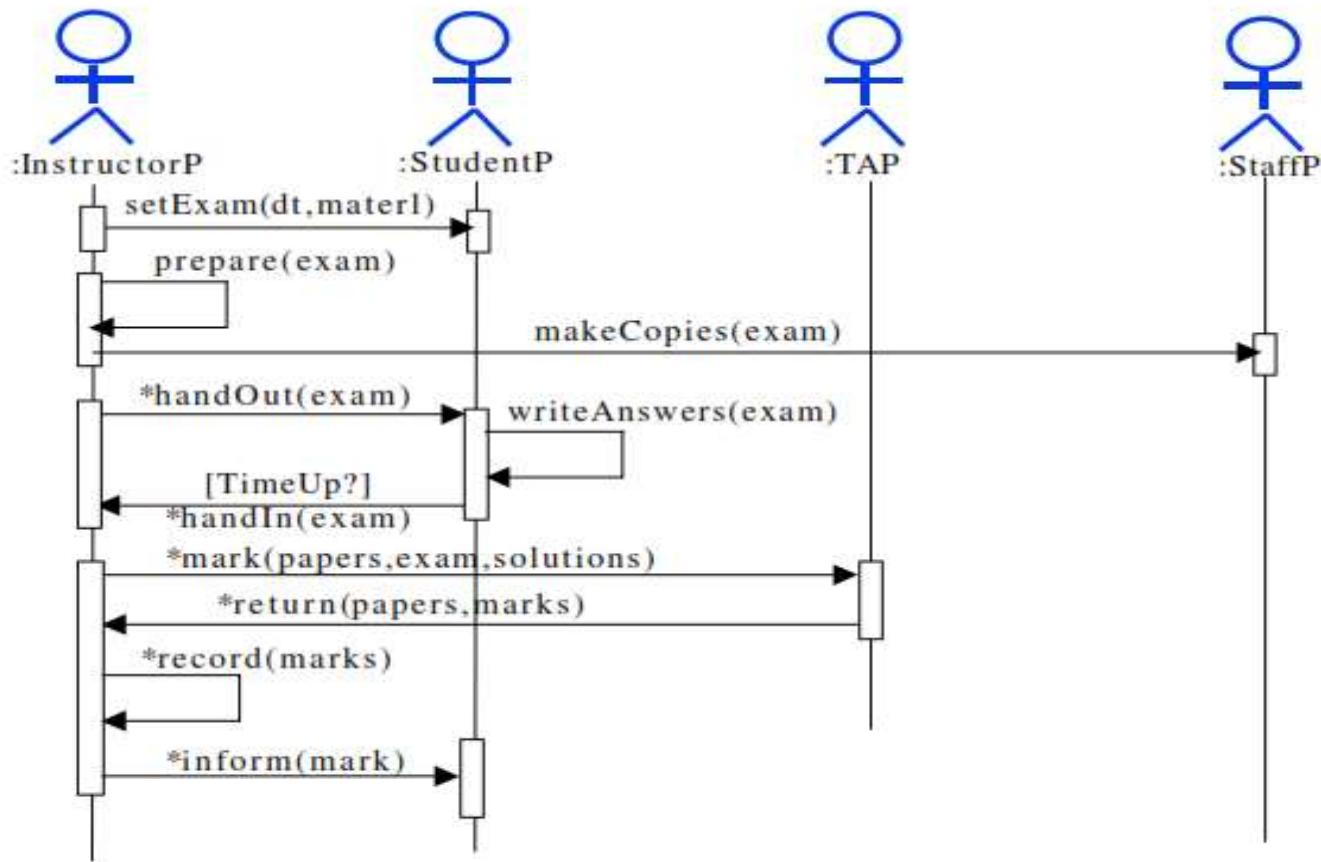


Example Scenario

Exercise

- ▶ To give an exam, an instructor first notifies the students of the exam date and the material to be covered. She then prepares the exam paper (with sample solutions), gets it copied to produce enough copies for the class, and hands it out to students on the designated time and location. The students write their answers to exam questions and hand in their papers to the instructor. The instructor then gives the exam papers to the TAs, along with sample solutions to each question, and gets them to mark it. She then records all marks and returns the papers to the students.
- ▶ Draw a sequence diagram that represents this process. Make sure to show when each actor is participating in the process. Also, show the operation that is carried out during each interaction, and what its arguments are.

Example Scenario

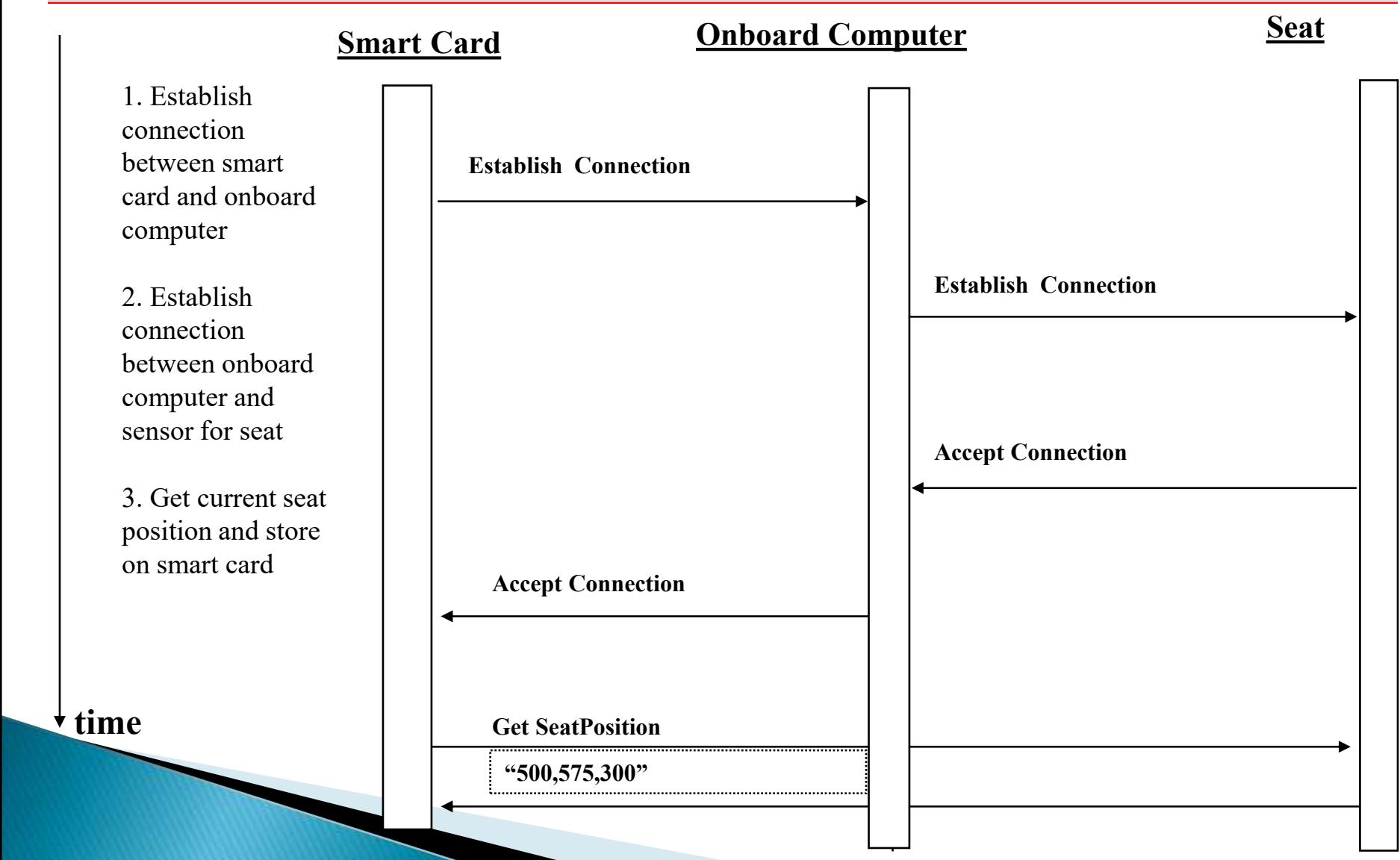


Example Scenario

GetSeatPosition: Passenger tries to find an empty seat in a train using an onboard computer connected to seat sensors and a smart card.

- ▶ Flow of events in a “Get SeatPosition” use case :
 1. Establish connection between smart card and onboard computer
 2. Establish connection between onboard computer and sensor for seat
 3. Get current seat position and store on smart card
- ▶ Which are the objects?

Sequence Diagram for “Get SeatPosition”



Normal Flow of Events: For withdrawal of cash

- ▶ 1.(SR) The ATM asks the user to insert a card.
- ▶ 2.(AA) The user inserts a cash card.
- ▶ 3.(SR) The ATM accepts the card and reads its serial number.
- ▶ 4.(SR) The ATM requests the password.
- ▶ 5.(AA) The user enters 1234.
- ▶ 6.(SR) The ATM verifies the serial number and password with the bank and gets the notification accordingly.
- ▶ 7.(SA)The ATM asks the user to select the kind of transaction.
- ▶ 8.(AA)User selects the withdrawal.
- ▶ 9.(SR)The ATM asks for the amount of cash; user enters Rs. 2500/-
- ▶ 10.(SR)The ATM verifies that the amount of cash is within predefined policy limits and asks the bank, to process the transaction which eventually confirms success and returns the new account balance.

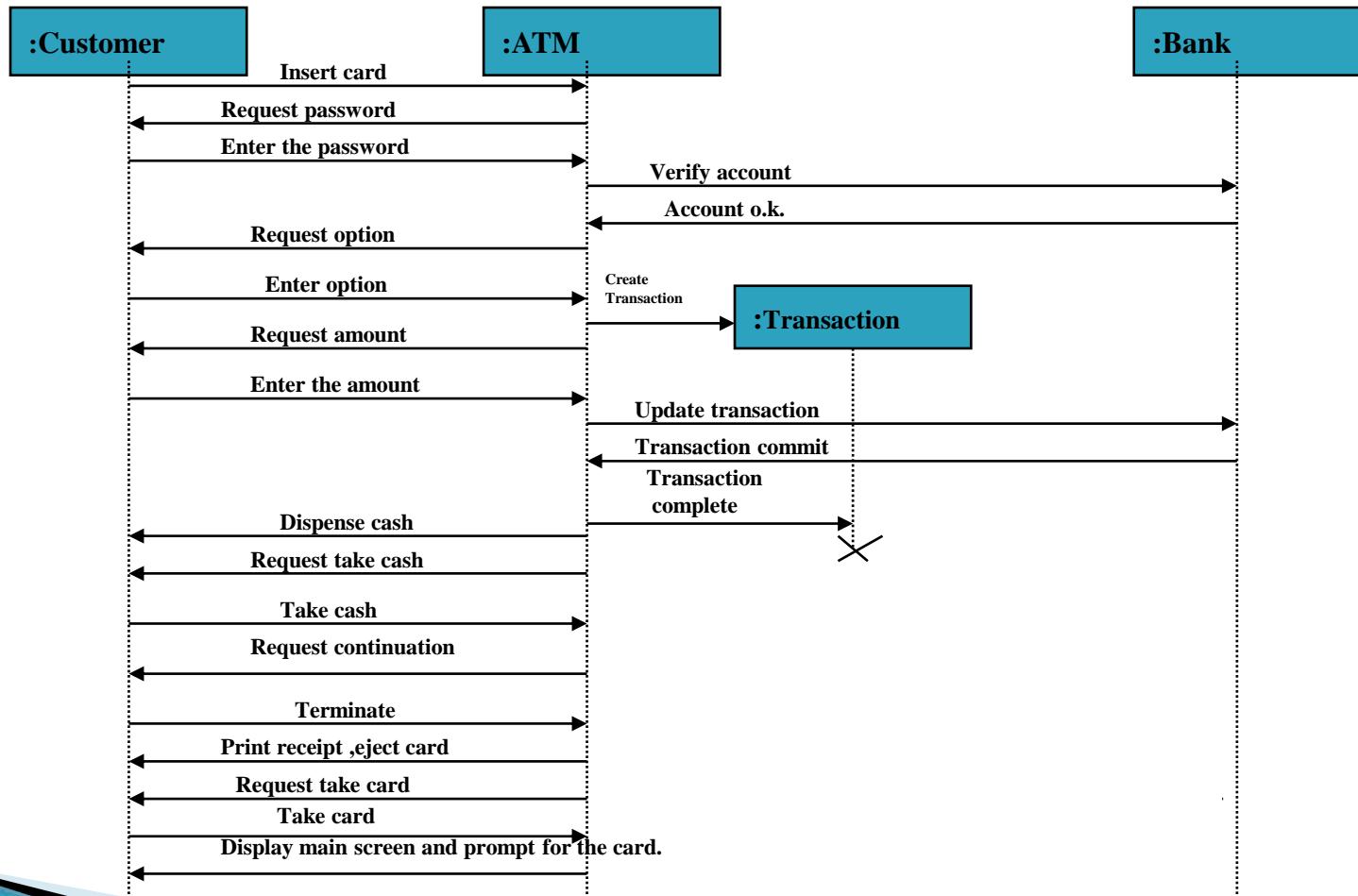
Normal Flow of Events: For withdrawal of cash

- ▶ 11.(SR) The ATM dispenses cash and asks the user to take it.
- ▶ 12.(AA) The user takes the cash.
- ▶ 13.(SR) The ATM asks whether the user wants to continue.
- ▶ 14.(AA) The user indicates no.
- ▶ 15.(SR) The ATM prints a receipt, ejects the card and asks the user to take them
- ▶ 16.(AA) The user takes the receipt and the card.
- ▶ 17.(SR) The ATM asks a user to insert a card.

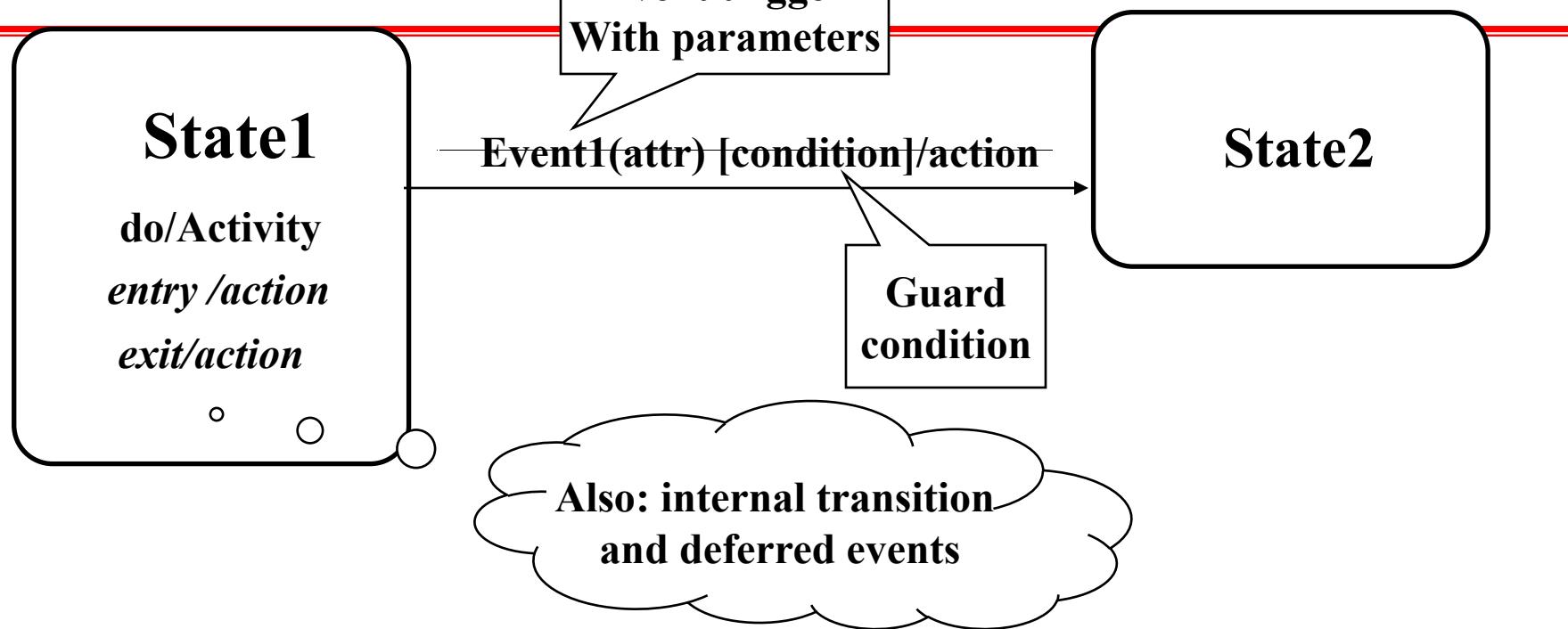
Alternative Flow of Events: For withdrawal of cash

- ▶ 9. The ATM asks for the amount of cash; the user has change of mind and hits the “cancel”.

Sequence diagram [for withdrawal of cash, normal flow]



UML Statechart Diagram Notation



- ▶ Notation based on work by Harel
 - Added are a few object-oriented modifications
- ▶ A UML statechart diagram can be mapped into a finite state machine

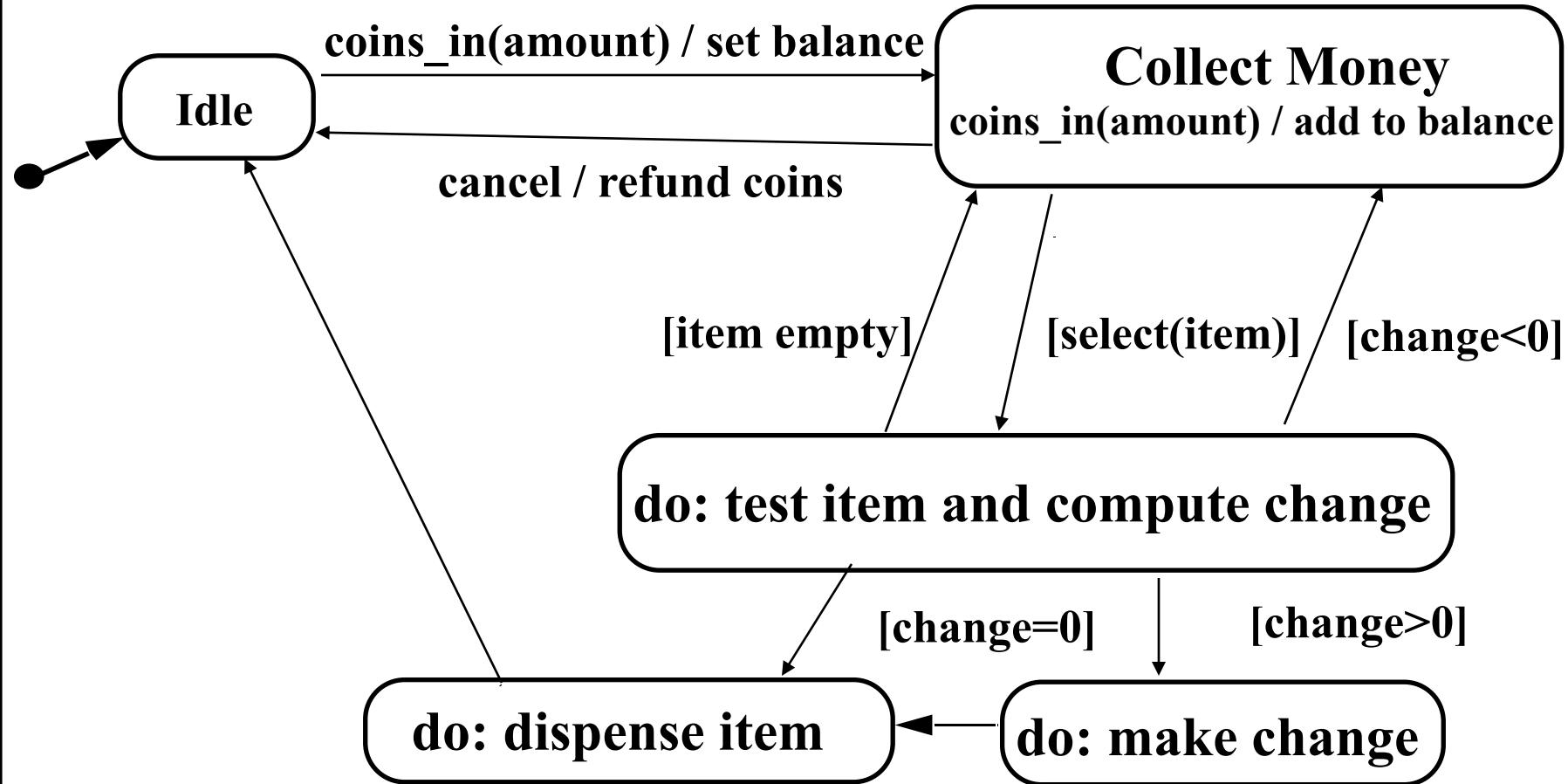
Statechart Diagrams

- ▶ Graph whose nodes are states and whose directed arcs are transitions labeled by event names.
- ▶ We distinguish between two types of operations in statecharts:
 - Activity: Operation that takes time to complete
 - associated with states
 - Action: Instantaneous operation
 - associated with events
 - associated with states (reduces drawing complexity): Entry, Exit, Internal Action
- ▶ A statechart diagram relates events and states for *one class*
 - An object model with a set of objects has a set of state diagrams

State

- ▶ An abstraction of the attributes of a class
 - State is the aggregation of several attributes a class
- ▶ Basically an equivalence class of all those attribute values and links that do no need to be distinguished as far as the control structure of the system is concerned
 - Example: State of a bank
 - A bank is either solvent or insolvent
- ▶ State has duration

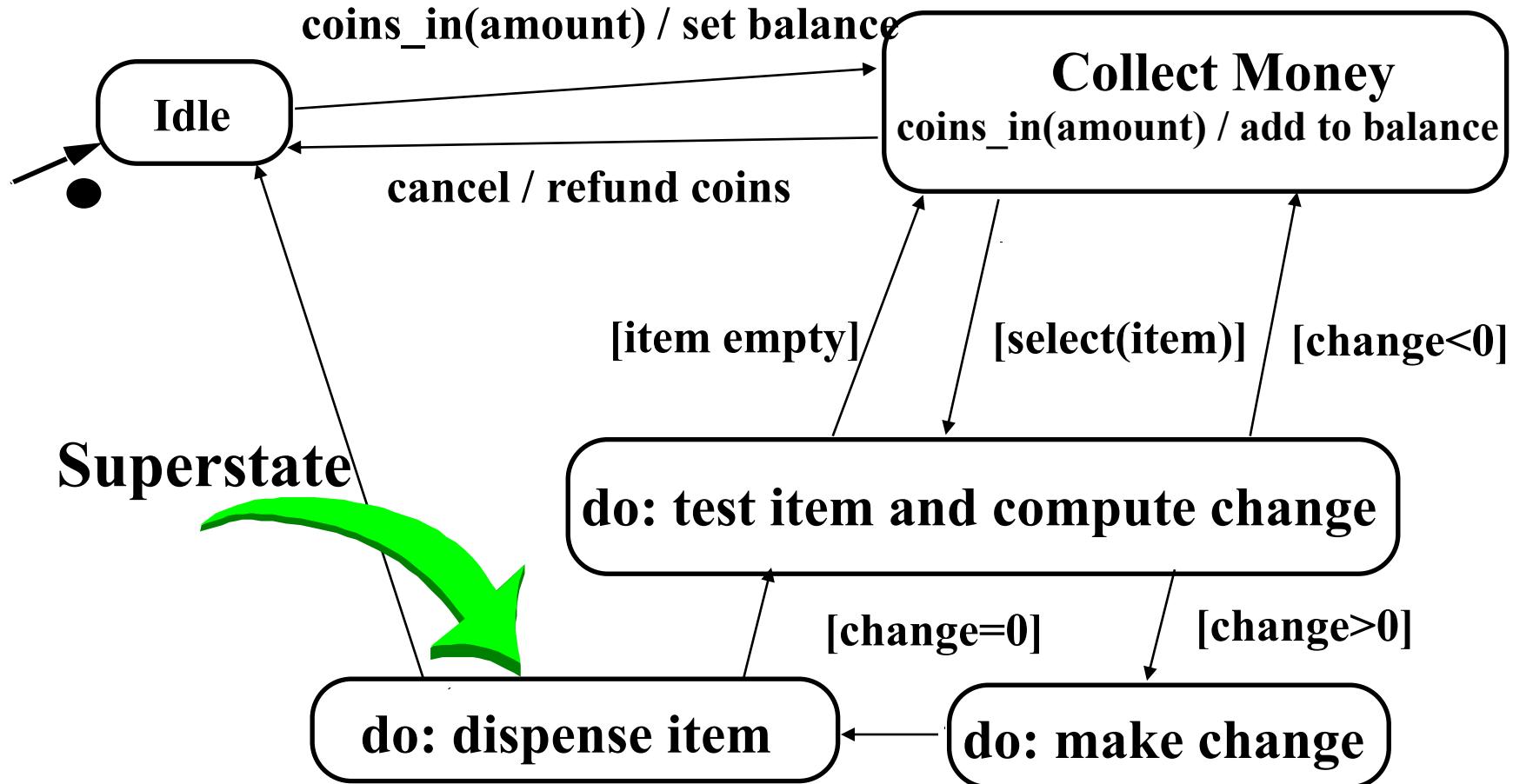
Example of a StateChart Diagram



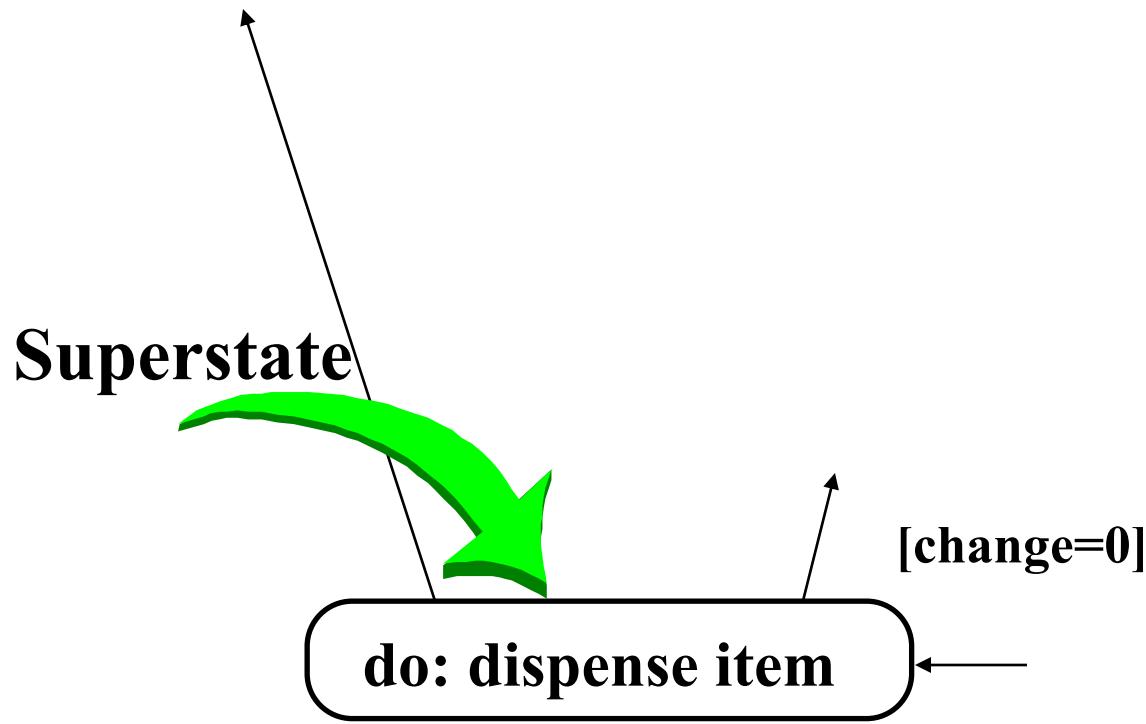
Nested State Diagram

- ▶ Activities in states are composite items denoting other lower-level state diagrams
- ▶ A lower-level state diagram corresponds to a sequence of lower-level states and events that are invisible in the higher-level diagram.
- ▶ Sets of substates in a nested state diagram denote a **superstate** are enclosed by a large rounded box, also called contour.

Example of a Nested Statechart Diagram

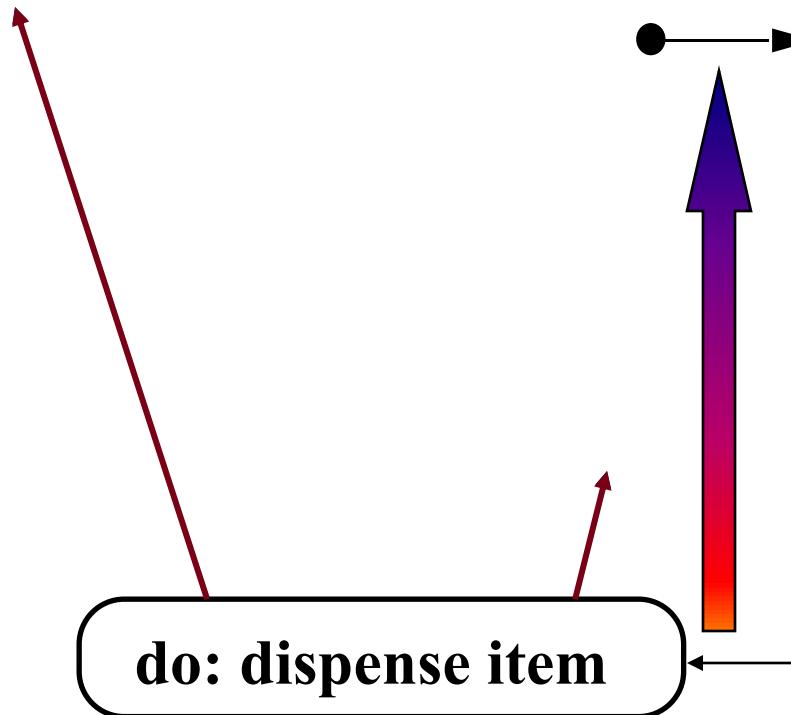


Example of a Nested Statechart Diagram

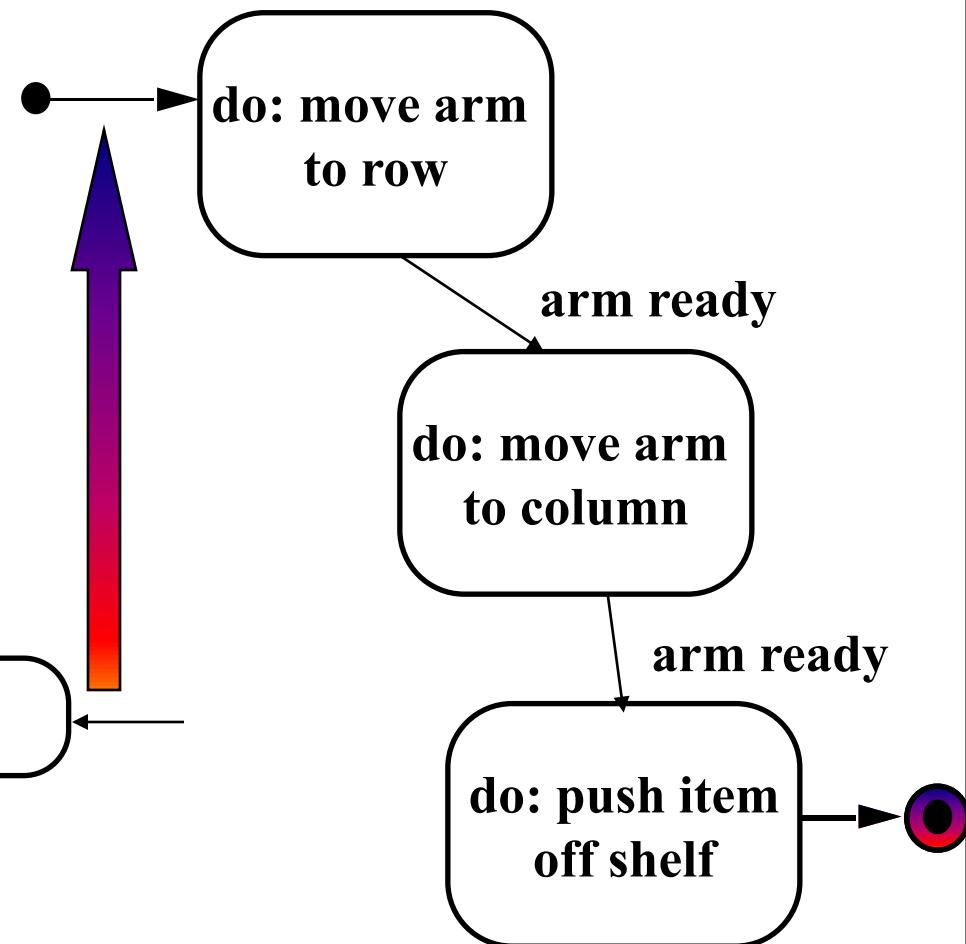


Example of a Nested Statechart Diagram

‘Dispense item’ as an atomic activity:

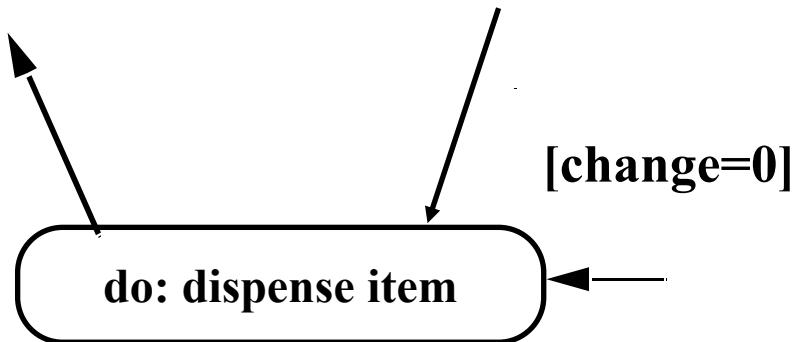


‘Dispense item’ as a composite activity:

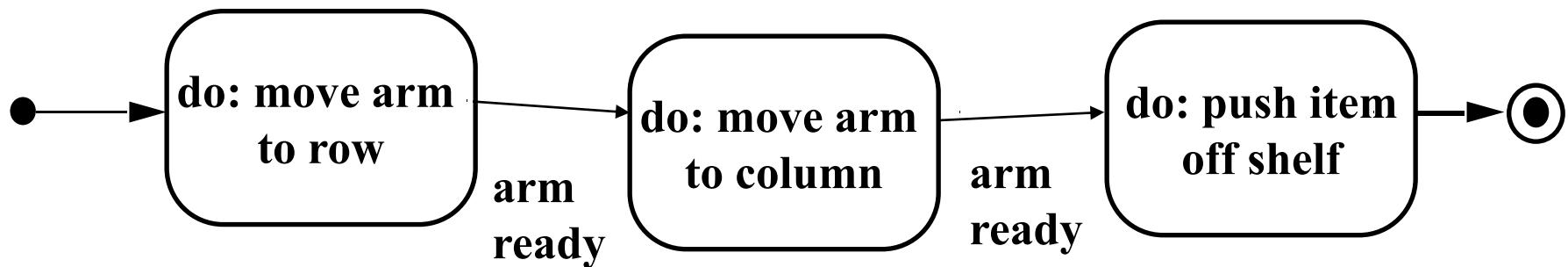


Expanding activity “do:dispense item”

‘Dispense item’ as an atomic activity:



‘Dispense item’ as a composite activity:



Superstates

- ▶ Goal:
 - Avoid spaghetti models
 - Reduce the number of lines in a state diagram
- ▶ Transitions from other states to the superstate enter the first substate of the superstate.
- ▶ Transitions to other states from a superstate are inherited by all the substates (state inheritance)

State Chart Diagram vs Sequence Diagram

- ▶ State chart diagrams help to identify:
 - *Changes* to an individual object over time
- ▶ Sequence diagrams help to identify
 - The *temporal relationship* of between objects over time
 - *Sequence of operations* as a response to one ore more events

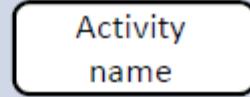
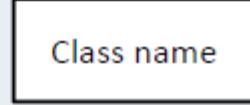
UML : ACTIVITY DIAGRAM



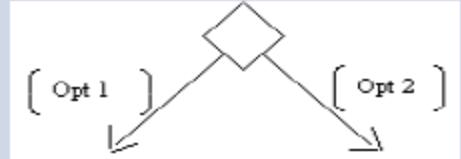
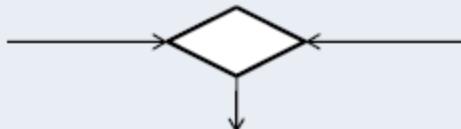
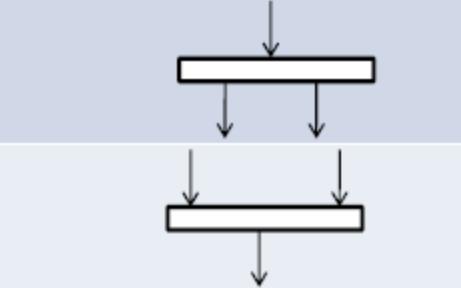
UML : ACTIVITY DIAGRAM

1. Activity diagrams are the object-oriented equivalent of flow charts and data-flow diagrams from structured development.
2. Activity diagrams describe the workflow behavior of a system.
3. The process flows in the system are captured in the activity diagram.
4. Activity diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity.

Elements of Activity Diagram

Description	Symbol
Activity : Is used to represent a set of actions	
A Control Flow: Shows the sequence of execution	
An Object Flow: Shows the flow of an object from one activity (or action) to another activity (or action).	
An Initial Node: Portrays the beginning of a set of actions or activities	
A Final-Activity Node: Is used to stop all control flows and object flows in an activity (or action)	
An Object Node: Is used to represent an object that is connected to a set of Object Flows.	

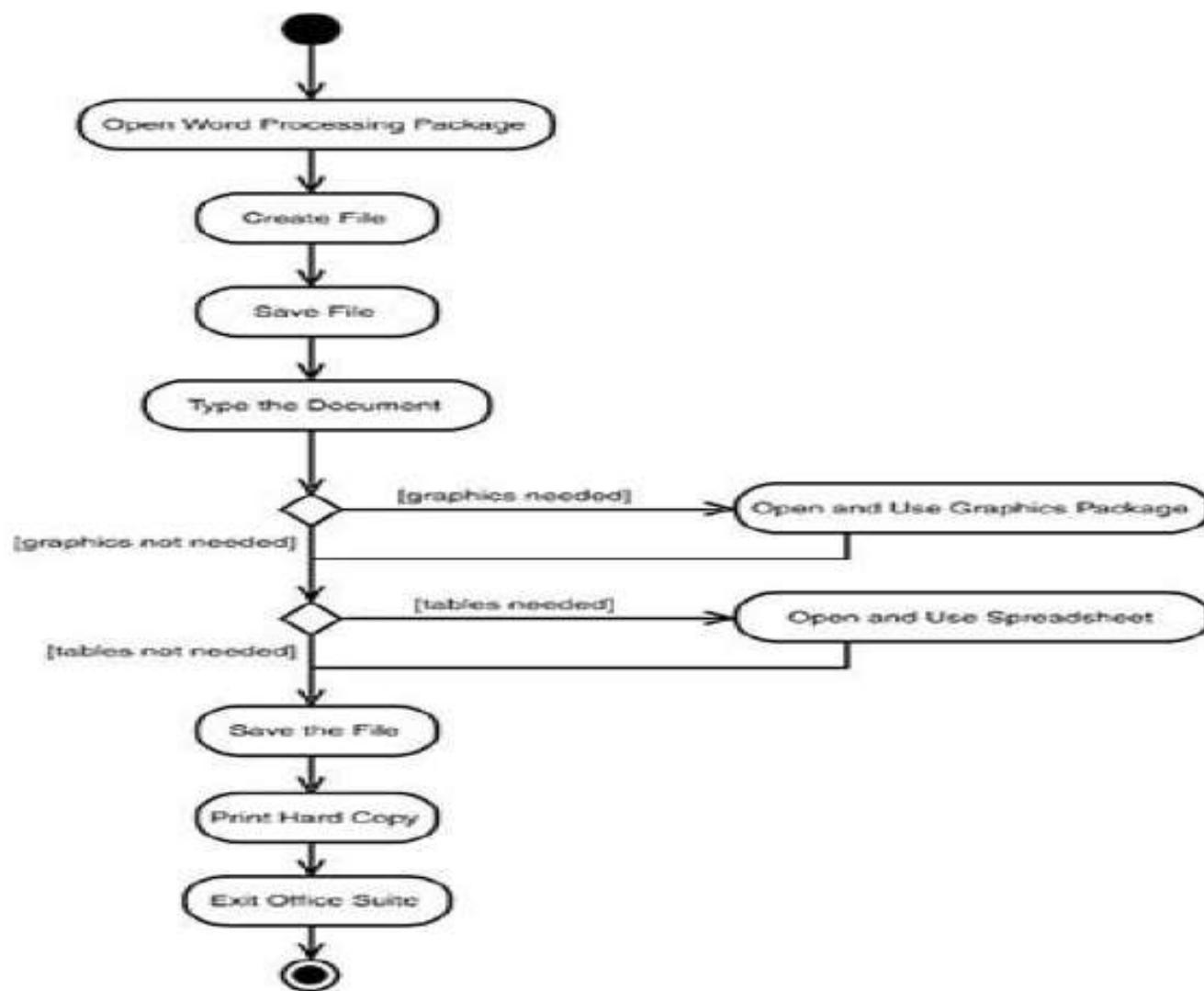
Elements of Activity Diagram

Description	symbol
A Decision Node: Is used to represent a test condition to ensure that the control flow or object flow only goes down one path	
A Merge Node: Is used to bring back together different decision paths that were created using a decision-node.	
A Fork Node: Is used to split behavior into a set of parallel or concurrent flows of activities (or actions)	
A Join Node: Is used to bring back together a set of parallel or concurrent flows of activities (or actions).	
A Swimlane :A swimlane is a way to group activities performed by the same actor on an activity diagram or to group activities in a single thread	

Example 1: Creating document

1. Open the word processing package.
2. Create a file.
3. Save the file under a unique name within its directory.
4. Type the document.
5. If graphics are necessary, open the graphics package, create the graphics, and paste the graphics into the document.
6. If a spreadsheet is necessary, open the spreadsheet package, create the spreadsheet, and paste the spreadsheet into the document.
7. Save the file.
8. Print a hard copy of the document.
9. Exit the word processing package.

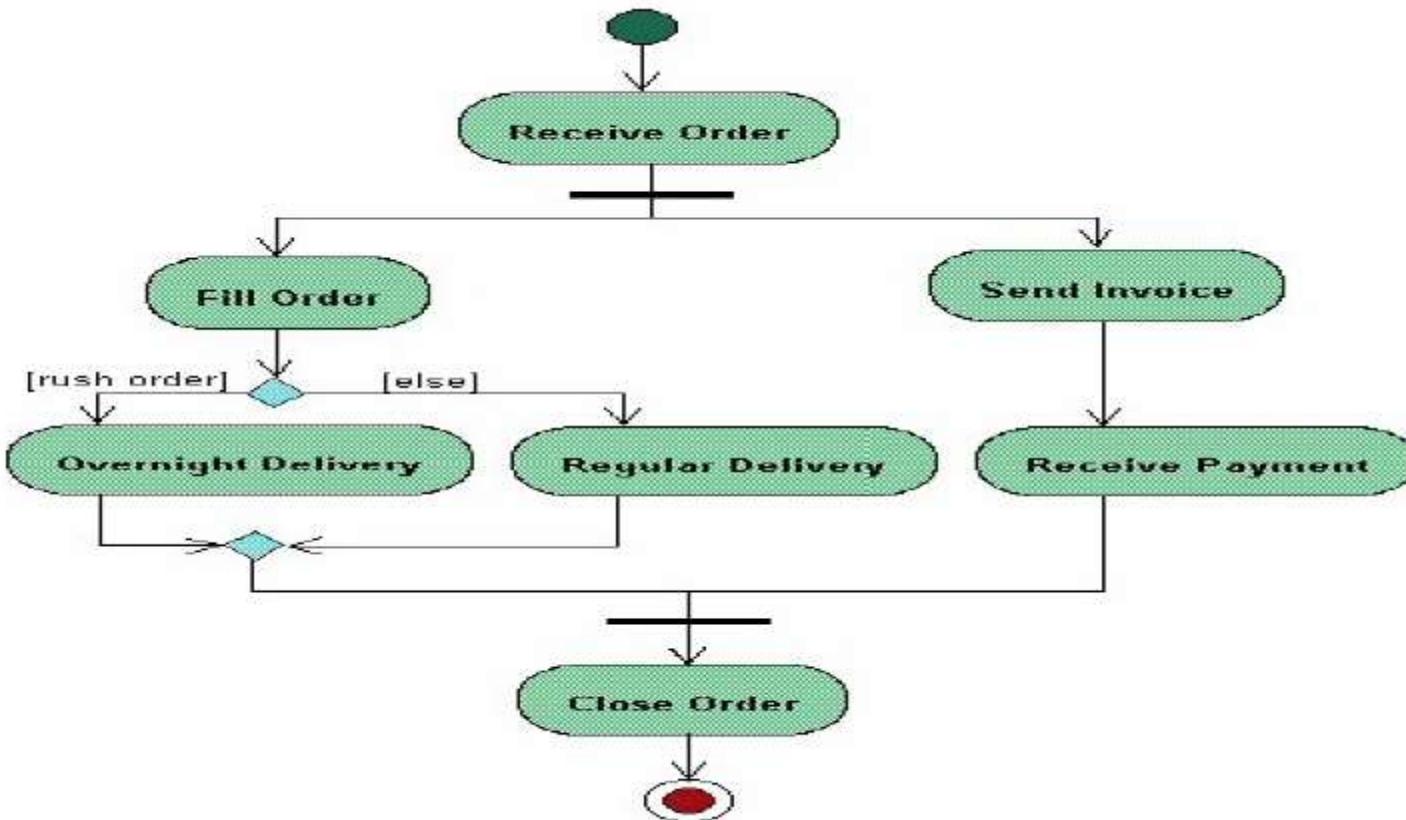
Example 1: Creating document



Example 2: Process Order

Once the order is received the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing. On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed. Finally the parallel activities combine to close the order.

Example 2: Process Order



Example 3: Enrollment in university

1. An applicant wants to enroll in the university.
2. The applicant hands a filled out copy of form *U113 University Application Form* to the registrar.
3. The registrar inspects the forms.
4. The registrar determines that the forms have been filled out properly.
5. The registrar informs student to attend in university overview presentation.
6. The registrar helps the student to enroll in seminars
7. The registrar asks the student to pay the initial.

Guards

A guard is a condition that must be true in order to traverse a transition.

1. **Each Transition Leaving a Decision Point Must Have a Guard .** This ensures that you have thought through all possibilities for that decision point.
2. **Guards Should Not Overlap.** For example guards such as $x < 0$, $x = 0$, and $x > 0$ are consistent whereas guard such as $x \leq 0$ and $x \geq 0$ are not consistent because they overlap – it isn't clear what should happen when x is 0.
3. **Guards on Decision Points Must Form a Complete Set.** For example, guards such as $x < 0$ and $x > 0$ are not complete because it isn't clear what happens when x is 0.

Parallel Activities guidelines

1. It is possible to show that activities can occur in parallel, as you see in example3 depicted using two parallel bars. The first bar is called a fork, it has one transition entering it and two or more transitions leaving it. The second bar is a join, with two or more transitions entering it and only one leaving it.
2. A Fork Should Have a Corresponding Join. In general, for every start (fork) there is an end (join). In UML 2 it is not required to have a join, but it usually makes sense.
3. Forks Have One Entry Transition.
4. Joins Have One Exit Transition
5. Avoid Superfluous Forks.

Swimlane Guidelines

A swimlane is a way to group activities performed by the same actor on an activity diagram or to group activities in a single thread.

Actions may be grouped into swimlanes to denote the object or subsystem that implements the actions.

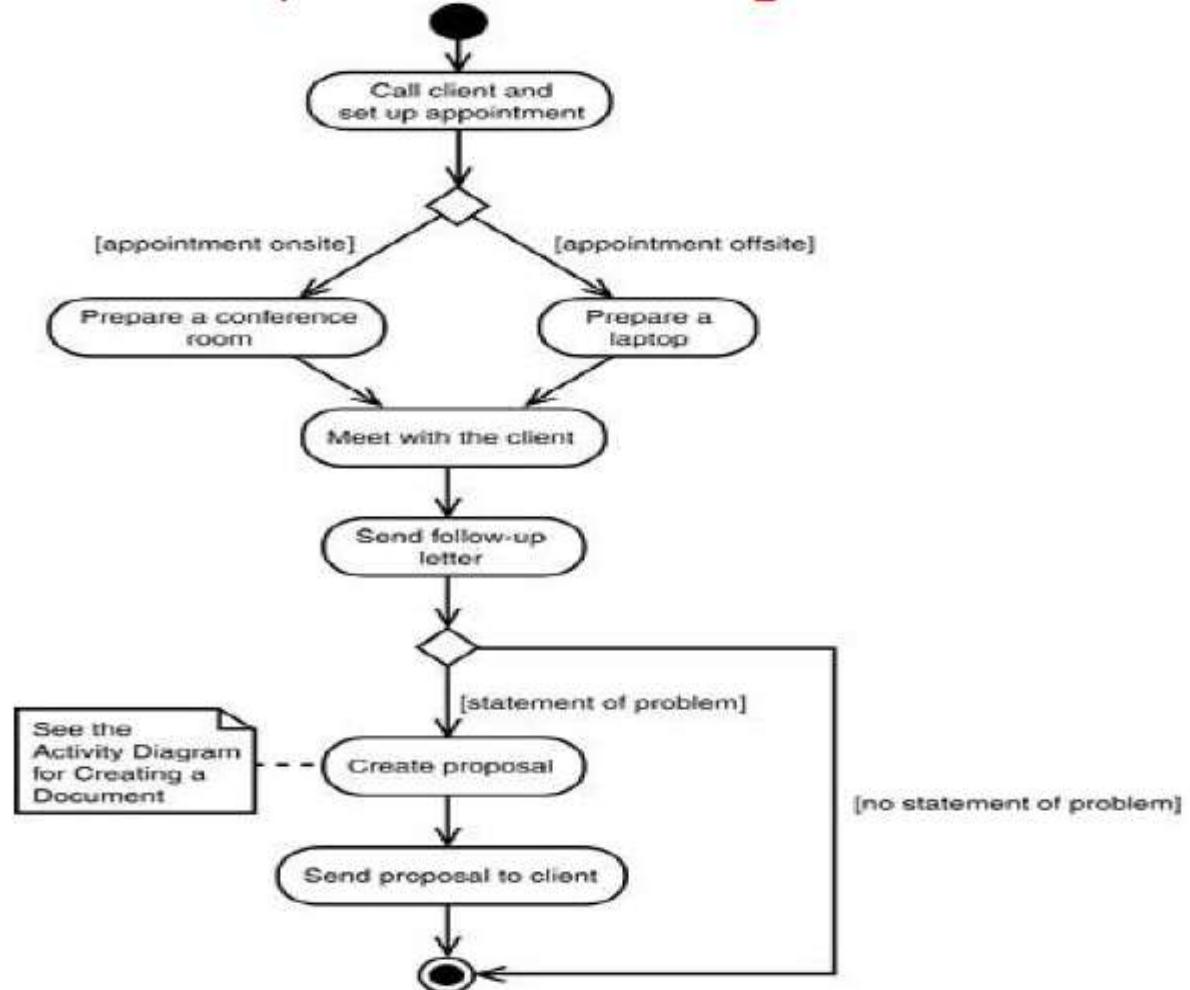
- ▶ Order Swimlanes in a Logical Manner
- ▶ Apply Swimlanes To Linear (sequential)
- ▶ Processes A good rule of thumb is that swimlanes are best applied to linear processes
- ▶ Have Less Than Five Swimlanes
- ▶ Consider Swimlanes For Complex Diagrams
- ▶ Swimlane Suggest The Need to Reorganize Into Smaller Activity Diagrams

Example 5: business process of meeting a new client

1. A salesperson calls the client and sets up an appointment.
2. If the appointment is onsite (in the consulting firm's office), corporate technicians prepare conference room for a presentation
3. If the appointment is offsite (at the client's office), a consultant prepares a presentation on a laptop.
4. The consultant and the salesperson meet with the client at the agreed-upon location and time.
5. The salesperson follows up with a letter
6. If the meeting has resulted in a statement of a problem, the consultant create a proposal and sends it to the client.

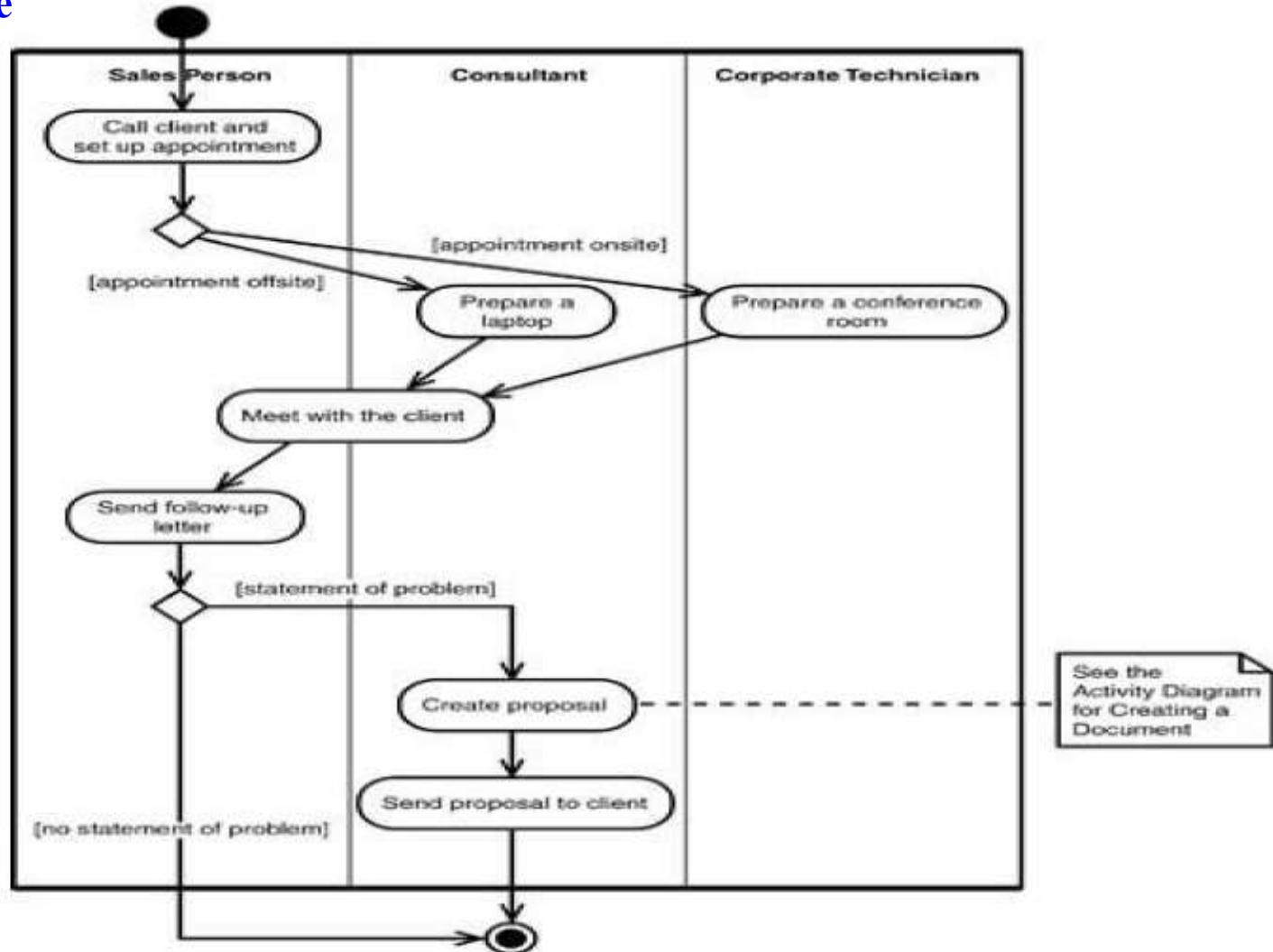
Example 5: business process of meeting a new client

Without Swimlane for
create a proposal



Example 5: business process of meeting a new client

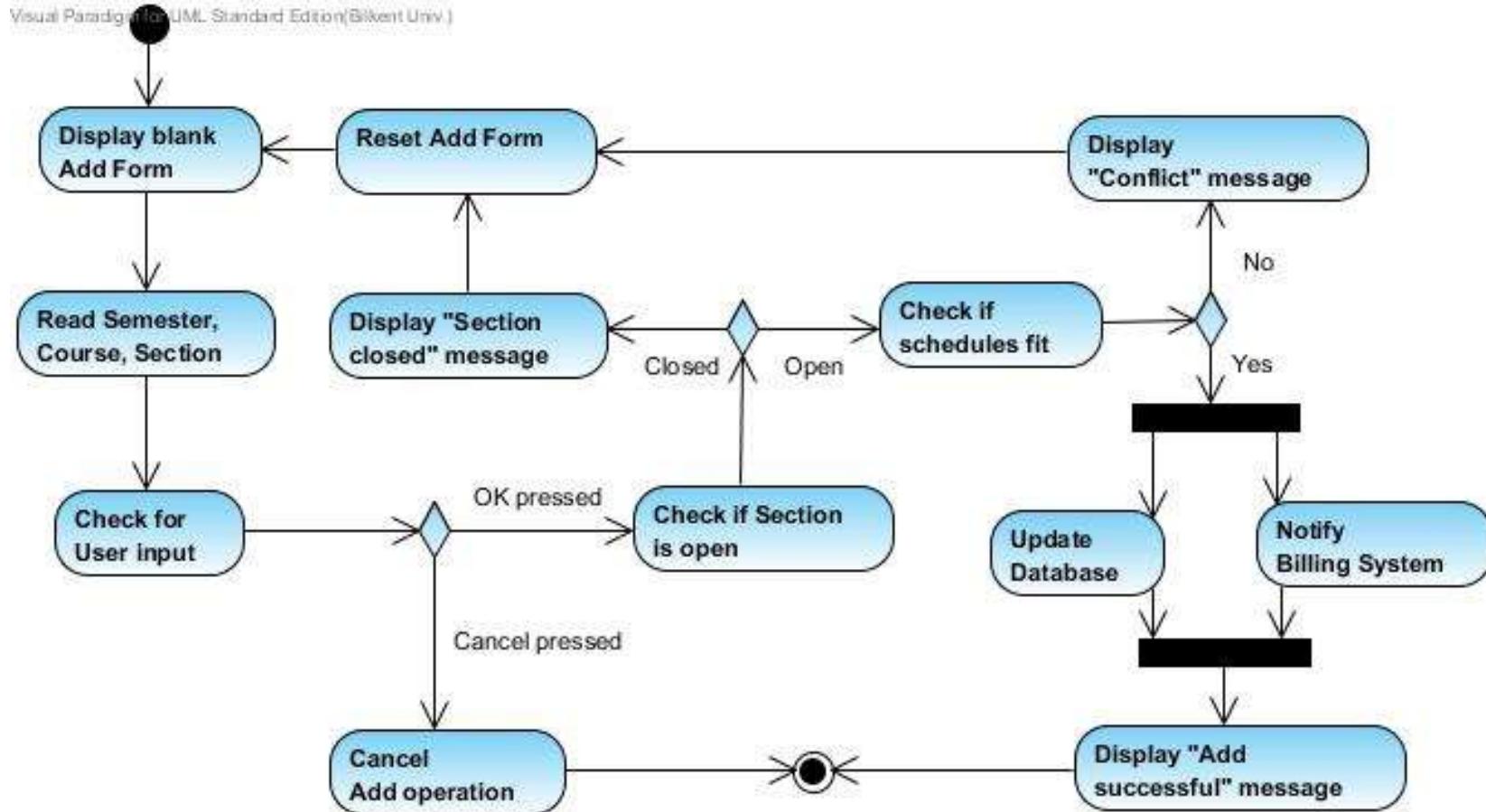
With Swimlane



Homework 1

A student wants to add a new course. The student fills out the form by specifying the semester, the course to take (department and course number) and the section, to which the student would like to be added. Then, the student clicks the OK button. The system checks whether the particular section is still open for registration and the maximum count hasn't been reached. If so, the system checks if the particular section of the added course fits the student's schedule. Add operation is not allowed when there are any conflicts in the schedule. **If there is no conflict**, the system updates the database and **simultaneously** notifies the billing system of the change. It **then** displays an appropriate message. The student may, of course, cancel the add operation at any point during this process.

Homework 1



Homework 2

Draw an activity diagram for the following problem:

Appointment system for doctor office.

1. A patient came to office, the scheduler get patient info.
2. If the patient is new the scheduler make new patient record.
3. The scheduler display list of possible appointments to patient.
4. Patient choose new appointments , modify appointments or cancel his appointments .
5. Patient make payment.

Hints:

There are about 6 to 8 activities and 2 to 5 objects.

Summary: Requirements Analysis

► 1. What are the transformations?



Functional Modeling

- Create *scenarios and use case diagrams*

- Talk to client, observe, get historical records, do thought experiments

2. What is the structure of the system?



Object Modeling

- Create *class diagrams*

- Identify objects.

- What are the associations between them? What is their multiplicity?

- What are the attributes of the objects?

- What operations are defined on the objects?

3. What is its behavior?



Dynamic Modeling

- Create *sequence diagrams*

- Identify senders and receivers

- Show sequence of events exchanged between objects. Identify event dependencies and event concurrency.

- Create *state diagrams*

- Only for the dynamically interesting objects.

Elements of Systems Design

King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

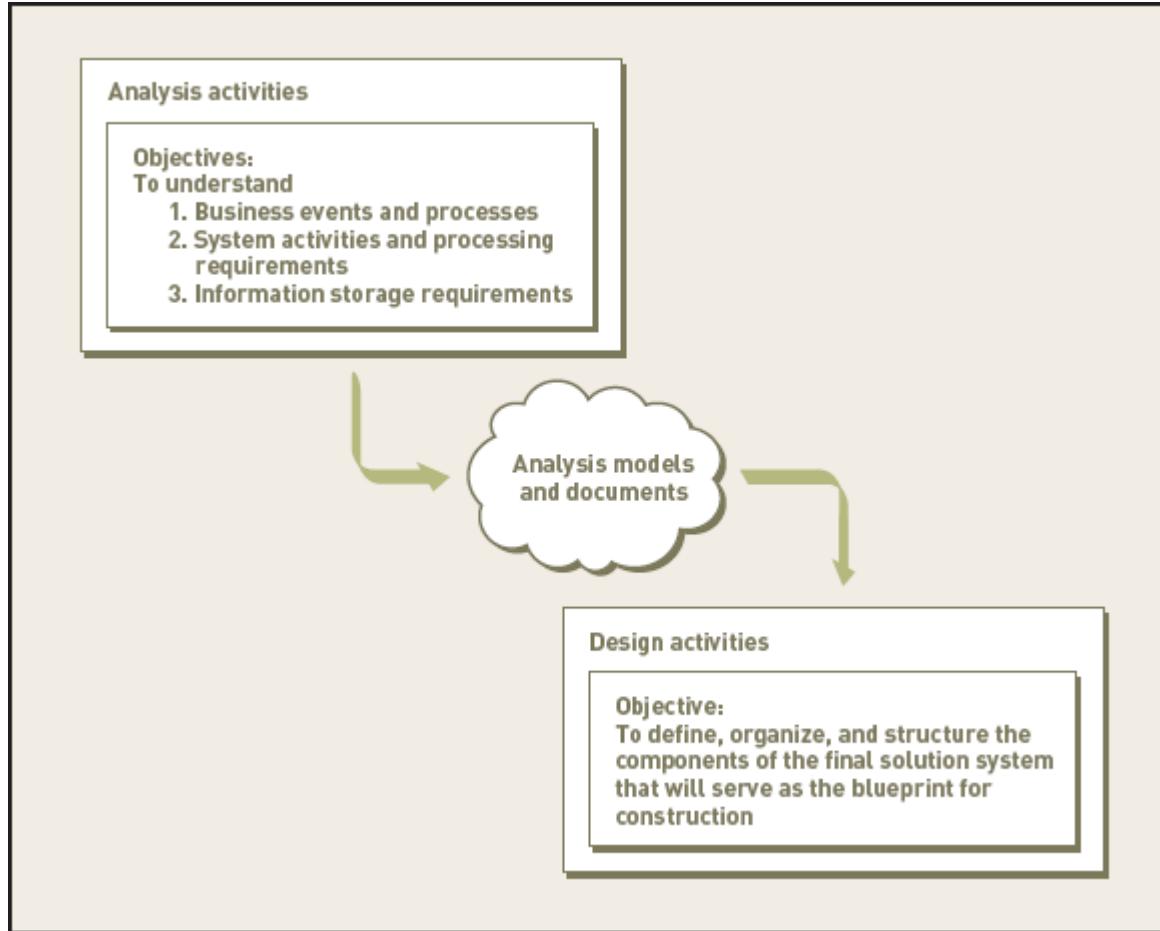
Objectives

- To Establish the overall structure of a software system
- To introduce architectural design and to discuss its importance
- To describe types of architectural model that may be used

Inputs for System Design

- Design
 - Converts functional models from analysis into models that represent the solution
 - Focused on technical issues
 - Requires less user involvement than analysis
- Design may use structured or OO approaches
 - Database can be relational, OO, or hybrid
 - User interface issues

Analysis versus Design



Design Phase Activities and Key Questions

Design activity	Key question
Design and integrate the network	<i>Have we specified in detail how the various parts of the system will communicate with each other throughout the organization?</i>
Design the application architecture and software	<i>Have we specified in detail how each system activity is actually carried out by the people and computers?</i>
Design the user interface(s)	<i>Have we specified in detail how all users will interact with the system?</i>
Design the system interface(s)	<i>Have we specified in detail how the system will work with all other systems inside and outside our organization?</i>
Design and integrate the database	<i>Have we specified in detail how and where the system will store all of the information needed by the organization?</i>
Prototype for design details	<i>Have we created prototypes to ensure all detailed design decisions have been fully understood?</i>
Design and integrate the system controls	<i>Have we specified in detail how we can be sure that the system operates correctly and the data maintained by the system is safe and secure?</i>

Architectural Design

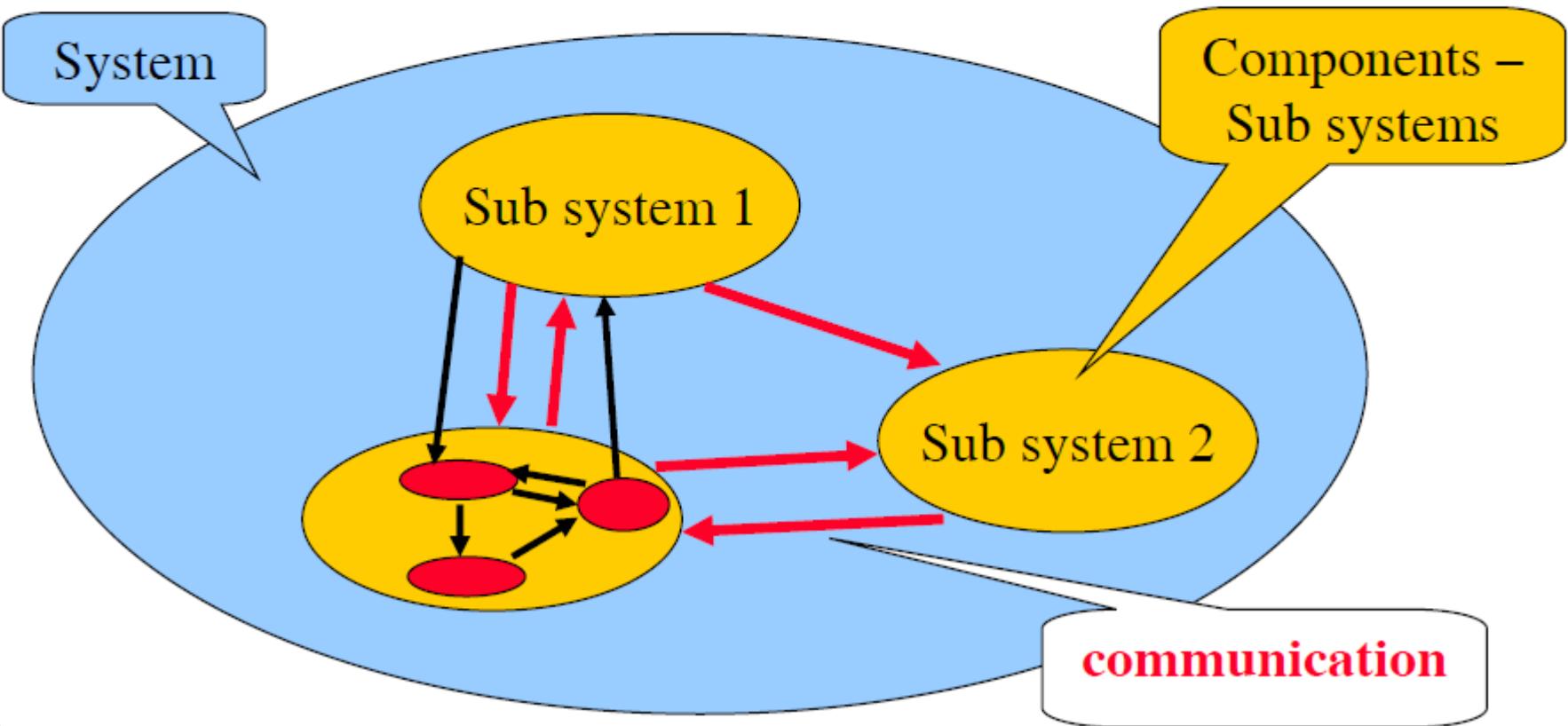
What is Software architecture?

Architectural design is the design process for:

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

Architectural design

- Identify system components and their communications



Architectural design process

- System structuring
 - The system is **decomposed into several principal sub-systems and communications** between these sub-systems are identified
- Control modelling
 - A **model of the control relationships** between the different parts of the system is established
- Modular decomposition
 - The identified **sub-systems are decomposed into modules**

Architectural design process: System structuring

- Concerned with decomposing the system into **interacting sub-systems**
- The architectural design is normally expressed as a **block diagram** presenting **an overview of the system structure**
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed

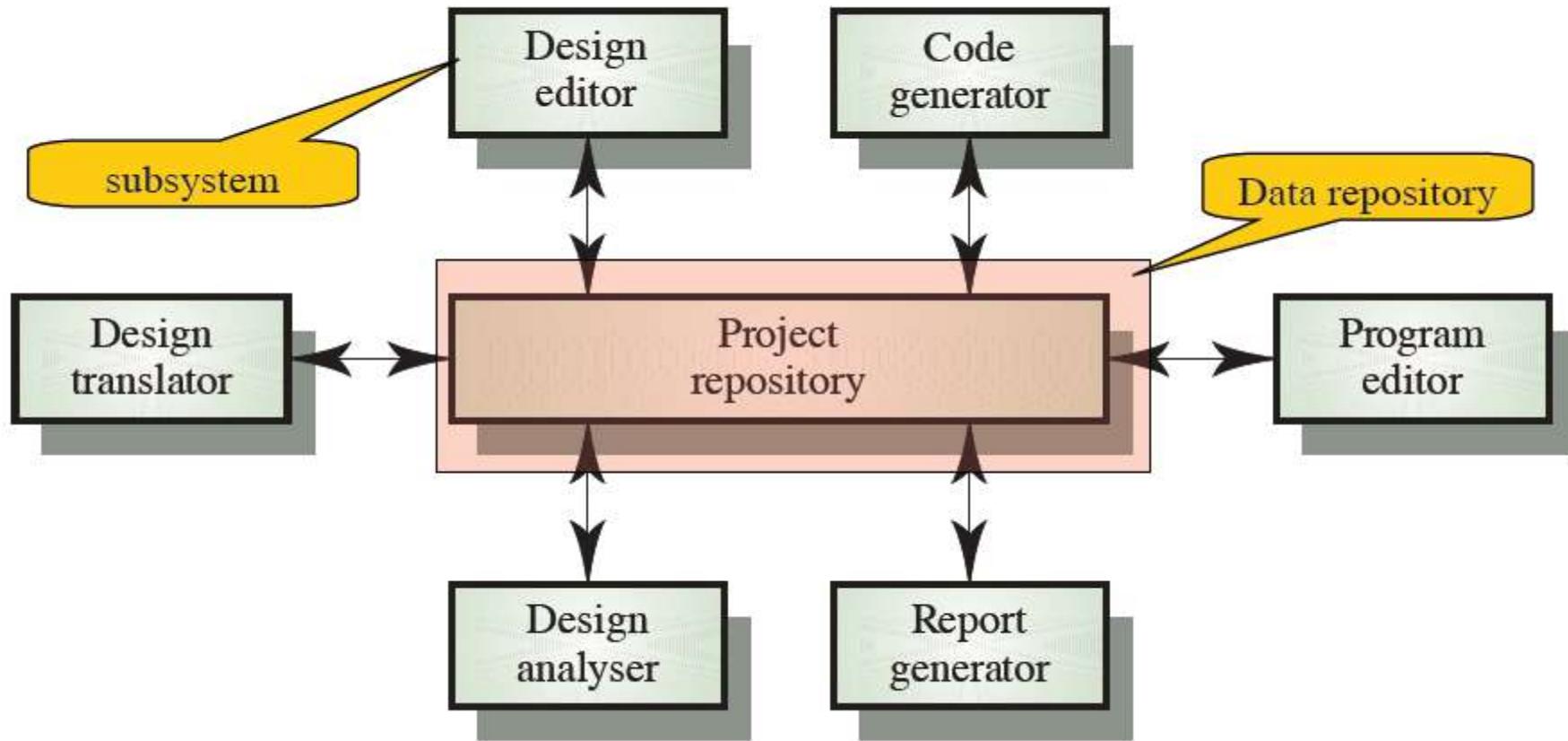
Architectural Design: System Organisation

- Reflects the basic strategy that is used to structure the system
- Three architectural styles are widely used:
 - Shared data repository
 - Client-server (services and servers)
 - Abstract machine or layered style

The repository model

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

Example: repository model for “CASE toolset architecture”



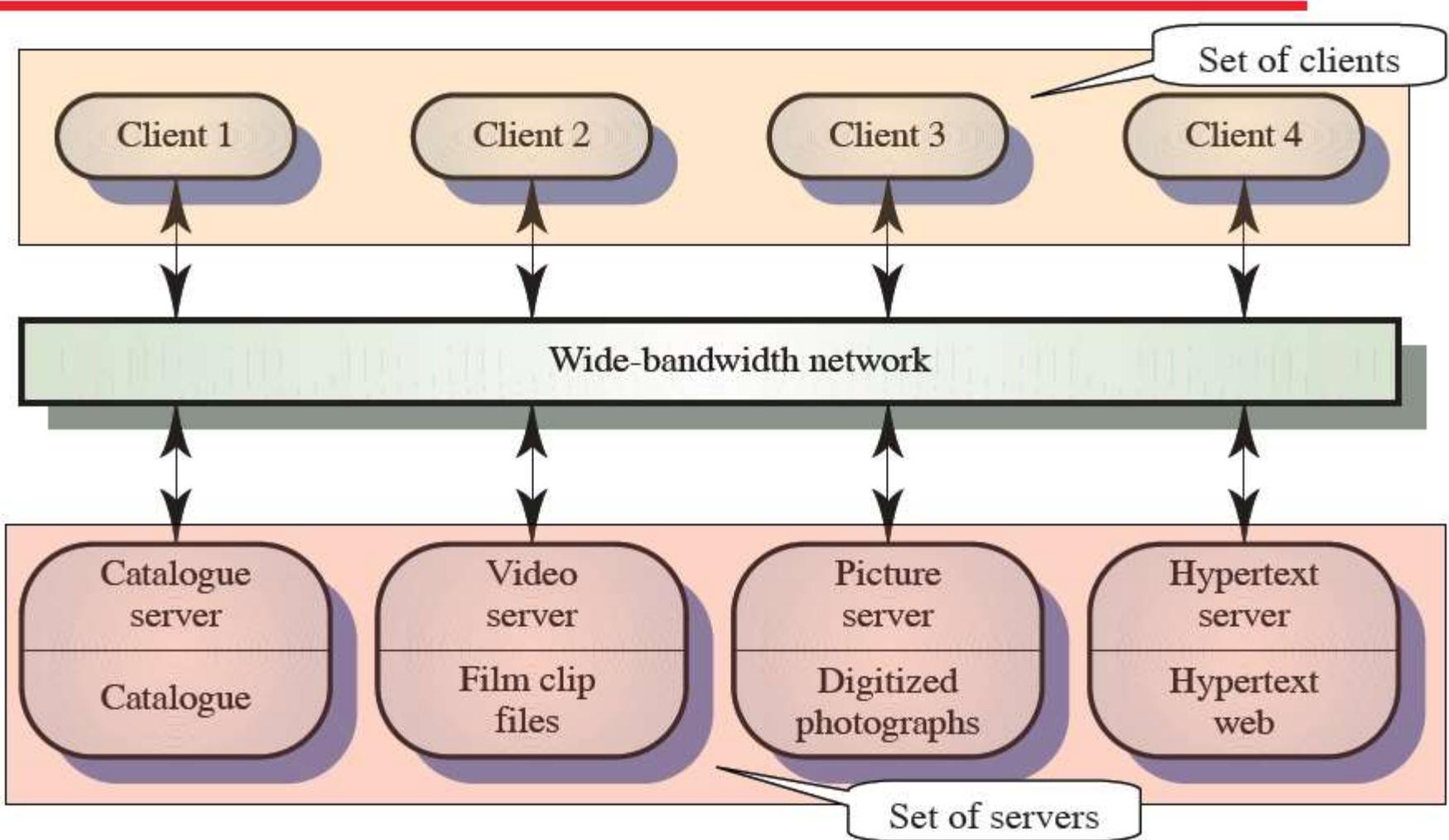
Repository Model Characteristics

- **Advantages**
 - Can efficiently share large amounts of data
 - Sub-systems need not be concerned with how data is produced by other sub-systems
 - Centralized backup, access control, and error recovery
 - New tools compatible with the repository schema (data model) are easily integrated
- **Disadvantages**
 - Sub-systems must agree on a repository data model, compromising on needs of each tool, affecting performance and integration with incompatible tools
 - Translating data into different data model is difficult, expensive, or impossible;
 - Same policy forced on all sub-systems
 - Difficult to distribute repository over many machines efficiently, leading to problems with data redundancy and inconsistency

Client-server architecture

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

Example: Client-server architecture for Film and picture library



Client-server characteristics

- Advantages
 - Distribution of data is straightforward
 - Makes effective use of networked systems. May require cheaper hardware
 - Easy to add new servers or upgrade existing servers
- Disadvantages
 - No shared data model so sub-systems use different data organisation. data interchange may be inefficient
 - Redundant management in each server
 - No central register of names and services - it may be hard to find out what servers and services are available

Abstract machine model (Layered Model)

- Used to model the interfacing of sub-systems
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected
- However, often difficult to structure systems in this way
- Difficult to structure system in layers:
 - Inner layers may provide services required in several layers, making outer layers depend on more than its adjacent layer
 - Performance may suffer when service requests must be interpreted across many layers before processing

Example: Abstract machine model for Version management system

Configuration management system layer

Object management system layer

Database system layer

Operating system layer

Topics covered

- Introduction
- Architectural design decisions
- System organisation
- **Decomposition models**
- Control models

Modular decomposition

- Another structural level where **sub-systems are decomposed into modules**
- Two modular decomposition models covered
 - **An object model** where the system is decomposed into interacting objects
 - **A data-flow model** where the system is decomposed into functional modules which transform inputs to outputs. Also known as the pipeline model
- If possible, decisions about concurrency should be delayed until modules are implemented

Object models decomposition

- Structure the system into a set of **loosely coupled objects** with well-defined interfaces
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations
- When implemented, objects are created from these classes and some control model used to coordinate object operations

Architecture Concepts

Some concepts related to architecture:

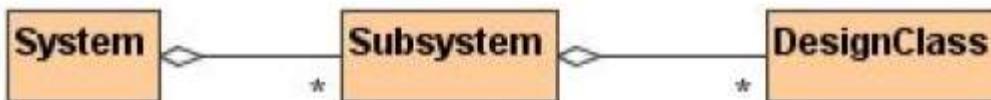
- 1) subsystems
 - a) classes
 - b) services
- 2) design principles for defining subsystems:
 - 1) coupling
 - 2) cohesion
- 3) layering strategy for defining subsystems:
 - 1) responsibility driven
 - 2) reuse driven

Subsystems: Classes

A solution domain may be decomposed into smaller parts called **subsystems**.

Subsystems may be recursively decomposed into simpler subsystems.

Subsystems are composed of solution domain classes (design classes).



Coupling

Definition

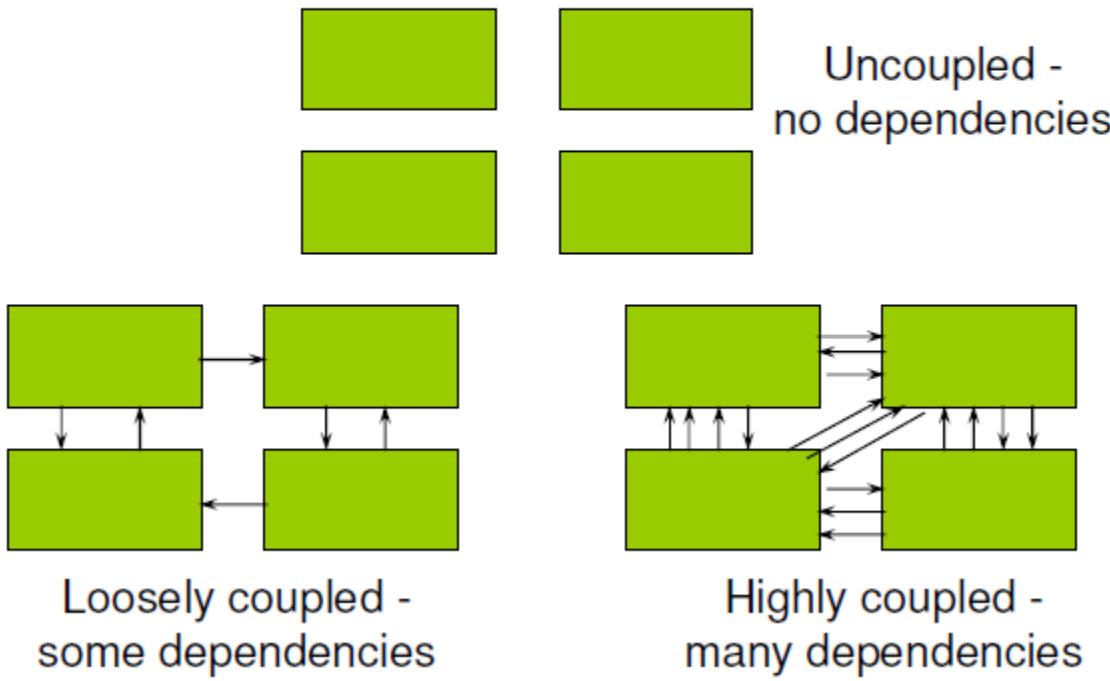
Coupling is the strength of dependencies between two sub-systems.

Loose coupling results in:

- 1) sub-system independence
- 2) better understanding of sub-systems
- 3) easier modification and maintenance

High coupling is generally undesirable.

Coupling Example



Cohesion

Definition

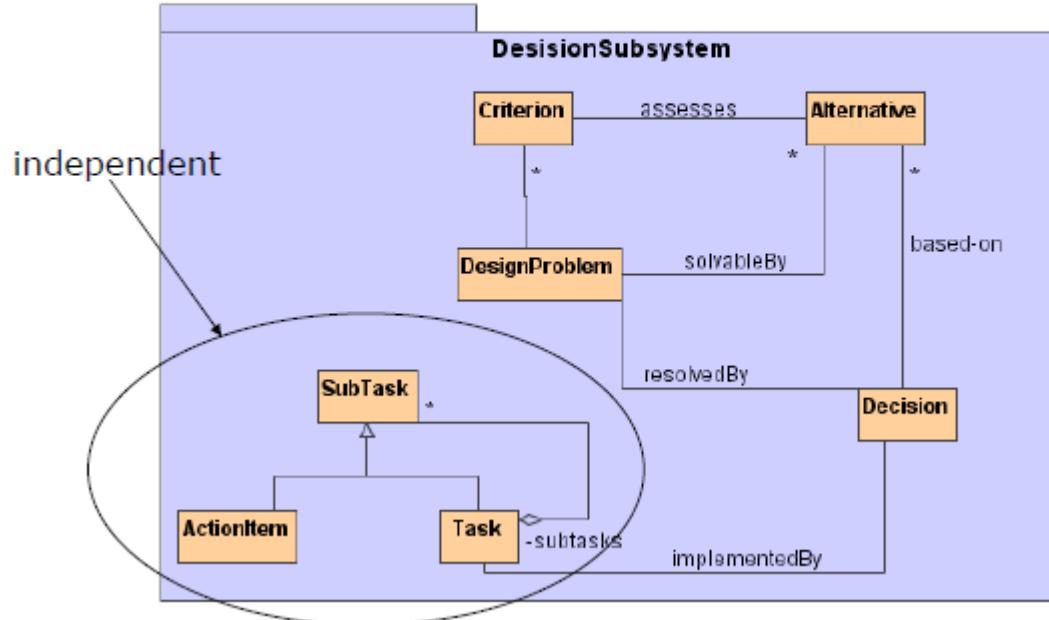
Cohesion or Coherence is the strength of dependencies within a subsystem.

In a highly cohesive subsystem:

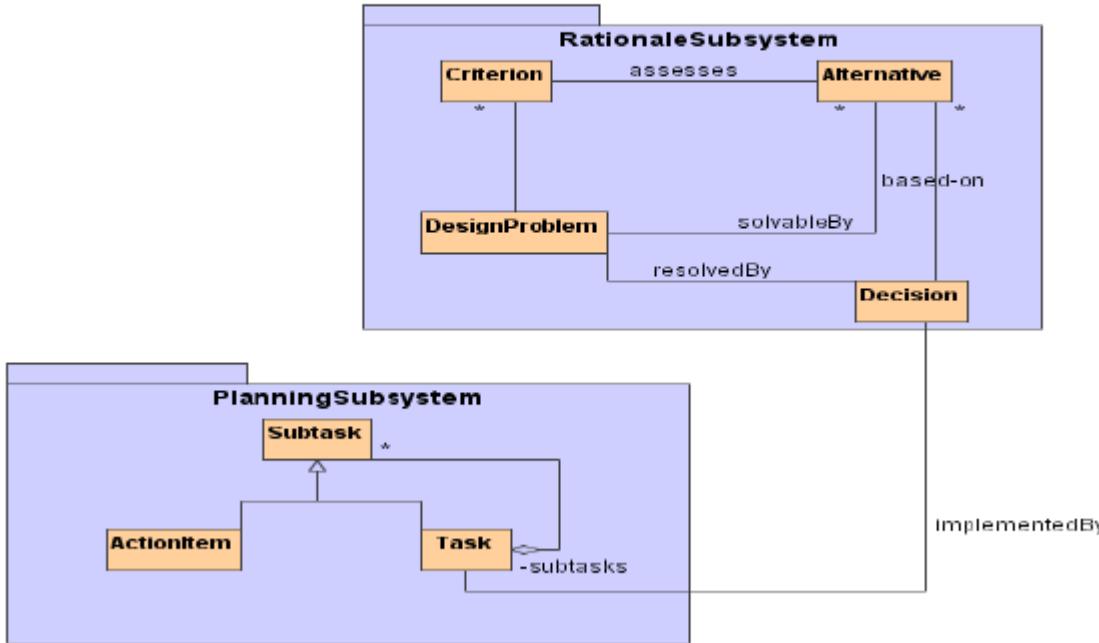
- subsystem contains related objects
- all elements are directed toward and essential for performing the same task.

Low cohesion is generally undesirable

Low Cohesion Example



High Cohesion Example



Object models decomposition

- **Advantages:**
 - Loose coupling ensures that changes in one object class does not affect other objects
 - Since objects tend to reflect real-world entities, object models are easy to understand

- **Disadvantages:**
 - Changes to the interface of an object have an impact on other users of the object
 - Complex entities may be difficult to represent as objects

Software Testing

King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

Objectives

- To discuss the distinctions between validation testing and defect testing
- To describe the principles of system and component testing
- To describe strategies for generating system test cases

Defect testing

- The goal of defect testing is to discover defects in programs
- A *successful* defect test is a test which causes a program to behave in an anomalous way
- Tests show the presence not the absence of defects

Testing & Verification & Validation

- Testing = Verification + Validation
- **Verification:** Static Testing (no run)
- **Validation:** Dynamic Testing (Run code)

Who Tests the Software



Developer

- understands the system
- has the source code
- white-box 'Unit' testing
- will test "gently"
- driven by delivery 'schedule' constraint



Independent tester

- must learn about the system
- has no source code
- black-box 'Acceptance' testing
- will attempt to break the sys (ME!!)
- driven by quality constraint

Testing policies

- Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible.
- Testing policies define the approach to be used in selecting system tests:
 - All functions accessed through menus should be tested;
 - Combinations of functions accessed through the same menu should be tested;
 - Where user input is required, all functions must be tested with correct and incorrect input.

The testing process

- Component (Unit) testing: needs source code (White-box)
 - ✚ Testing of individual program components
 - ✚ Usually the responsibility of the component developer
 - ✚ Tests are derived from the developer's experience
- System Testing: Involves integrating components to create a system or sub-system. May involve testing an increment to be delivered to the customer.
 - ✚ Integration testing - the test team have access to the system source code. The system is tested as components are integrated.
 - ✚ Release testing - the test team test the complete system to be delivered as a black-box.

Component testing

- Component or unit testing is the process of testing individual components in isolation.
- It is a 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.

System testing

- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- Two phases:
 - **Integration testing** - the test team have access to the system source code. The system is tested as components are integrated.
 - **Release testing** - the test team test the complete system to be delivered as a **black-box**.

Integration testing

- **Top-down integration testing**
 - ✚ Start with high-level system and integrate from the top-down replacing individual components by **stubs**
 - ✚ **Stubs** have the **same interface** as the components but **very limited functionality**
- **Bottom-up integration testing (XP)**
 - ✚ Integrate and test low-level components (or stories in XP), with **full functionality**, before developing higher level components, until the complete system is created
- In practice, combination of both

Release testing

- The process of testing a release of a system that will be distributed to customers.
- Primary goal is 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.

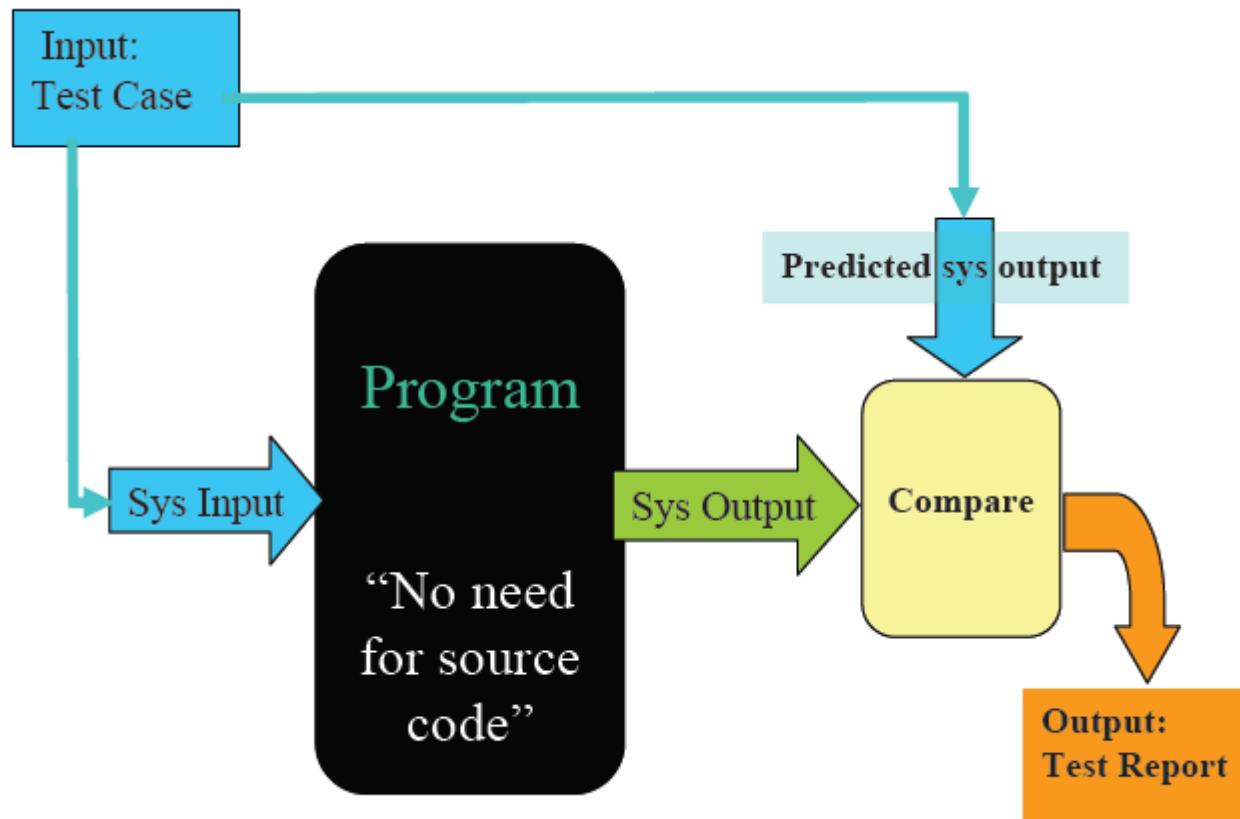
Black-box testing

- Program is considered as a ‘black-box’
- No need to know or access source code
- Functionality testing
- No implementation testing (implementation testing needs source code)
- Test cases are based on the system specification
- Test planning can begin early in the software process

Black-box testing

- Testers provide the system with inputs and observe the outputs
 - They can see none of:
 - The source code
 - The internal data
 - Any of the design documentation describing the system's internals

Black-box testing



Test case design

- Involves designing the test cases (inputs and outputs) used to test the system.
- The goal of test case design is to create a set of tests that are effective in validation and defect testing.
- Design approaches:
 - Requirements-based testing;
 - Partition testing;
 - Structural testing.
 - Path 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.

Equivalence partitioning

- Objective:

Reduce the number of test cases

Example

AGE *Accepts value 18 to 56

EQUIVALENCE PARTITIONING		
Invalid	Valid	Invalid
<=17	18-56	>=57

Assume, we have to test a field which accepts Age 18 – 56

- Valid Class: 18 – 56 = Pick any one input test data from 18 – 56
- Invalid Class 1: <=17 = Pick any one input test data less than or equal to 17
- Invalid Class 2: >=57 = Pick any one input test data greater than or equal to 57

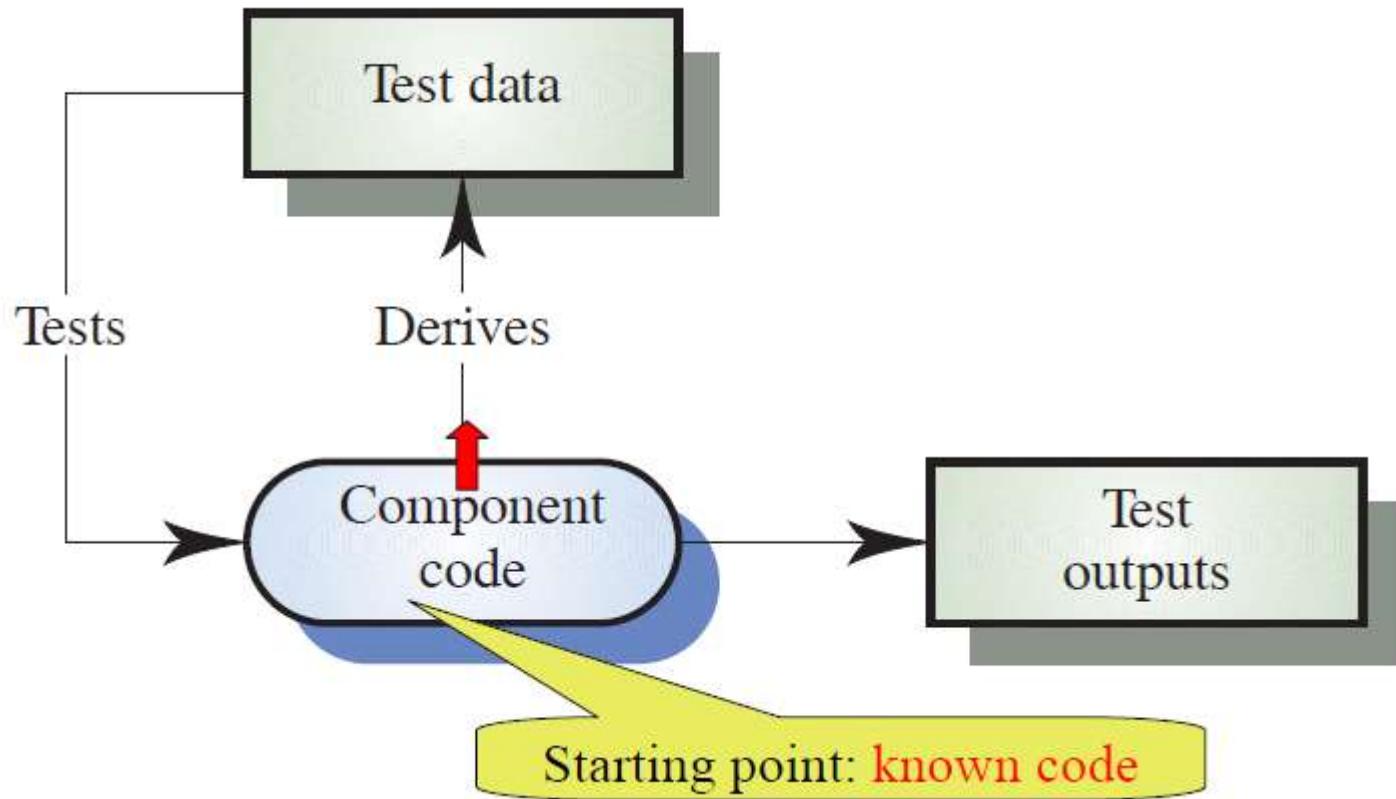
Structural testing: White-box testing

- Sometime called white-box testing.
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases.
- Objective is to exercise all program statements (not all path combinations).

Structural testing: White-box testing

- **Synonyms:**
 -  **Glass-box, Clear-box, Transparent-box**
- For small program units
- Needs **source code**
- Objective: is to exercise **all program statements**
 - (not all path combinations)

Structural testing: White-box testing



Path testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
- The 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 are therefore nodes in the flow graph.

Program flow graphs

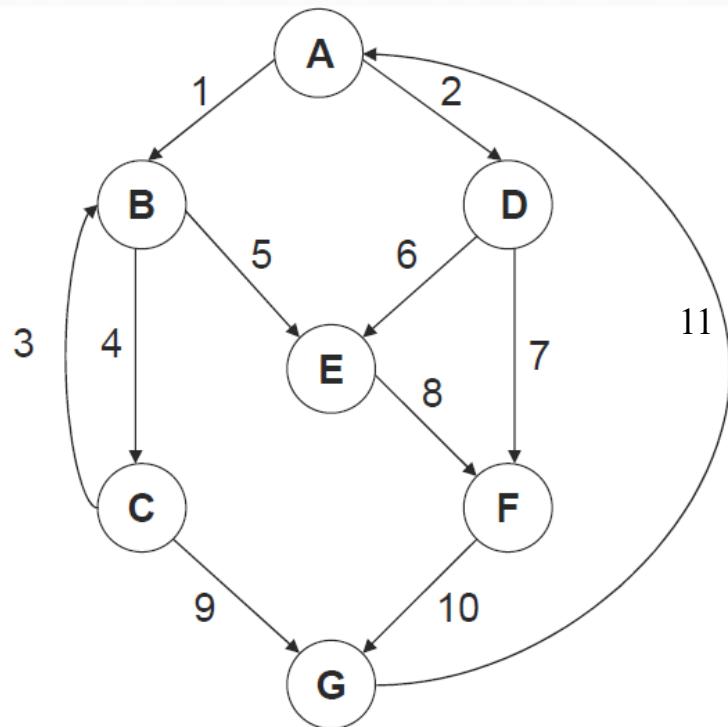
- **Flow Graph:**
 - ✚ nodes representing program decisions
 - ✚ arcs representing the flow of control
 - ✚ Ignore sequential statements (assignments, procedures calls, I/O)
- Statements with conditions are therefore nodes in the flow graph

- **Cyclomatic complexity =**
Number of edges - Number of nodes + 2

Cyclomatic complexity

- Cyclomatic complexity = **number of tests** to test all control statements
- Cyclomatic complexity = number of conditions in a program
- Although all paths are executed, all combinations of paths are not executed

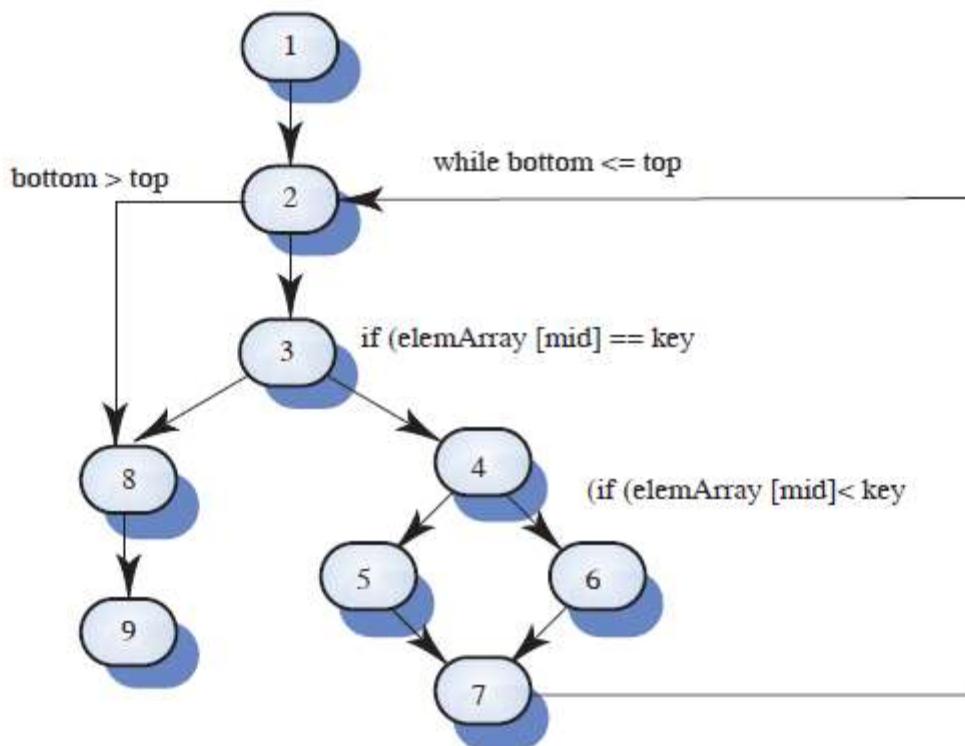
Example



- Path 1: A, B, C, G.
- Path 2: A, B, C, B, C, G.
- Path 3: A, B, E, F, G.
- Path 4: A, D, E, F, G.
- Path 5: A, D, F, G.
- Path 6: A, D, F, G, A, B, C, G

$$\text{Cyclomatic Complexity} = 11 - 7 + 2 = 6$$

Binary search flow graph



Independent paths

$$\text{Cyclomatic Complexity} = 11 - 9 + 2 = 4$$

Independents Paths:

1-2-8-9

1-2-3-8-9

1-2-3-4-5-7-2-8-9

1-2-3-4-6-7-2-8-9

Key points

- Testing can show the presence of faults in a system; 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.

Key points

- Use experience and guidelines to design test cases in defect testing.
- 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.