

CS251

Dept. of Computer Science
Undergraduate - Level 2

Software Engineering I

Module Outline

Spring “Semester 2” 2018 / 2019

Module Info.

Lectures:

Group A: Thursdays (*from 10:00 a.m. to 12:00 p.m.*)

Group B: Thursdays (*from 12:00 p.m. to 2:00 p.m.*)

Instructor:

Dr. Amr S. Ghoneim

- Office hours: *Thursdays from 9 a.m. – 10 a.m.*
- Short Bio:
 - Bachelor in Computer Science from Helwan University
 - Masters in Computer Science from Helwan University
 - PhD. in Computer Science (Artificial Intelligence) from University of New South Wales, Australia

Module Info. (*Continued*)

TAs:

- To be announced ...

Indicative Reading List (textbooks):

Textbook:

- Ian Sommerville, "Software Engineering (9th Edition)", Addison Wesley, ISBN: 978-0137035151, 2010.

For OO Design Principles, Analysis & Design, UML Modelling:

- Bernd Bruegge, Allen H. Dutoit , Object-Oriented Software Engineering: Using UML, Patterns and Java, 3rd Edition, Prentice Hall 2009.

For the Design Patterns:

- Vlissides, J., Helm, R., Johnson, R. and Gamma, E., 1995. Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley.

And / Or

- Freeman E, Freeman E, Robson E, Bates B, Sierra K. Head first design patterns. O'Reilly Media Inc., 2004.

Assessment Scheme

- | | |
|--|-----|
| <input type="checkbox"/> Group Project (Weeks 4, 6, & 12): | 25% |
| • Practical Project 15% and 10% for Individual Assessment | |
| • 10,000 words (<i>approx.</i>) report + the detailed diagrams for software modelling + code (<i>implementation</i>) | |
|
 | |
| <input type="checkbox"/> Midterm Test: (week 7) | 15% |
|
 | |
| <input type="checkbox"/> 3-hours Final Written Exam: | 60% |

Group Project 25%

Register with the TA(s); Your group members, & the project idea, by the end of week 3.

To apply the concepts discussed in lectures the student will get engaged in a practical project.

Project Ideas: The students will have the freedom to present the project idea that they would like to develop (however the idea should be within one of themes that will be announced later).

Project groups: Each group will be 5 to 6 students.

The students will be required to perform a 10-15 min demo of their project during the final discussion.

Group Project 25%

Tentative/Indicative Project deliverables:

Week 4: (Part of registering the project)

A report of 200 -500 word indicating the following:

- Project idea
- Main functionalities
- Similar applications in the market
- Development platform

Group Project 25%

Week 7 or 8: (Phase I Submission - 10%)

UML design document indicating the following:

- Functional Requirements (1 mark)
- Non-Functional Requirements (1 mark)
- System Architecture (1 mark)
- Activity Diagram(s) (1 mark)
- Use-Case Diagram(s) (1 mark)
 - *including general use-cases for the system, and the detailed use-cases description*
- Sequence Diagram(s) (1 mark)
 - *including system sequence diagrams*
- Collaboration Diagram(s) (1 mark)
- Database Specification (1 mark)
- Class Diagram (2 mark)
- Snap shots of User Interface

Group Project 25%

Tentative/Indicative Project deliverables:

Week 12: (Phase II Submission - 15%)

- A final report that includes:
 - Finalized UML Diagrams
 - Design Patterns (if applicable)
 - Snap Shots of User Interfaces
 - Test plan and Test cases performed
- Working code

(details will be announced later)

What is expected from you...

- Attend the class regularly.
- Study and learn the material presented in the class (*and refer to your reading list*).
- Do the project (*in a team of 5 to 6*).
- Perform well in the exams.
- Don't cheat (*Plagiarism*).

Tentative Weekly Plan

- **Week 1: 9 February – 15 February**
 - Lecture 0 (Module Outline, Introduction)
 - Lecture 1 (Software Processes - Part I)
- **Week 2: 16 February – 22 February**
 - Lecture 2 (Requirements Engineering)
- **Week 3: 23 February – 1 March**
 - Lecture 3 (Introduction to Modelling and Object Oriented Modelling)
- **Week 4: 2 March – 8 March**
 - Lecture 4 (Class Diagrams, & Package Diagrams)
- **Week 5: 9 March – 15 March**
 - Lecture 5 (UML Functional & Dynamic Diagrams)
- **Week 6: 16 March – 22 March**
 - Lecture 6 (Intro. to Design & Architectural Patterns)

Tentative Weekly Plan

- **Week 7: 23 March – 29 March**
 - Midterm Exam
- **Week 8: 30 March – 5 April**
 - Project Discussions (Phase 1)
 - Lecture 7 (UML Dynamic Diagrams – Part II)
- **Week 9: 6 April – 12 April**
 - Lecture 8 (Software Processes - Part II)
- **Week 10: 13 April – 19 April**
 - Lecture 9 (UML for Realtime Systems, & Testing)
- **Week 11: 20 April – 26 April**
 - Public Holiday (Sinai Liberation Day)
- **Week 12: 27 April – 3 May**
 - Project Discussions (Phase 2)
 - Lecture 10 (More on Design Patterns)

Why ... Software Engineering?

- 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 (*Gross National Product*) in all developed countries.

Why ... Software Engineering?

- The Standish Group has been publishing reports on the success of software projects since 1994. In its latest report (2012) it identified that:
 - Only **39%** of all projects were being delivered on time, on budget, with the required features and functions.
 - .. **43%** were late, or over budget, or not delivering all the required features.
 - .. **18%** were either cancelled before delivery, or delivered but never used.
- In those reports, the major problems have been identified. **Among these problems were the following:**
 - Poor, or lack of, user involvement.
 - Lack of clear business objectives.
 - Under and over building – building features that are never used and not building all the required features.

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 (useful) and should be flexible, usable, reliable, available, affordable, and maintainable (greatly influenced by how it is designed and written).

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.

Frequently asked questions about Software Engineering

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.

What is a System boundary?

- a conceptual line that divides the system that we wish to study from ‘everything else’, (scope of a system).

What is a System’s environment?

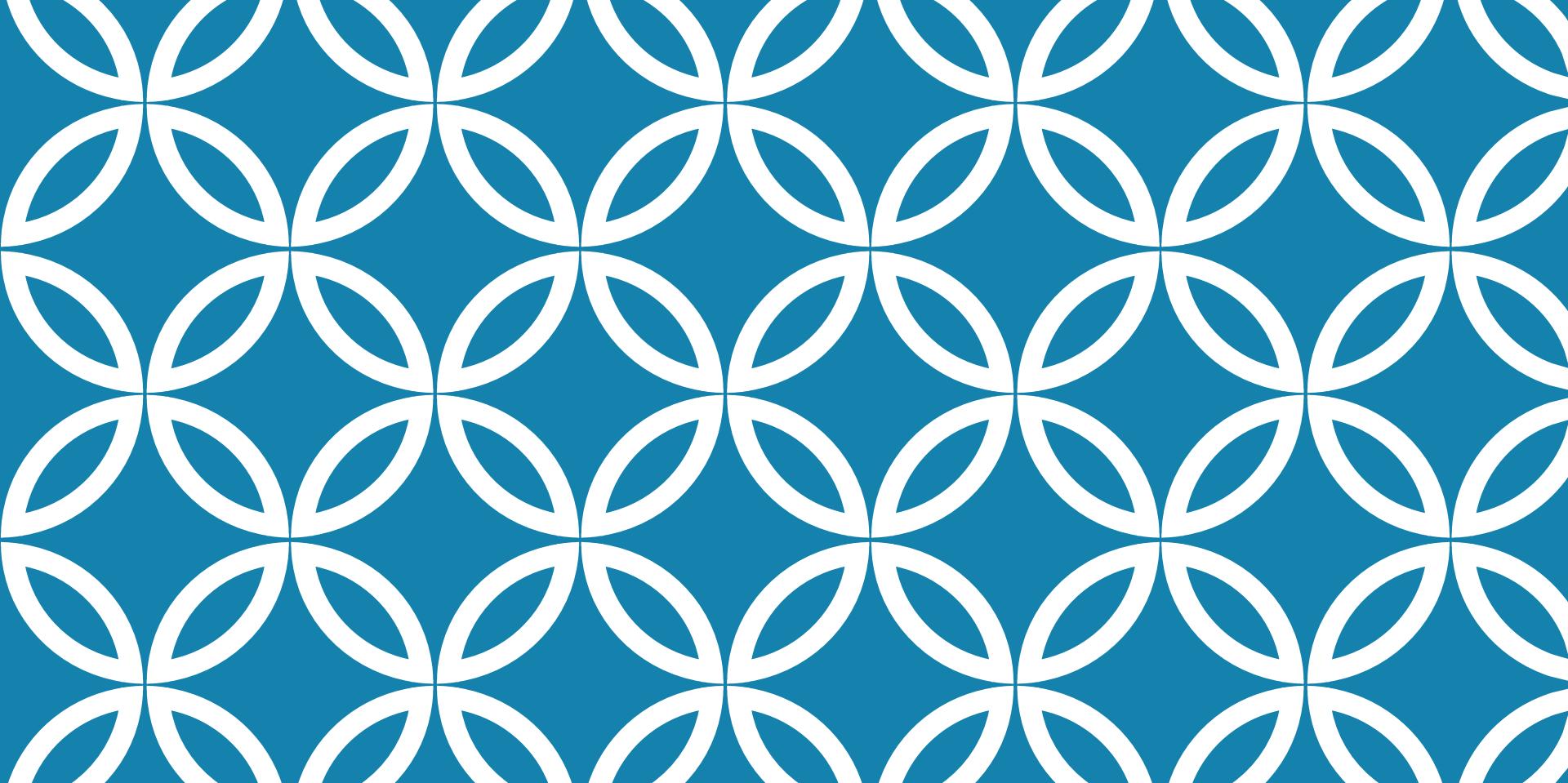
- It is made up of those things which are not part of the system, but which can either affect the system or be affected by it.

What is a domain?

- It is a particular area of interest.

The Important characteristics of Software that affect its development

- **Malleability:** Software is easy to change. This malleability creates a constant pressure for software to be changed rather than replaced.
- **Complexity:** Software is often complex. Complexity can usually be recognized, but it is less easy to define. One item of software can be considered more complex than another if the description of the first requires more explanation than that of the second. If complexity is increased, errors will be increased.
- **Size:** It is likely that there will be more errors in a large piece of software than there will be in a small one..



(CS251) SOFTWARE ENGINEERING I

Lecture 1
Software Processes
(Chapter 2)

TOPICS COVERED

- **Software Process Models**
 - Plan-Driven and Agile Processes
 - The Waterfall Model
 - Incremental Development
 - Reuse-Oriented Software Engineering
- **Process Activities**
 - Software Specification
 - Software Design and Implementation
 - Design activities
 - Software Validation
 - Testing stages
 - Software Evolution

THE SOFTWARE PROCESS

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

Many different software processes but all involve:

- **Specification** – defining what the system should do;
- **Design and implementation** – defining the organization of the system and implementing the system;
- **Validation** – checking that it does what the customer wants;
- **Evolution** – changing the system in response to changing customer needs.

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

SOFTWARE PROCESS DESCRIPTIONS

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

Process descriptions may also include:

- **Products**, which are the outcomes of a process activity;
- **Roles**, which reflect the responsibilities of the people involved in the process;
- **Pre- and post-conditions**, which are statements that are true before and after a process activity has been enacted or a product produced.

PLAN-DRIVEN AND AGILE PROCESSES

Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.

In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.

In practice, most practical processes include elements of both plan-driven and agile approaches.

There are no right or wrong software processes.

SOFTWARE PROCESS MODELS

The waterfall model

- Plan-driven model. Separate and distinct phases of specification and development.

Incremental development

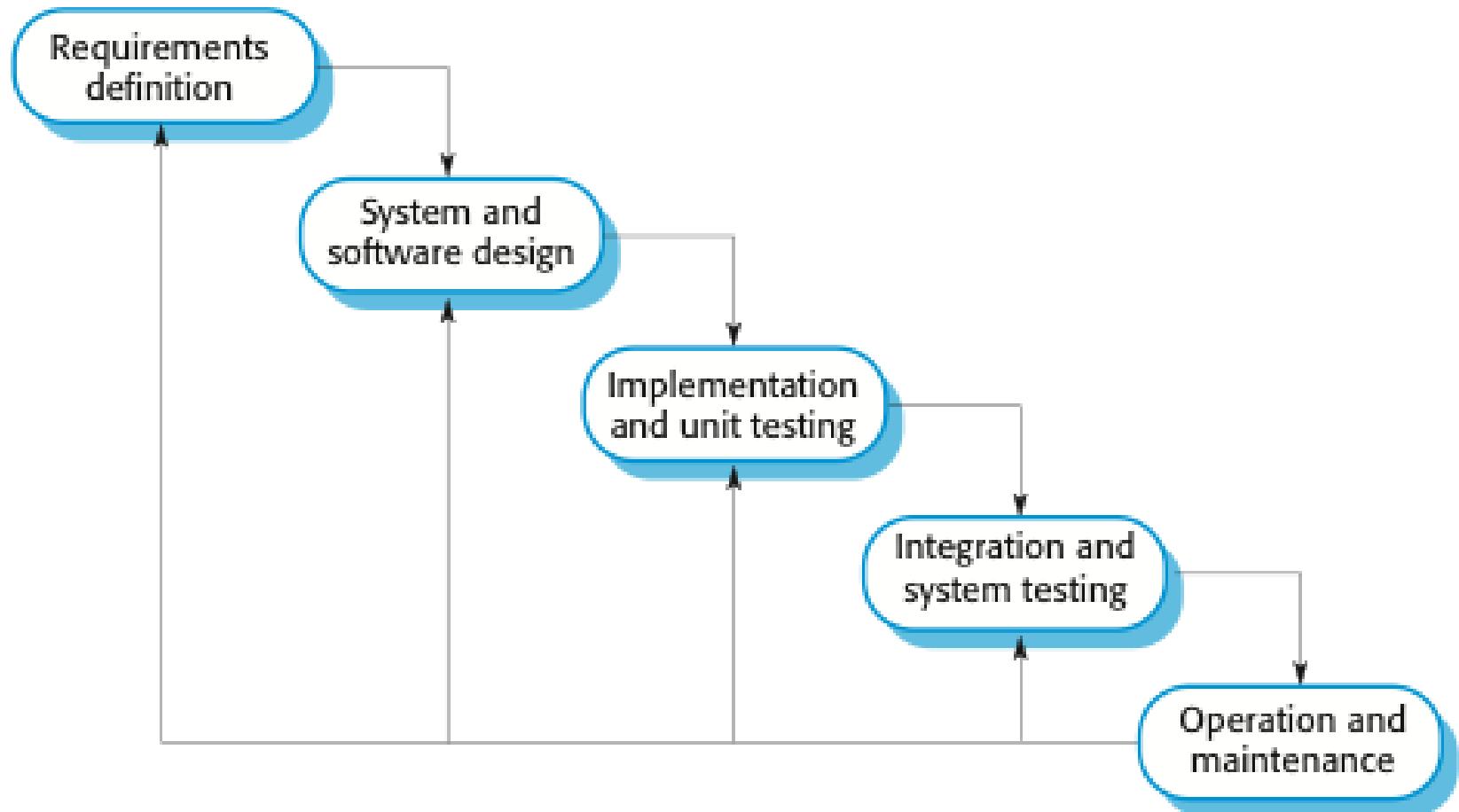
- Specification, development and validation are interleaved. May be plan-driven or agile.

Reuse-oriented software engineering

- The system is assembled from existing components. May be plan-driven or agile.

In practice, most large systems are developed using a process that incorporates elements from all of these models.

THE WATERFALL MODEL



WATERFALL MODEL PHASES

There are separate identified phases in the waterfall model:

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance

The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

WATERFALL MODEL PROBLEMS

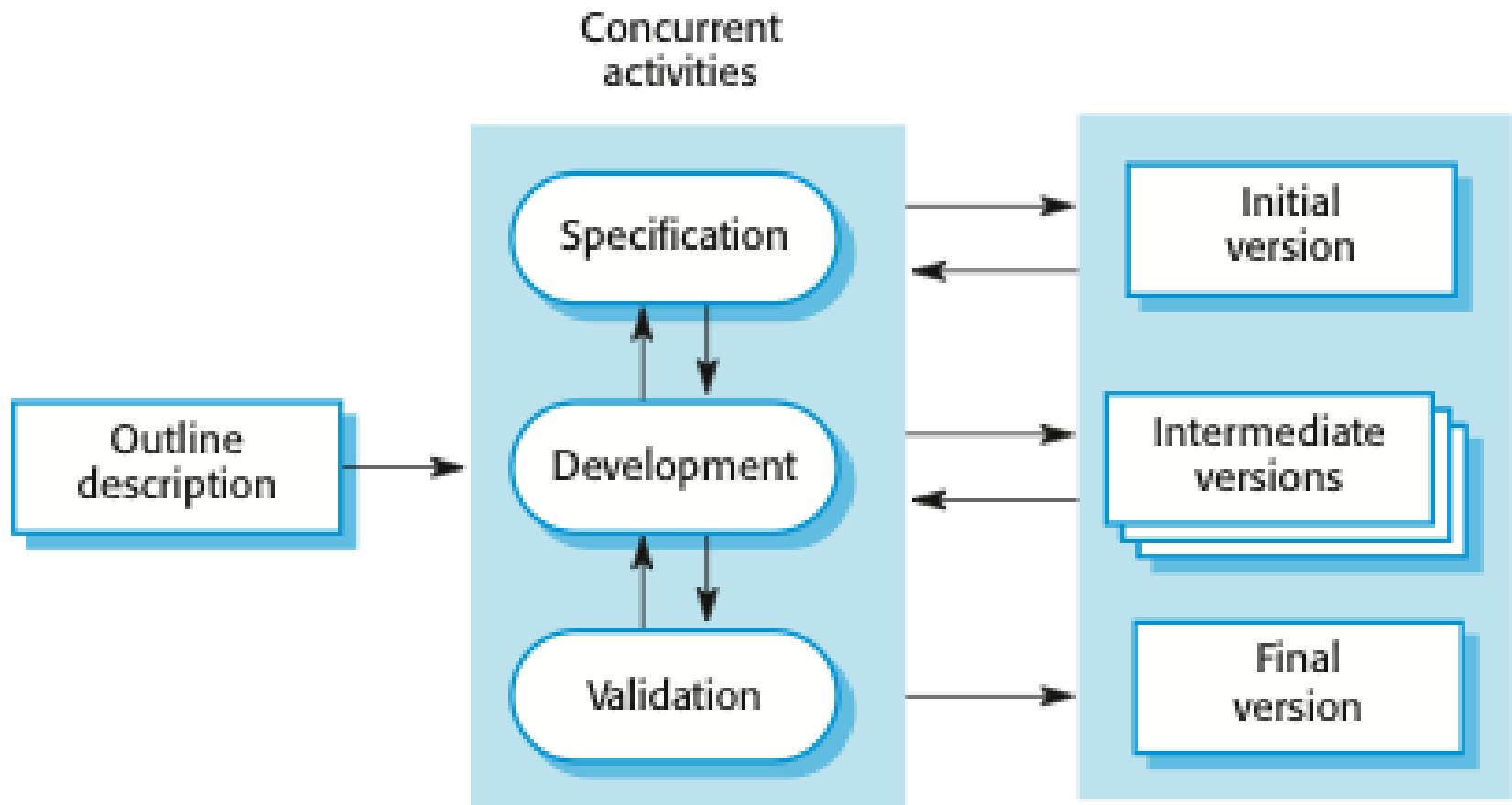
Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.

- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
- Few business systems have stable requirements.

The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.

- In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

INCREMENTAL DEVELOPMENT



INCREMENTAL DEVELOPMENT BENEFITS

The cost of accommodating changing customer requirements is reduced.

- The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.

It is easier to get customer feedback on the development work that has been done.

- Customers can comment on demonstrations of the software and see how much has been implemented.

More rapid delivery and deployment of useful software to the customer is possible.

- Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

INCREMENTAL DEVELOPMENT PROBLEMS

The process is not visible.

- Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.

System structure tends to degrade as new increments are added.

- Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

REUSE-ORIENTED SOFTWARE ENGINEERING

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

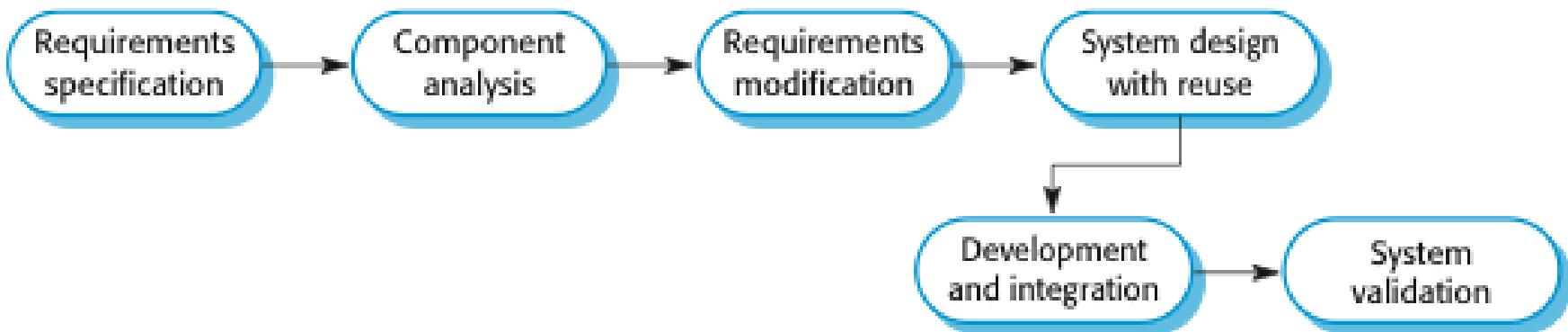
Process stages

- Component analysis;
- Requirements modification;
- System design with reuse;
- Development and integration.

Reuse is now the standard approach for building many types of business system

- Reuse covered in more depth in Chapter 16.

REUSE-ORIENTED SOFTWARE ENGINEERING



TYPES OF SOFTWARE COMPONENT

Web services that are developed according to service standards and which are available for remote invocation.

Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.

Stand-alone software systems (COTS) that are configured for use in a particular environment.

PROCESS ACTIVITIES

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

The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are inter-leaved.

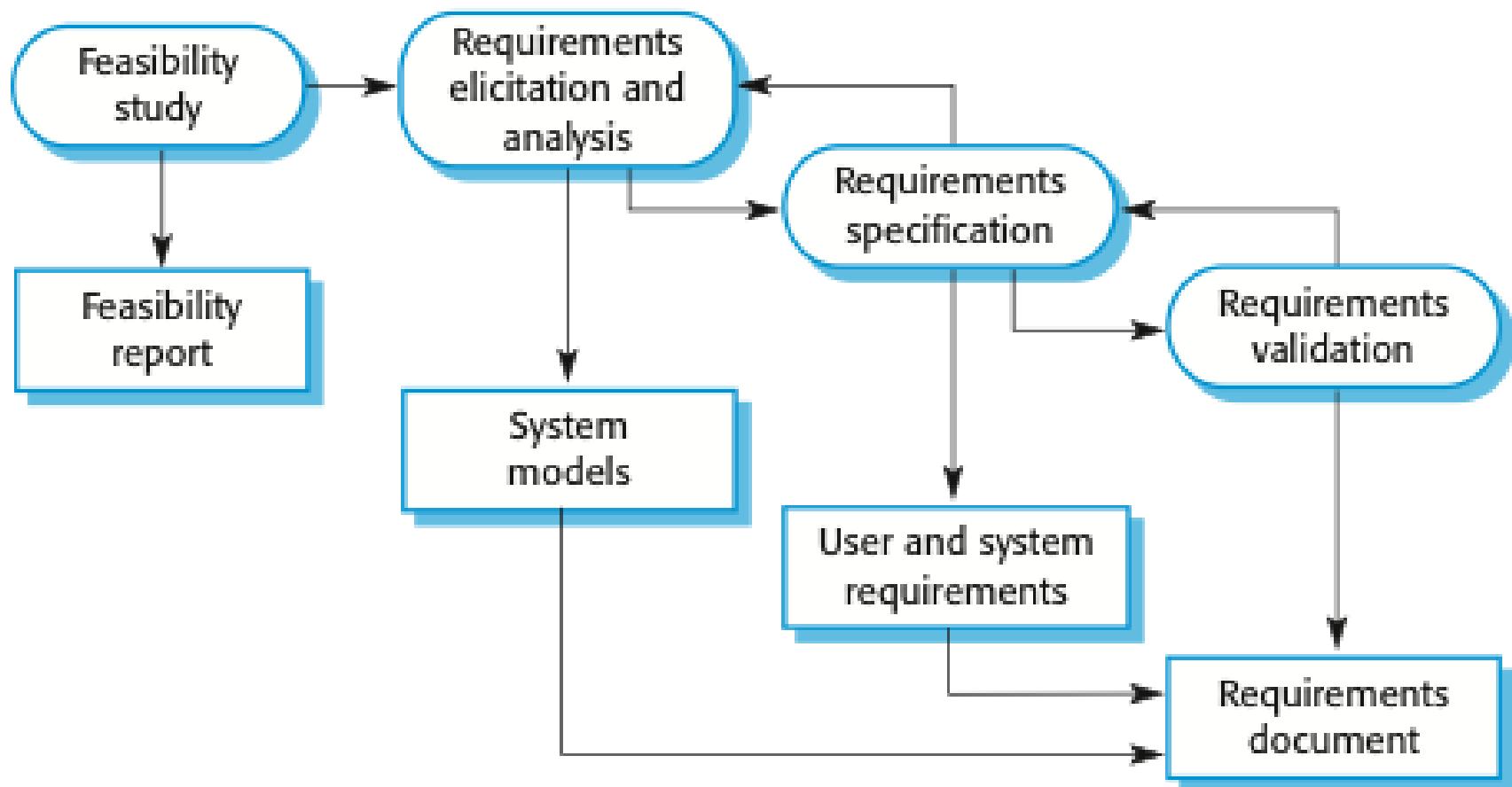
SOFTWARE SPECIFICATION

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

Requirements engineering process

- Feasibility study
 - Is it technically and financially feasible to build the system?
- Requirements elicitation and analysis
 - What do the system stakeholders require or expect from the system?
- Requirements specification
 - Defining the requirements in detail
- Requirements validation
 - Checking the validity of the requirements

THE REQUIREMENTS ENGINEERING PROCESS



SOFTWARE DESIGN AND IMPLEMENTATION

The process of converting the system specification into an executable system.

Software design

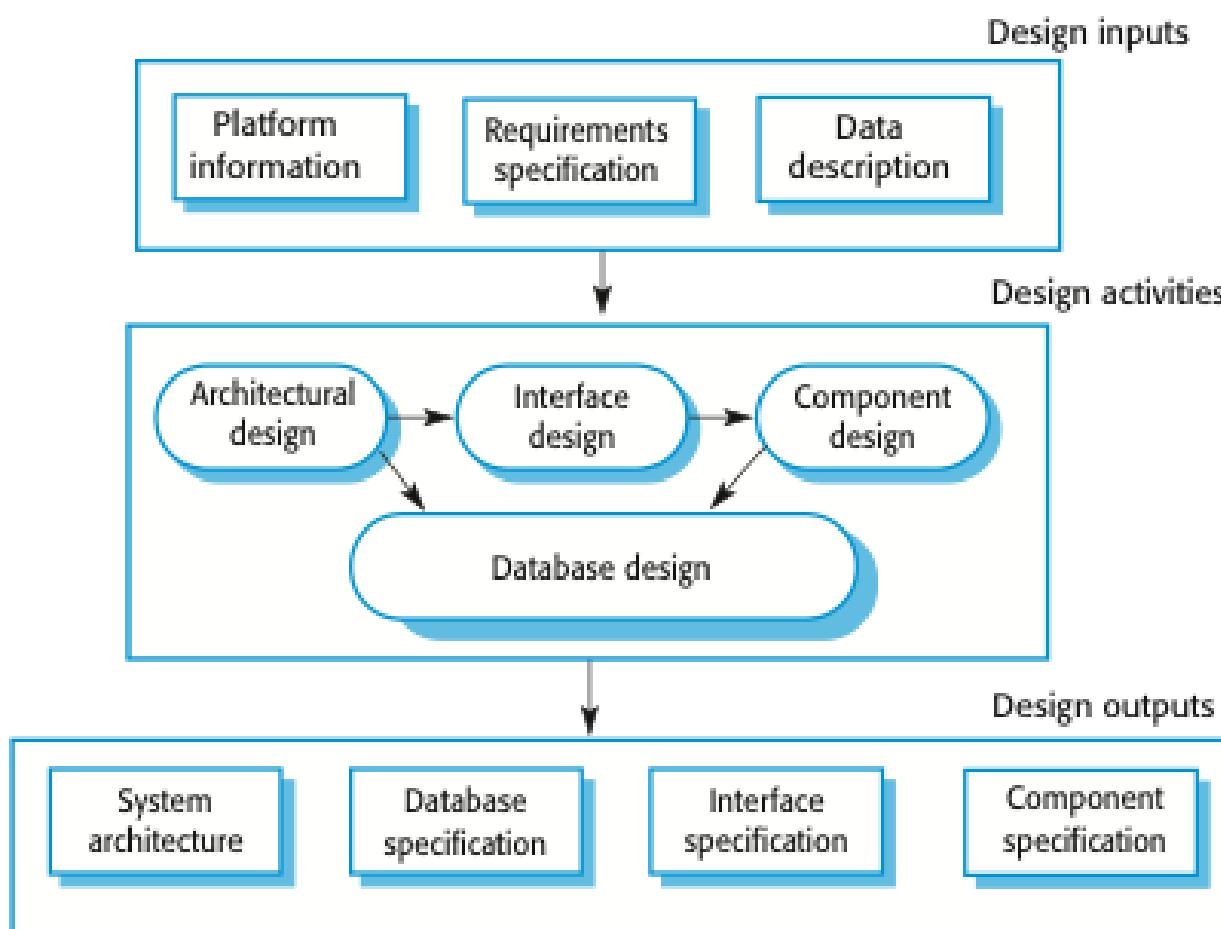
- Design a software structure that realises the specification;

Implementation

- Translate this structure into an executable program;

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

A GENERAL MODEL OF THE DESIGN PROCESS



DESIGN ACTIVITIES

Architectural design, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships and how they are distributed.

Interface design, where you define the interfaces between system components.

Component design, where you take each system component and design how it will operate.

Database design, where you design the system data structures and how these are to be represented in a database.

SOFTWARE VALIDATION

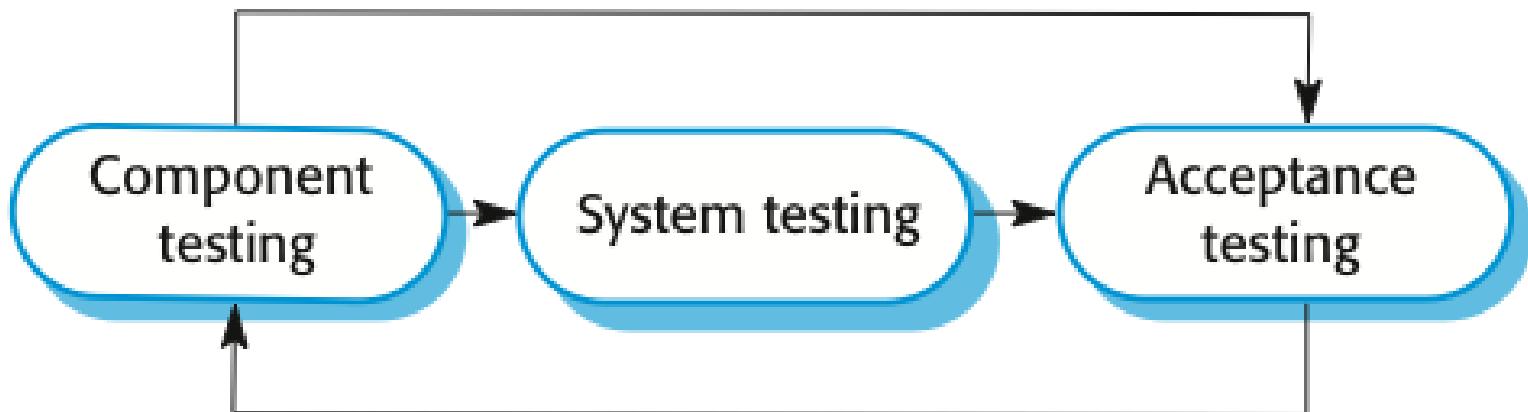
Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.

Involves checking and review processes and system testing.

System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.

Testing is the most commonly used V & V activity.

STAGES OF TESTING



TESTING STAGES

Development or component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

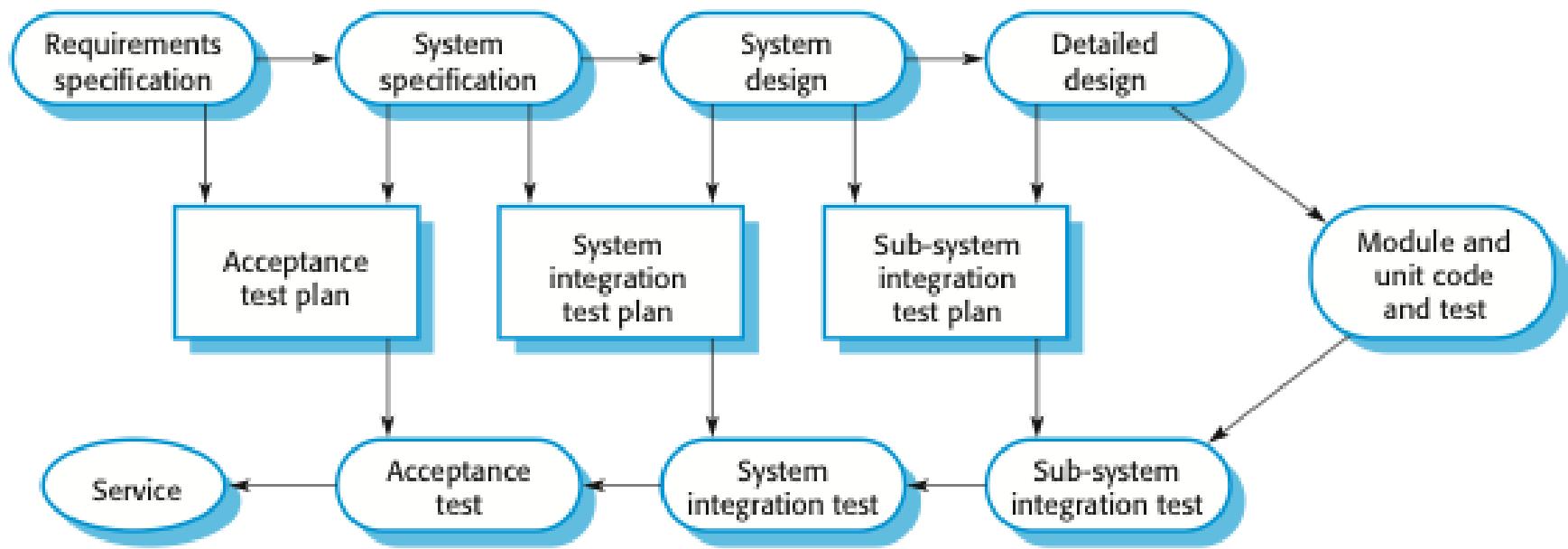
System testing

- Testing of the system as a whole. Testing of emergent properties is particularly important.

Acceptance testing

- Testing with customer data to check that the system meets the customer's needs.

TESTING PHASES IN A PLAN-DRIVEN SOFTWARE PROCESS



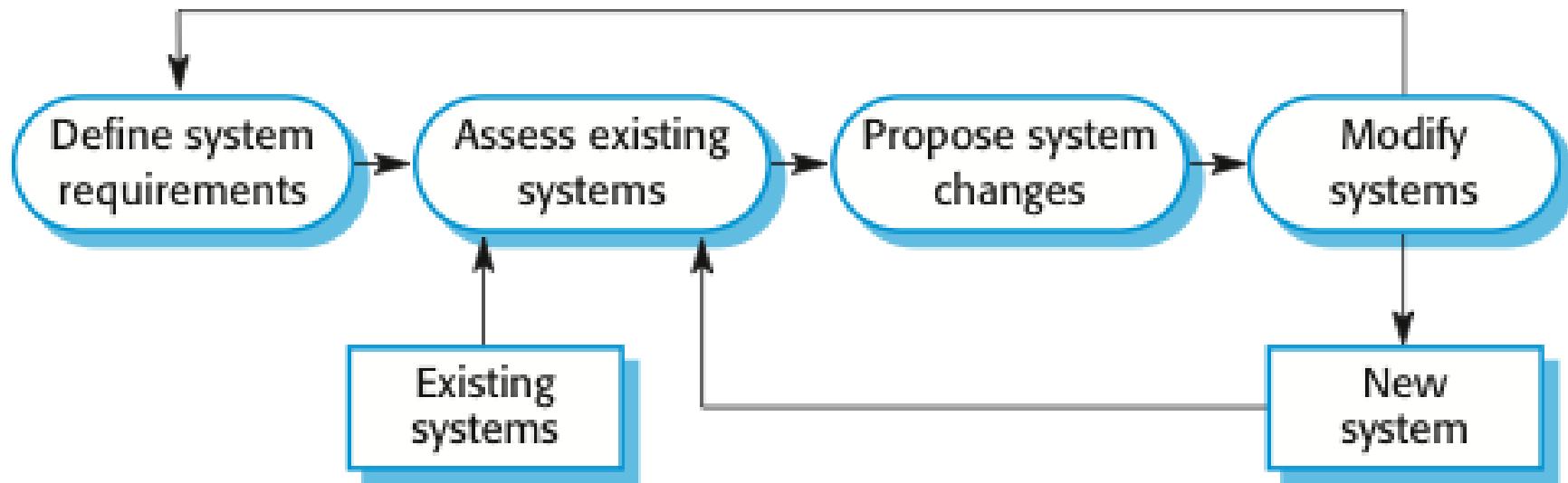
SOFTWARE EVOLUTION

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



KEY POINTS

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

General process models describe the organization of software processes. Examples of these general models include the ‘waterfall’ model, incremental development, and reuse-oriented development.

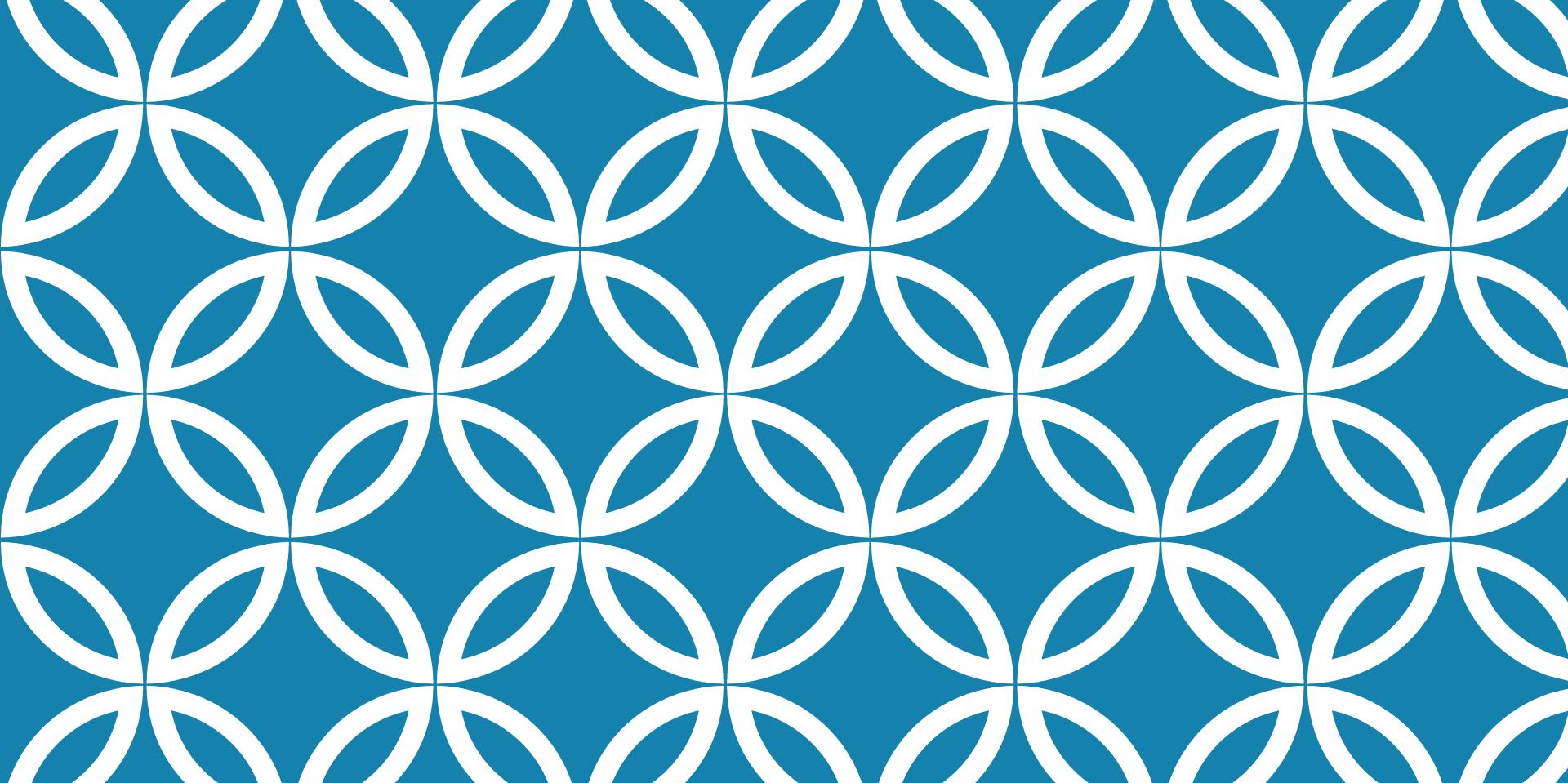
KEY POINTS

Requirements engineering is the process of developing a software specification.

Design and implementation processes are concerned with transforming a requirements specification into an executable software system.

Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.

Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.



(CS251) SOFTWARE ENGINEERING I

Lecture 2
Requirements Engineering
(Chapter 4)

TOPICS COVERED

- The importance of Requirements Engineering, and What is a Requirement.
- Functional, and Non-functional, and Domain requirements.
- Categories of Non-functional requirements, and their Fit-Criteria.
- The Software Requirements Document (SRD).
- Requirements Specification.
- Requirements Engineering processes.
- Requirements Elicitation & Analysis, and the Approaches for their Discovery.
- Requirements Validation & Evolution.

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.

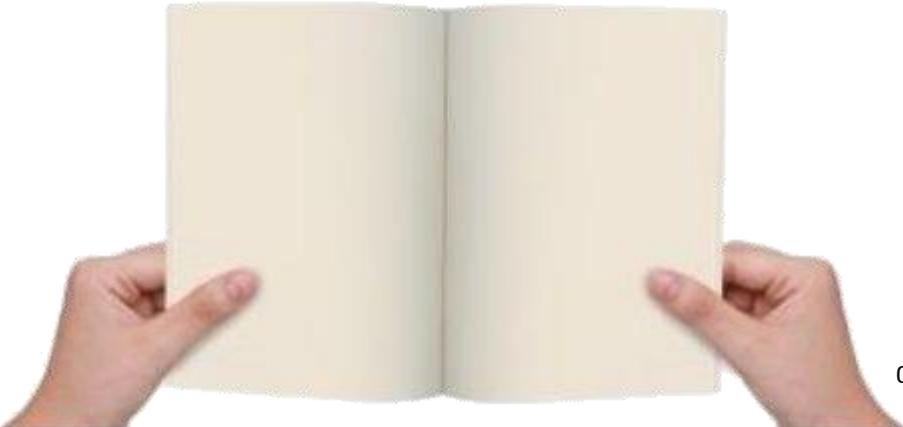
REQUIREMENTS ENGINEERING .. WHY?

**“Without requirements or design, programming
is the art of adding bugs to an empty text file.”**

- Louis Srygley

REQUIREMENTS ENGINEERING .. WHY?

SOFTWARE TESTING
without requirements



WHAT IS A REQUIREMENT?

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

This is inevitable as requirements may serve a dual function:

- May be the basis for a bid for a contract - therefore must be open to interpretation;
- May be the basis for the contract itself - therefore must be defined in detail;
- Both these statements may be called requirements.

REQUIREMENTS AND DESIGN/ARCHITECTURE

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

In practice, requirements and design are inseparable

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

REQUIREMENTS AND DESIGN/ARCHITECTURE

AN EXAMPLE

- ❖ Requirements may contradict each other, for example portability and usability in a mobile device, and a compromise will need to be achieved.
- ❖ Solving these conflicts involves taking decisions that will affect the architecture of a system.

REQUIREMENTS AND TESTING

- Testing is usually regarded as an activity that takes place at the end of software development, and this is what a waterfall process model suggests.
- Testing can be done as soon as requirements are being analyzed.
- Requirements do not make much sense if there is no way that they can be tested.
- When defining requirements, a developer may be implicitly thinking about tests that a system will have to go through.

Requirements, if they are to be implemented successfully, must be measurable and testable. (Robertson and Robertson, 2012, Truth 10, Chapter 1)

USER AND SYSTEM REQUIREMENTS

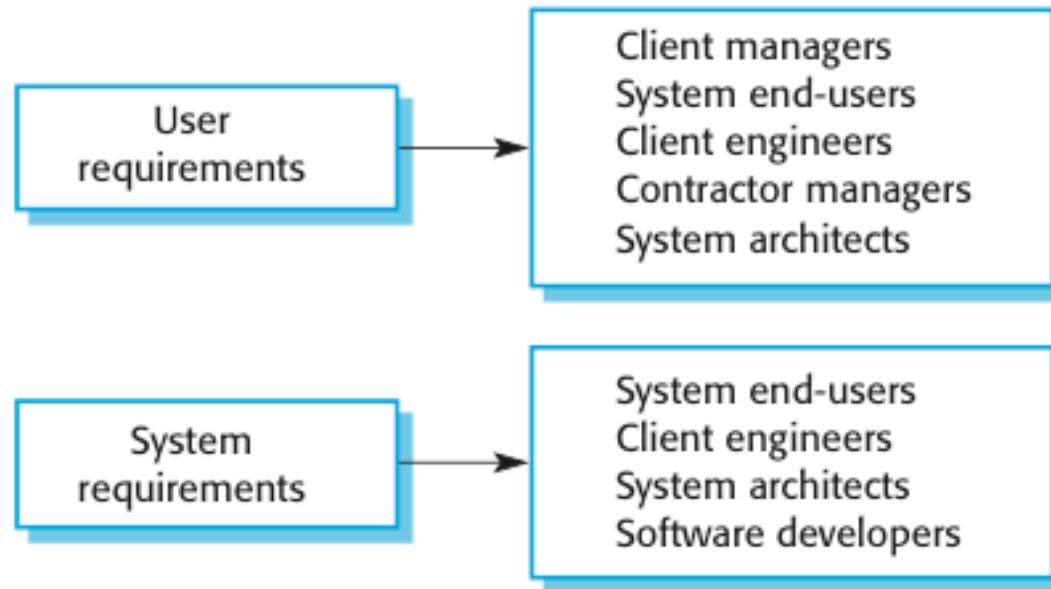
User requirement definition

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

System requirements specification

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

READERS OF DIFFERENT TYPES OF REQUIREMENTS SPECIFICATION



FUNCTIONAL & NON-FUNCTIONAL REQUIREMENTS

Functional requirements

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

Non-functional requirements

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

Domain requirements

- Constraints on the system from the domain of operation

EXAMPLE: FUNCTIONAL REQUIREMENTS FOR THE MHC-PMS*

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

*A mental health case patient management system (MHC-PMS);
A system used to maintain records of people receiving care for mental health problems.

REQUIREMENTS IMPRECISION

Problems arise when requirements are not precisely stated.

Ambiguous requirements may be interpreted in different ways by developers and users.

Consider the term ‘search’ in requirement 1

- User intention – search for a patient name across all appointments in all clinics;
- Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

NON-FUNCTIONAL REQUIREMENTS

These define system properties and constraints e.g. reliability, response time, and storage requirements. Constraints are I/O device capability, system representations, etc.

Process requirements may also be specified mandating a particular IDE (integrated development environment), programming language or development method.

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

Can be grouped into 3 classes ...

NON-FUNCTIONAL CLASSIFICATIONS

Product requirements

- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

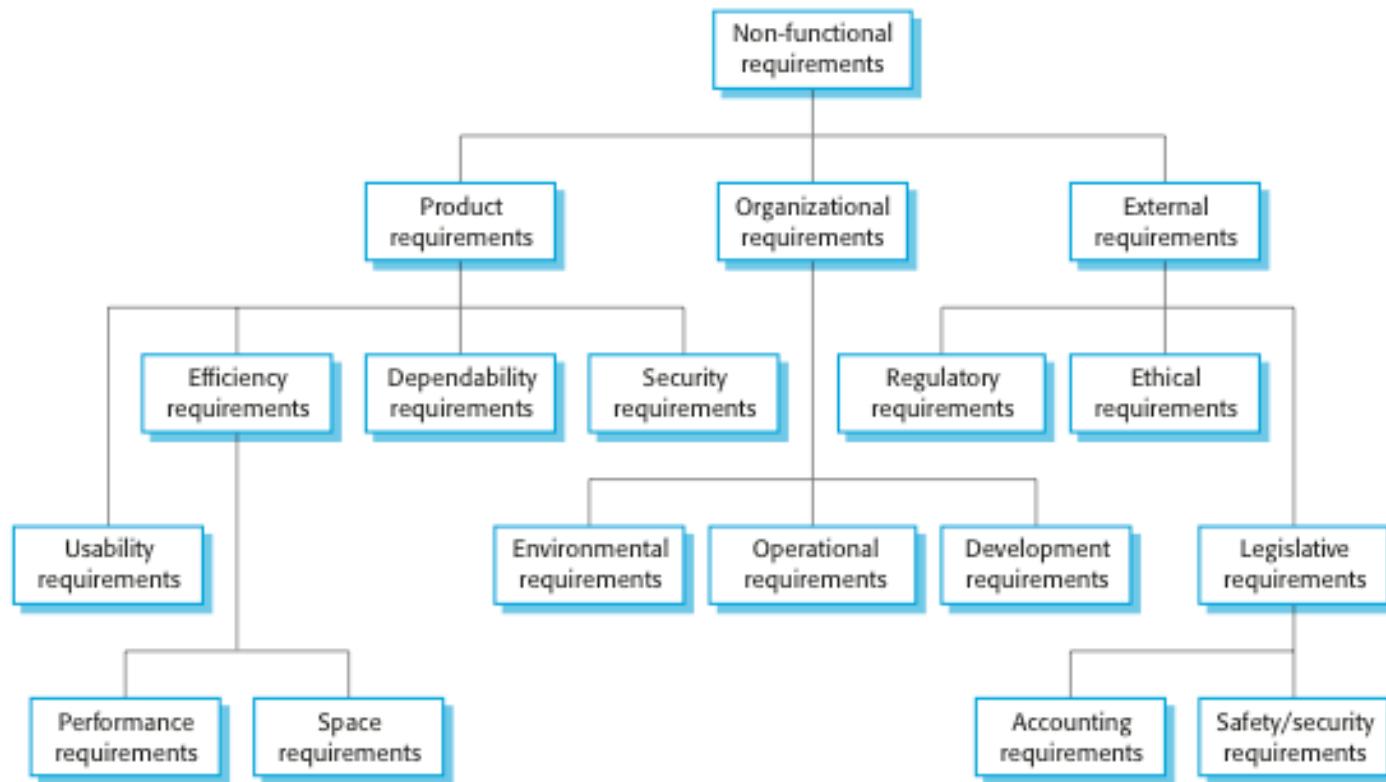
Organisational requirements

- 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.

TYPES OF NONFUNCTIONAL REQUIREMENTS



TYPES OF NONFUNCTIONAL REQUIREMENTS

ROBERTSON AND ROBERTSON (2012)

- 1) **Look-and-Feel requirements.** The spirit of the product's appearance.
- 2) **Usability & Humanity requirements.** The product's ease of use, and any special usability considerations.
- 3) **Performance requirements.** How fast, how safe, how accurate, how reliable, and how available the functionality must be.
- 4) **Operational & Environmental requirements.** The environment on which the product will have to work (e.g., under water), & what considerations must be made for this environment.
- 5) **Maintainability & Support requirements.** Expected changes, and the time allowed to make them.
- 6) **Cultural requirements.** Special requirements that come about because of the people involved in the product's development and operation.
- 7) **Legal requirements.** The laws and standards that apply to the product.
- 8) **Security requirements.** The security and confidentiality of the product.

EXAMPLES OF NONFUNCTIONAL REQUIREMENTS IN THE MHC-PMS

Product requirement

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

Organizational requirement

Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

External requirement

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

METRICS (FIT-CRITERIA) FOR SPECIFYING NONFUNCTIONAL REQUIREMENTS

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

EXAMPLE: USABILITY REQUIREMENTS

The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal)

Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

(Testable non-functional requirement)

DOMAIN REQUIREMENTS

The system's operational domain imposes requirements on the system.

- For example, a train control system has to take into account the braking characteristics in different weather conditions.

Domain requirements may be new functional requirements, constraints on existing requirements, or define specific computations.

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

DOMAIN REQUIREMENTS PROBLEMS

Understandability

- Requirements are expressed in the language of the application domain;
- This is often not understood by software engineers developing the system.

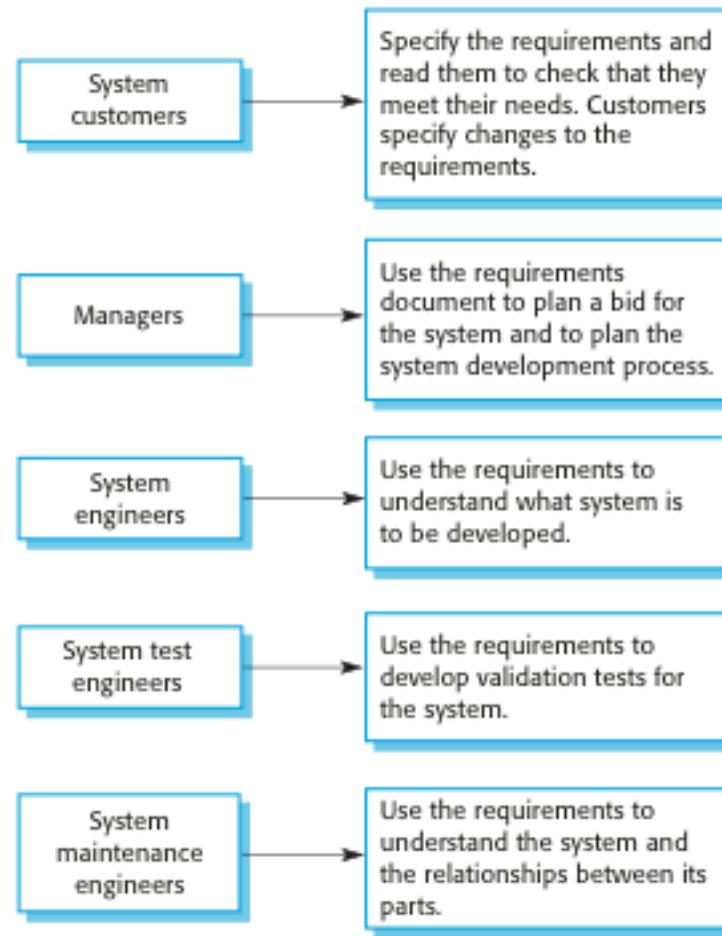
Implicitness

- Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

THE SOFTWARE REQUIREMENTS DOCUMENT

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

TYPICAL USERS OF A REQUIREMENTS DOCUMENT



A TYPICAL STRUCTURE OF A REQUIREMENTS DOCUMENT

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

A TYPICAL STRUCTURE OF A REQUIREMENTS DOCUMENT

Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

REQUIREMENTS SPECIFICATION

The process of writing down the user and system requirements in a requirements document.

User requirements have to be understandable by end-users and customers who do not have a technical background.

System requirements are more detailed requirements and may include more technical information.

The requirements may be part of a contract for the system development

- It is therefore important that these are as complete as possible.

WAYS OF WRITING A SYSTEM REQUIREMENTS SPECIFICATION

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

NATURAL LANGUAGE SPECIFICATION

Requirements are written as natural language sentences supplemented by diagrams and tables.

Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

Guidelines for Writing Requirements:

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

EXAMPLE: REQUIREMENTS FOR THE INSULIN PUMP SOFTWARE SYSTEM

- 3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)
- 3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

STRUCTURED SPECIFICATIONS

An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.

This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business 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.
- Information about the information needed for the computation and other entities used.
- Description of the action to be taken.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

EXAMPLE: A STRUCTURED SPECIFICATION OF A REQUIREMENT FOR AN INSULIN PUMP

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

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

Inputs Current sugar reading (r_2); the previous two readings (r_0 and r_1).

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

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

EXAMPLE: A STRUCTURED SPECIFICATION OF A REQUIREMENT FOR AN INSULIN PUMP

Action

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

Requirements

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

Pre-condition

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

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

Side effects None.

TABULAR SPECIFICATION

Used to supplement natural language.

Particularly useful when you have to define a number of possible alternative courses of action.

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

EXAMPLE: TABULAR SPECIFICATION OF COMPUTATION FOR AN INSULIN PUMP

Condition	Action
Sugar level falling ($r_2 < r_1$)	$\text{CompDose} = 0$
Sugar level stable ($r_2 = r_1$)	$\text{CompDose} = 0$
Sugar level increasing and rate of increase decreasing $((r_2 - r_1) < (r_1 - r_0))$	$\text{CompDose} = 0$
Sugar level increasing and rate of increase stable or increasing $((r_2 - r_1) \geq (r_1 - r_0))$	$\text{CompDose} = \text{round}((r_2 - r_1)/4)$ If rounded result = 0 then $\text{CompDose} = \text{MinimumDose}$

REQUIREMENTS ENGINEERING PROCESSES

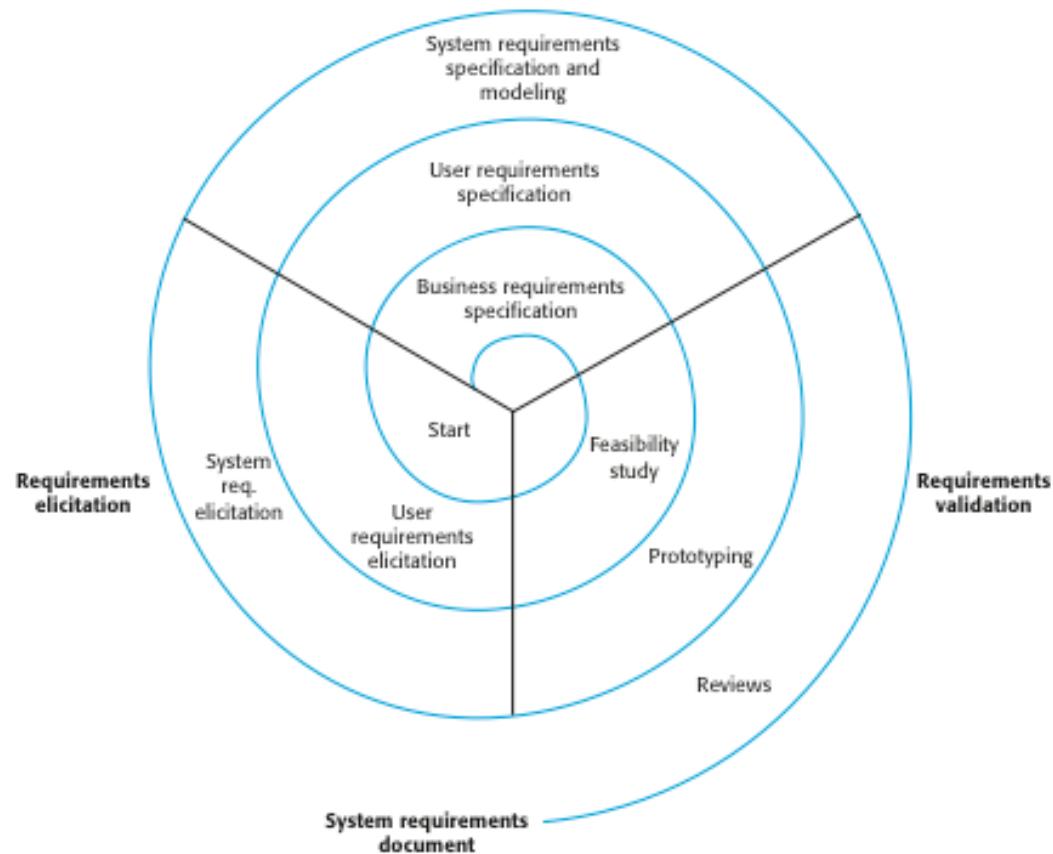
The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.

However, there are a number of generic activities common to all processes

- Requirements elicitation;
- Requirements analysis;
- Requirements validation;
- Requirements management.

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

A SPIRAL VIEW OF THE REQUIREMENTS ENGINEERING PROCESS



REQUIREMENTS ELICITATION AND ANALYSIS

Sometimes called requirements elicitation or requirements discovery.

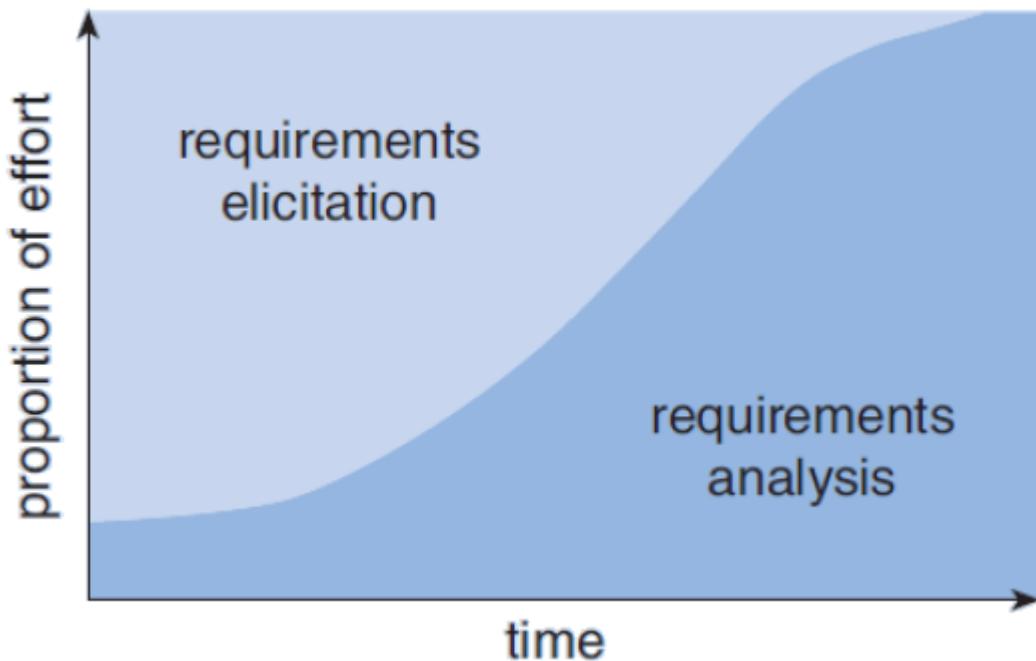
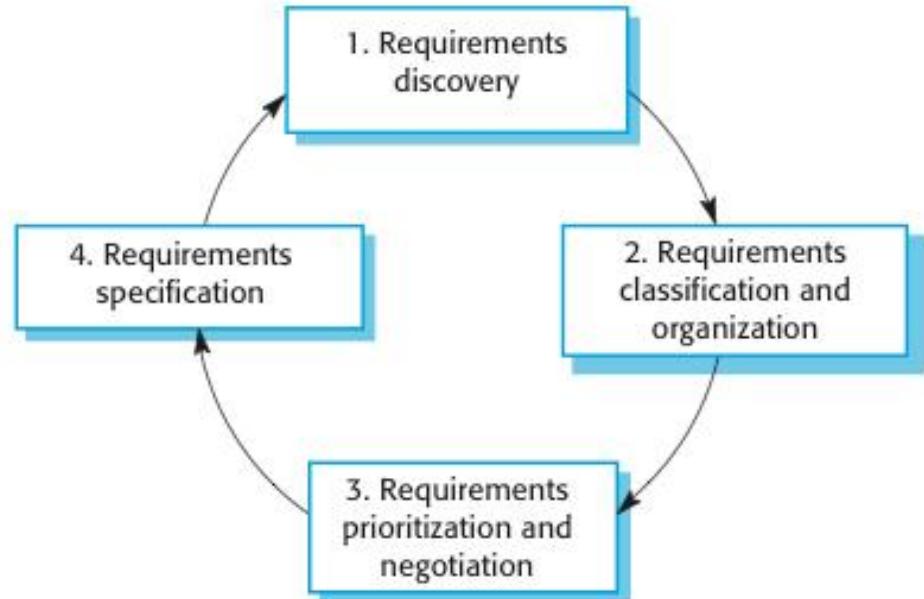
Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.

May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

PROBLEMS OF REQUIREMENTS ANALYSIS

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

THE REQUIREMENTS ELICITATION AND ANALYSIS PROCESS



REQUIREMENTS DISCOVERY

The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.

Interaction is with system stakeholders from managers to external regulators.

Systems normally have a range of stakeholders.

STAKEHOLDERS IN THE MHC-PMS

Patients whose information is recorded in the system.

Doctors who are responsible for assessing and treating patients.

Nurses who coordinate the consultations with doctors and administer some treatments.

Medical receptionists who manage patients' appointments.

IT staff who are responsible for installing and maintaining the system.

A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.

Health care managers who obtain management information from the system.

Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

REQ. DISCOVERY APPROACHES: INTERVIEWING

Formal or informal interviews with stakeholders are part of most RE processes.

Types of interview

- Closed interviews based on pre-determined list of questions
- Open interviews where various issues are explored with stakeholders.

Effective interviewing

- Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
- Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

REQ. DISCOVERY APPROACHES: SCENARIOS

Scenarios are real-life examples of how a system can be used.

They should include

- A description of the starting situation;
- A description of the normal flow of events;
- A description of what can go wrong;
- Information about other concurrent activities;
- A description of the state when the scenario finishes.

REQ. DISCOVERY APPROACHES: USE CASES

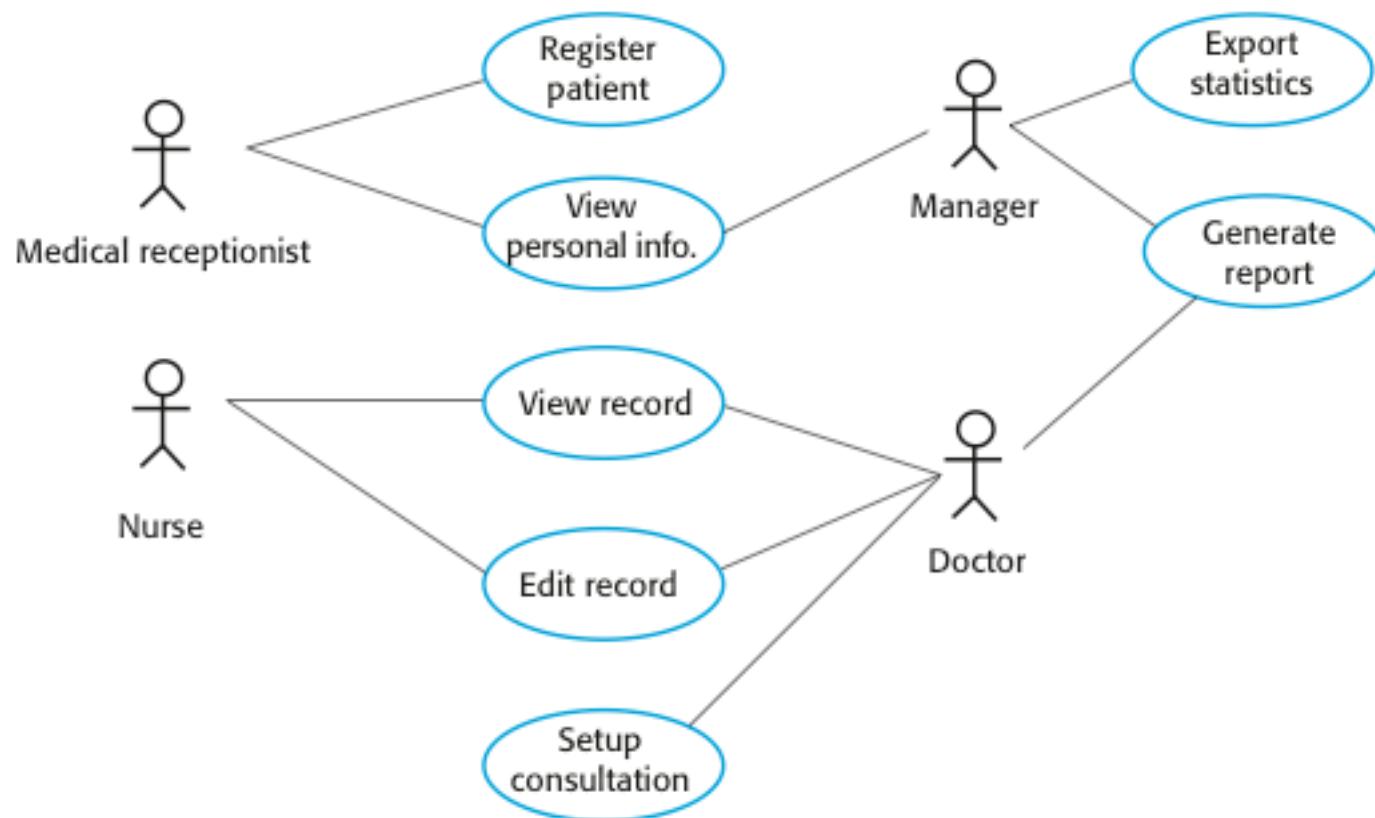
Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.

A set of use cases should describe all possible interactions with the system.

High-level graphical model supplemented by more detailed tabular description (see Chapter 5).

Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

REQ. DISCOVERY APPROACHES: EXAMPLE: USE CASES FOR THE MHC-PMS



REQ. DISCOVERY APPROACHES: ETHNOGRAPHY

A social scientist spends a considerable time observing and analysing how people actually work.

People do not have to explain or articulate their work.

Social and organisational factors of importance may be observed.

Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

REQUIREMENTS VALIDATION

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

Requirements error costs are high so validation is very important

- Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

REQUIREMENTS CHECKING

Validity. Does the system provide the functions which best support the customer's needs?

Consistency. Are there any requirements conflicts?

Completeness. Are all functions required by the customer included?

Realism. Can the requirements be implemented given available budget and technology

Verifiability. Can the requirements be checked?

REQUIREMENTS VALIDATION TECHNIQUES

Requirements reviews

- Systematic manual analysis of the requirements.

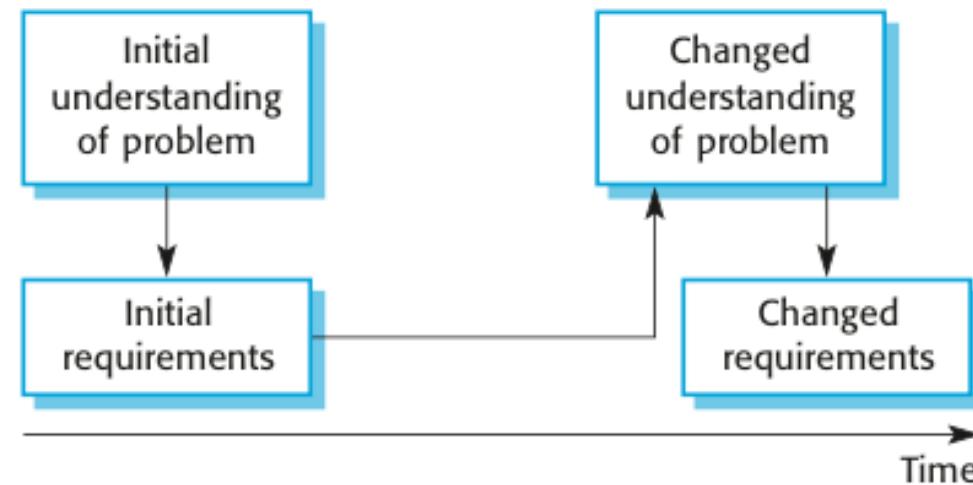
Prototyping

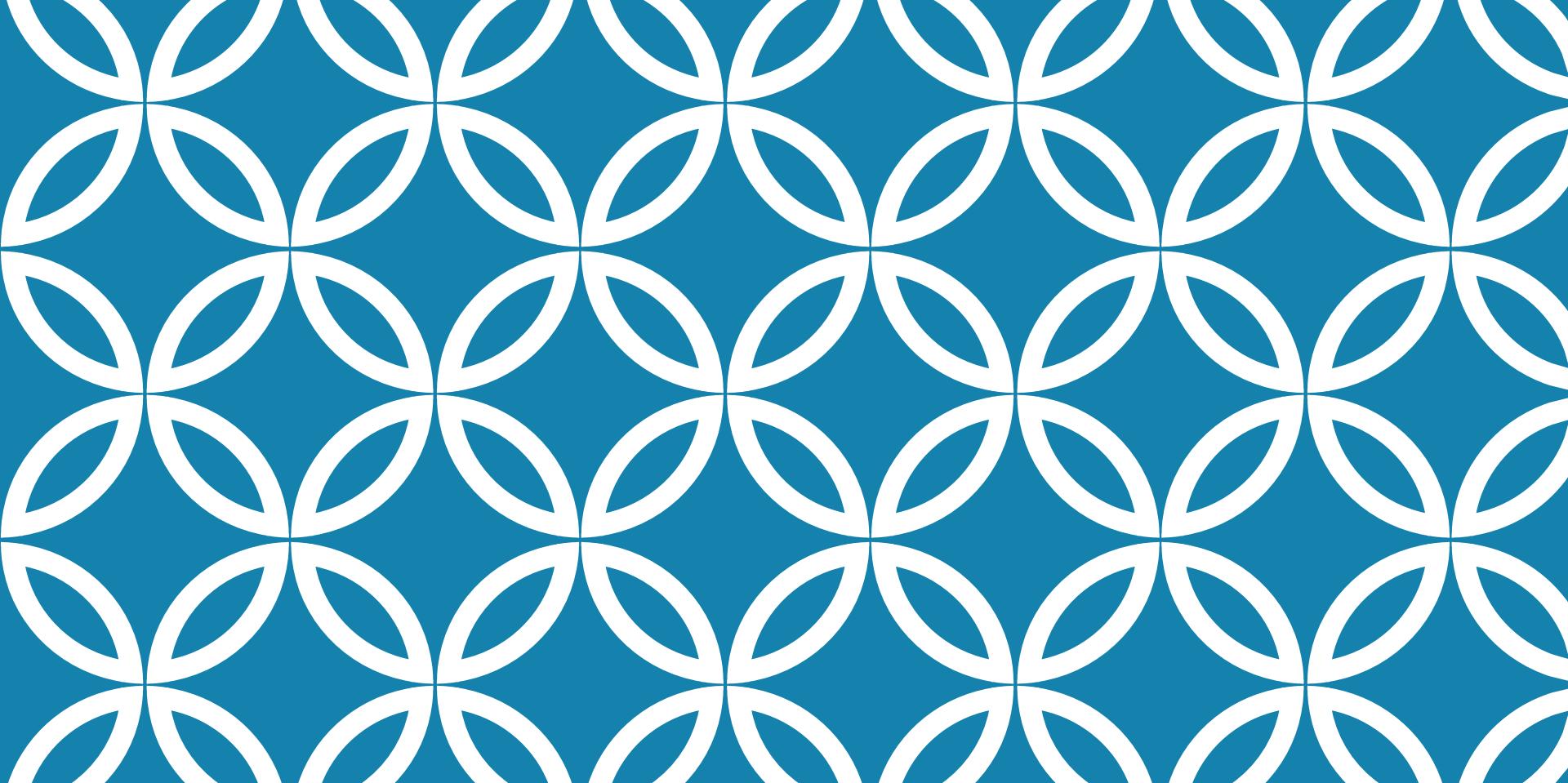
- Using an executable model of the system to check requirements.
Covered in Chapter 2.

Test-case generation

- Developing tests for requirements to check testability.

REQUIREMENTS EVOLUTION





(CS251) SOFTWARE ENGINEERING I

Lecture 3
From Domain to
Requirements; Use-Case &
Activity Diagrams

TOPICS COVERED

- Business Processes Vs. Business Rules
 - Business Rules representation techniques
 - Business Processes representation techniques
- Activity Diagrams
 - Advantages
 - Elements & Symbols
 - Examples
 - Linking activities to user interfaces
- Use-Case Models
 - Advantages
 - Elements & Symbols
 - Descriptions
 - System Boundary
 - Scenarios
 - Generalization Among Actors
 - Relationships between Use-Cases
 - «Include» .. Sharing behaviour
 - «Extend» Alternatives to main scenarios
 - Extending descriptions
 - Avoiding over-complex use case diagrams
 - Use-Case as a planning aid
 - Use-Case .. & Architecture
 - Use-Case .. & Testing

BUSINESS PROCESSES VS. BUSINESS RULES

Business processes define what is done in a business, by whom, in what order, needing which resources, and with what consequences.

Business rules constrain how a business is run.

- There is a clear distinction between the business processes and the constraints.
- Example, in a car rental company:
 - renting a car (business process);
 - the car allocated is the lowest mileage car that is available in the chosen group (business rule);

BUSINESS PROCESSES VS. BUSINESS RULES

Business processes are important to:

- get an understanding of what a business does,
- and to gain the domain expertise needed to develop software solutions in the business context.

Business rules need to be identified and recorded:

1. to provide quick and easy access whenever there is a change.
2. Business rules are the basis of decision making.

Traceability of business rules should be possible from the business needs to the software solution.

BUSINESS RULES REPRESENTATION TECHNIQUES

The language used in expressing business rules should be well defined and structured, so that verification can be made easier if not automatic:

1. Using natural language to make it easy for business people to understand it (.. quite *informal*).
2. Using UML and OCL;

OCL is **Object Constraint Language**, which is a formal language used to represent constraints in UML.

BUSINESS PROCESSES REPRESENTATION TECHNIQUES

Business processes are represented using **UML activity diagrams**.

- An activity diagram shows a process as a set of activities, showing their sequences, and where activities can be carried out in parallel.
- It can also be extended to show which person is responsible for which activity.
- An activity could be a task that a person or a computer might perform.

ACTIVITY DIAGRAMS .. ADVANTAGES

1. Help investigate the **workflow*** (flow of control) from one activity to another.
2. Help in understanding the basic behavior of a system, and understand the business situation.
3. Can be used to model concurrent systems.
4. Can record scenarios of use cases.
5. Can help identify the stages at which each role requires some interaction with the process. This is of particular benefit when we want to investigate the steps that people take in order to do their jobs.

***Workflow** is defined as a sequence of activities that produces a result of observable value.

ACTIVITY DIAGRAMS .. ELEMENTS & SYMBOLS

Element	Meaning	Symbol
Activities	Business processes and tasks	
Start node	Start of process	
End node	End of process	
Transition	move from one activity to another	
Synchronization bar	represents fork and join . Where fork represents the concurrent activities, and join means end of concurrent activities and is used to move to next sequence activity after finishing concurrent ones.	
Decision node	represent alternative ways out of an activity. It has 2 outgoing arcs	
Merge node	brings together alternative flow. It has one outgoing flow.	
Guard	represents the condition (Boolean test); Guards are associated with transitions. A transition takes place only if its guard evaluates to true at the time the flow of control reaches that transition.	[<i>condition</i>]
Swimlane	Swimlanes group activities associated with different roles . Each swimlane shows who is responsible for which set of related activities.	

ACTIVITY DIAGRAMS .. EXAMPLE 1

Suppose, after a hard day's work, you decide to make yourself a hot drink. What do you do to achieve that goal? What are the actions and how do you organise them?

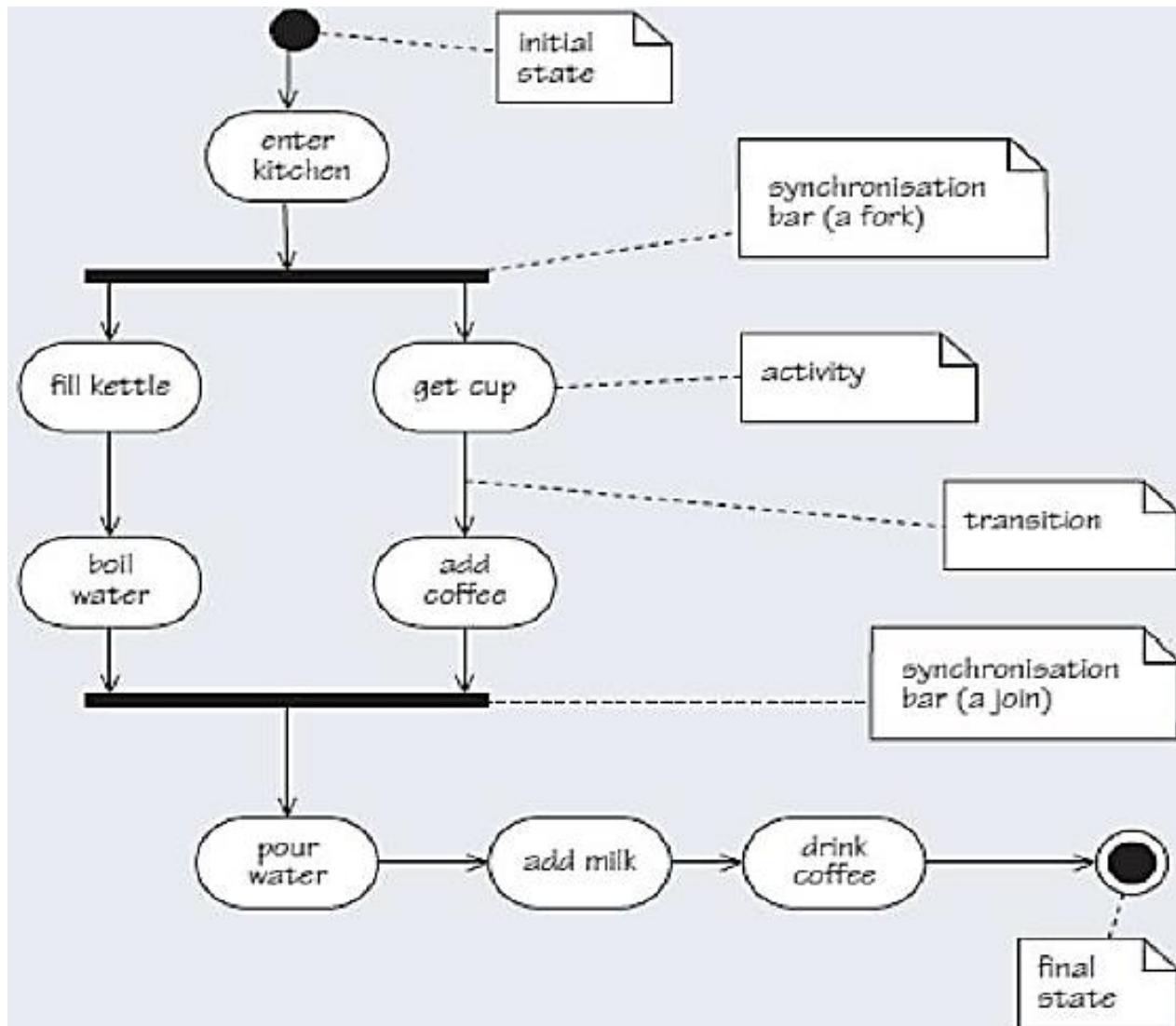
We start with a process called make hot drink. At the centre of this task is boiling water in a kettle.

Usually, there are two steps.

1. Fill the kettle with water.
2. Boil the water.

There are also some decisions to be made, such as the choice of beverage. We will consider instant coffee (as a particular scenario) to begin with. Finally, you need a cup to drink from.

ACTIVITY DIAGRAMS .. EXAMPLE 1



ACTIVITY DIAGRAMS .. EXAMPLE 1

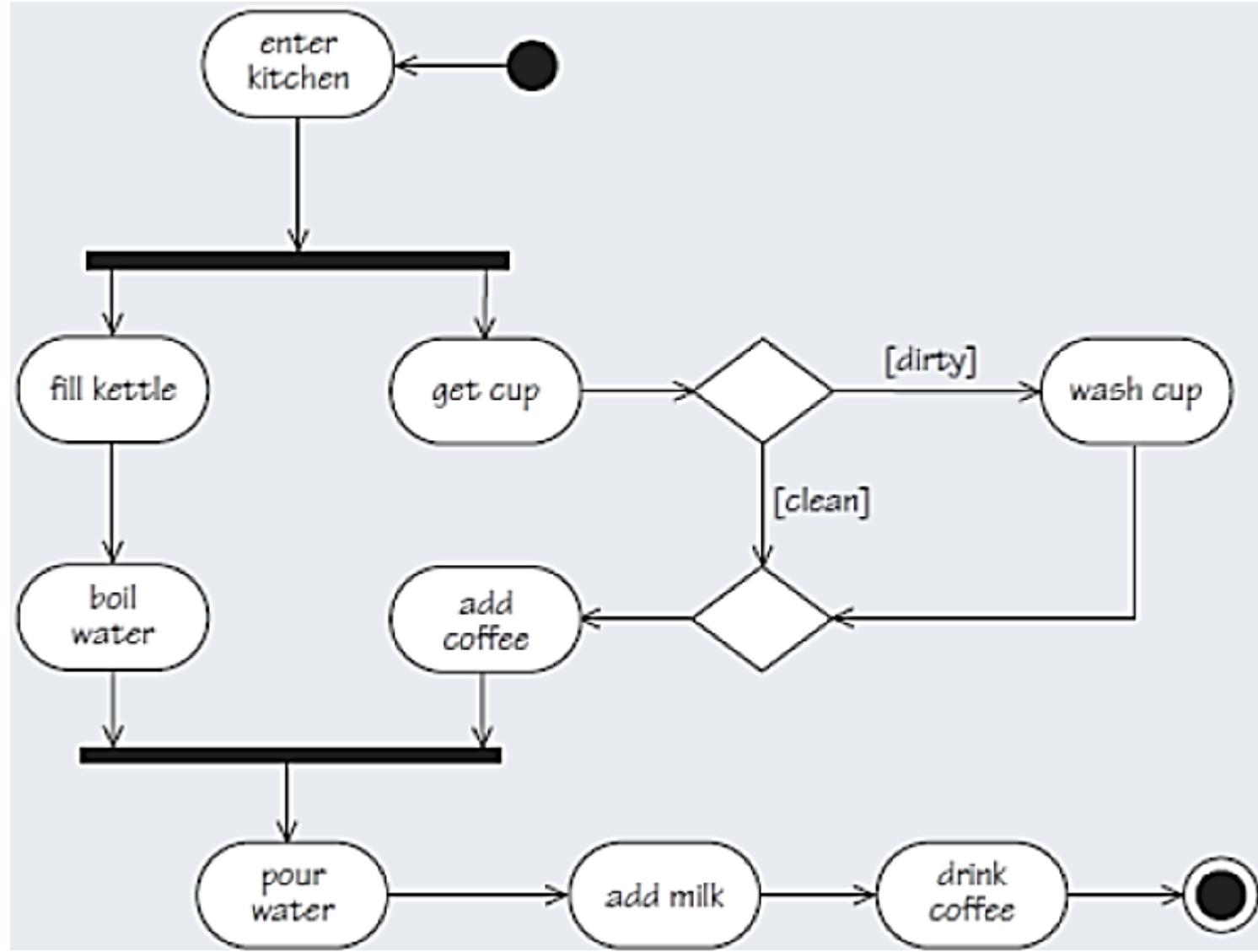
.. ELEMENTS & SYMBOLS

The basic elements of an activity diagram are activities and transitions. The completion of an activity triggers an outgoing transition. Similarly, once the incoming transition has been triggered an activity may commence.

- In an activity diagram, activities are shown as rounded boxes; transitions are shown as lines with arrows.
- There are two predefined activities, start and stop, which is represented as filled circles, a diagram may include more than one stop node if it makes it clearer.
- The Figure shows two synchronisation bars, denoted by thick horizontal lines.
- The upper one denotes a fork and the lower one a join. The upper bar allows for separate activities to be carried out in parallel, so between the two synchronisation bars the two separate strands can be performed concurrently.
- In UML, a note (with annotation) is shown as a rectangle with the top right-hand corner folded down.
- A dashed line is used to attach the note to the model element to which it refers.

ACTIVITY DIAGRAMS .. EXAMPLE 2

Redraw the activity diagram of preparing a cup coffee so that it allows to check if cup is clean continue the process, if not wash the cup.

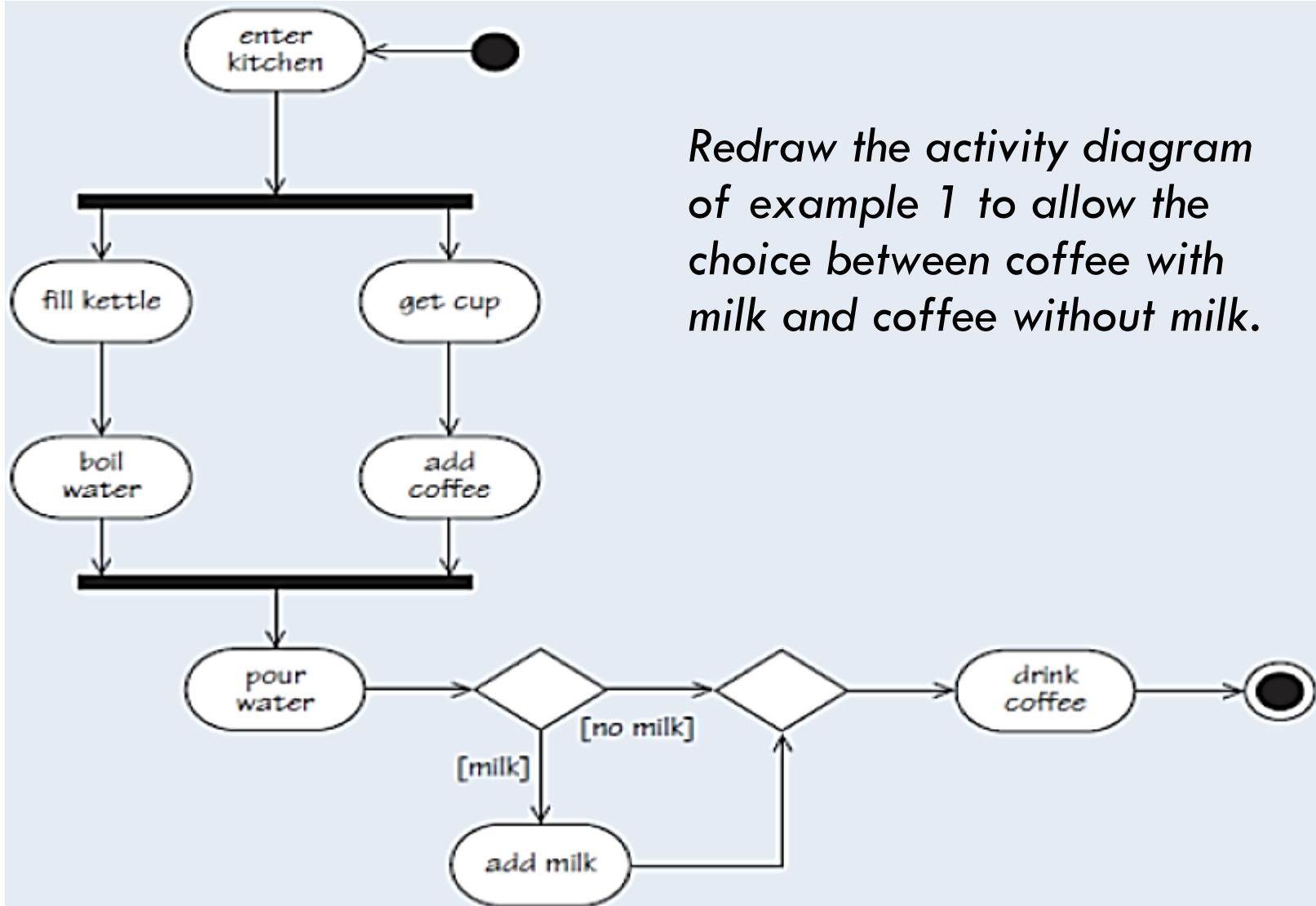


ACTIVITY DIAGRAMS .. EXAMPLE 2

.. ELEMENTS & SYMBOLS

- In Activity 2, the flow between activities is constrained with a Boolean test, known as a guard, which is represented inside a pair of square brackets.
- Each of the transitions leaving the first decision diamond has a guard to determine which path should be taken under a given condition.
- One outgoing transition should always be taken – this means that exactly one of the relevant guards must always evaluate to true.
- A diamond shape is also used as a merge node, which brings together alternative mutually exclusive flows, as shown in Figure 2.
- A merge node will be reached only by one of the alternative flows and has a single outgoing flow.
- An activity has only one transition into and out of it.
- This is enforced by ‘decision’ and ‘merge’ nodes for alternative outgoing and incoming transitions, and by synchronisation bars for concurrent outgoing (fork) and incoming (join) transitions.

ACTIVITY DIAGRAMS .. EXAMPLE 3



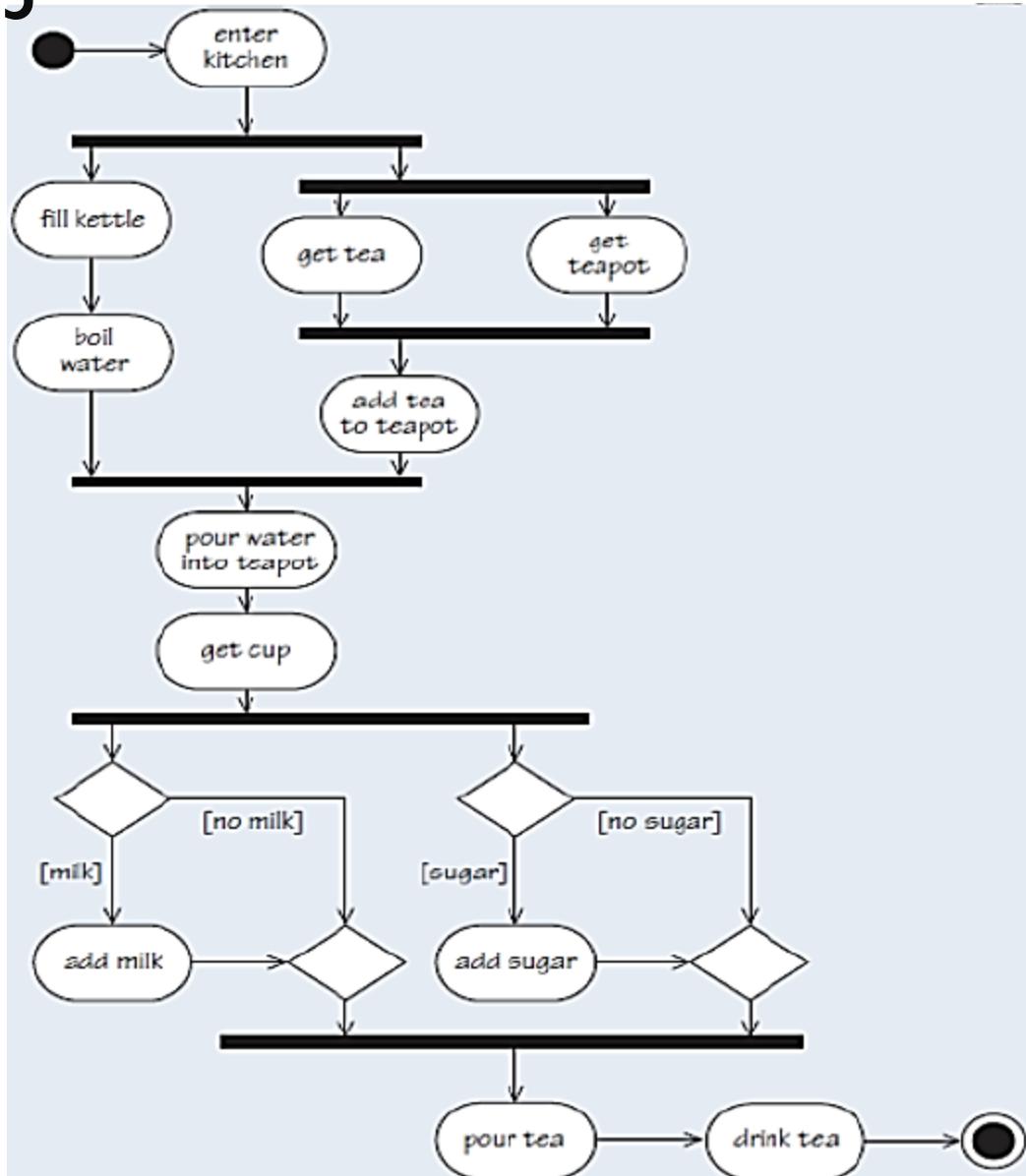
Redraw the activity diagram of example 1 to allow the choice between coffee with milk and coffee without milk.

ACTIVITY DIAGRAMS

.. EXAMPLE 4

Draw an activity diagram to illustrate the preparation of a pot of tea, which replaces the activities relating to coffee shown in example

1. The new diagram should allow the choice of both milk and sugar when the tea is poured.



ACTIVITY DIAGRAMS .. EXAMPLE 5

Consider the lending process from a library :

"A typical lending library keeps a stock of books for the use of its members. Each member can look on the library shelves to select a copy to borrow. Then take out a number of books, up to a certain limit. After a given period of time, the library expects members to return the books that they have on loan.

When borrowing books, a member is expected to wait in queue, then to hand their chosen books to the librarian, who records each new loan before issuing the books to the member. After that the librarian will prepare for next member in the queue. When a book is on loan to a member, it is associated with that member: possession of the book passes from the library to the member for a defined period. The normal loan period for each book is two weeks. If the member fails to bring the book back on or before the due date, the library imposes a fine."

ACTIVITY DIAGRAMS .. EXAMPLE 5

What are the business processes and business rules in the above system?

ACTIVITY DIAGRAMS .. EXAMPLE 5

What are the business processes and business rules in the above system?

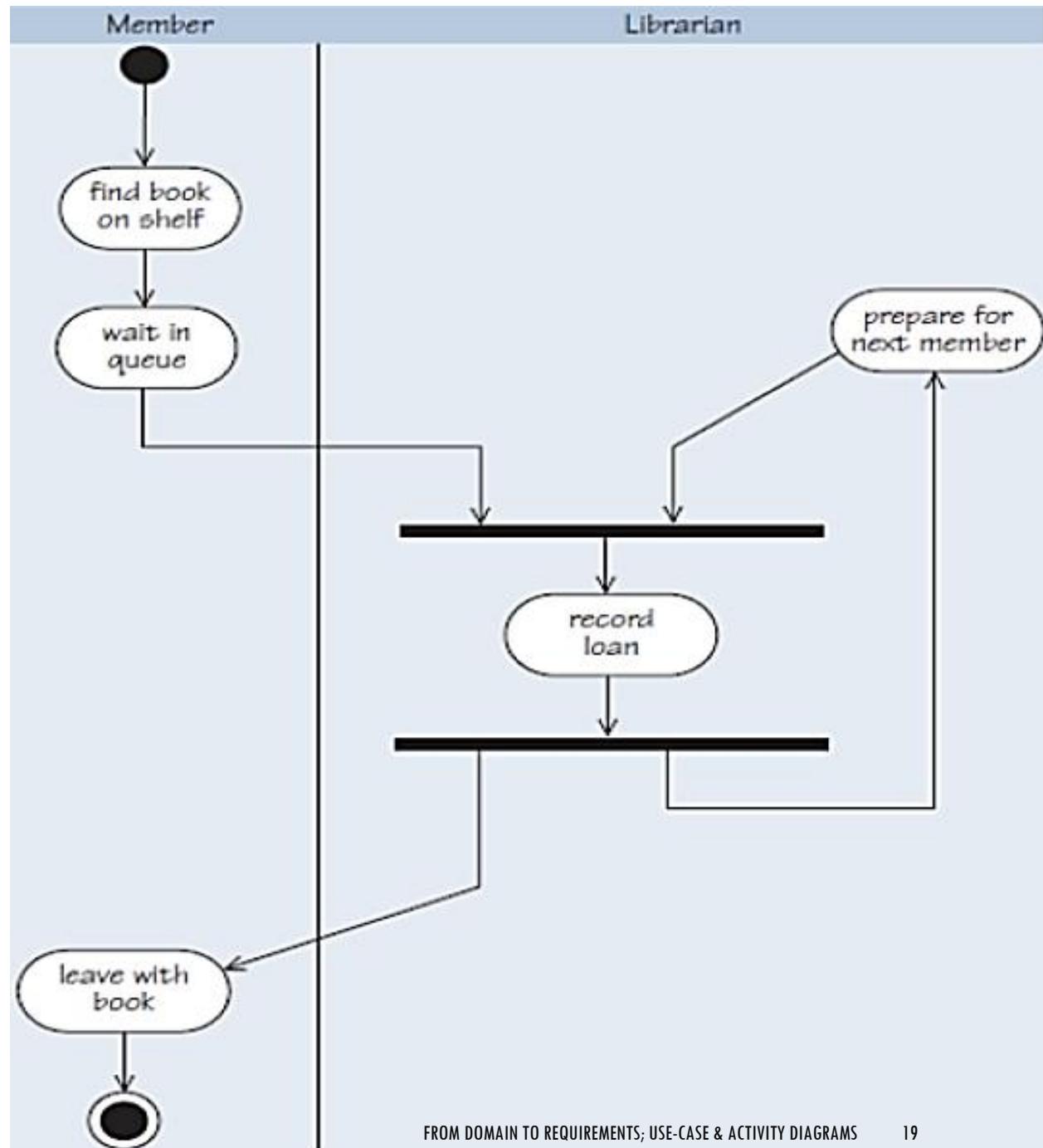
Business processes: find book on a shelf; wait in the queue; issue a book, and return a book.

Business rules:

- There is a limit to the number of books a member can take out.
- A loan is for a period that is normally two weeks.
- Late returns incur a fine.

Draw an activity diagram for issuing a copy of a book.

ACTIVITY DIAGRAMS .. EXAMPLE 5



ACTIVITY DIAGRAMS .. SWIMLANES

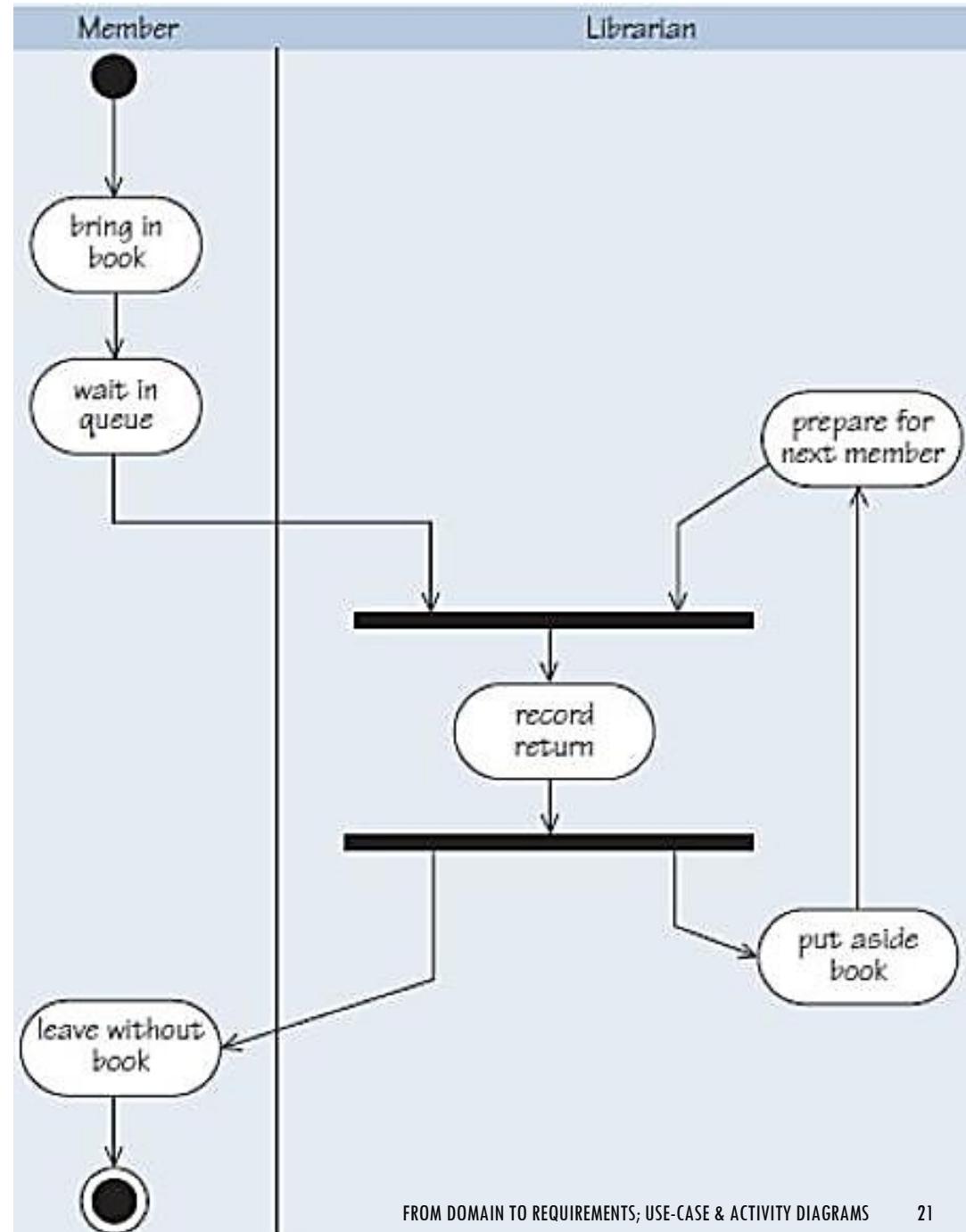
Swimlanes group activities associated with different roles.

- The swimlanes show the role that is responsible for each activity.
- Activity diagrams represent the sequence of activities.
- When you are modelling a workflow that involves more than one role, it is possible to identify which role is responsible for a particular activity.
- An activity diagram can help identify the stages at which each role requires some interaction with the process.

In example 5 The left-hand swimlane contains the activities that a library member performs when borrowing a book, while the right-hand swimlane shows the library's self-service system workflow for the same process ('issue copy of book').

ACTIVITY DIAGRAMS .. EXAMPLE 6

An activity diagram of returning a copy of a book to the library.

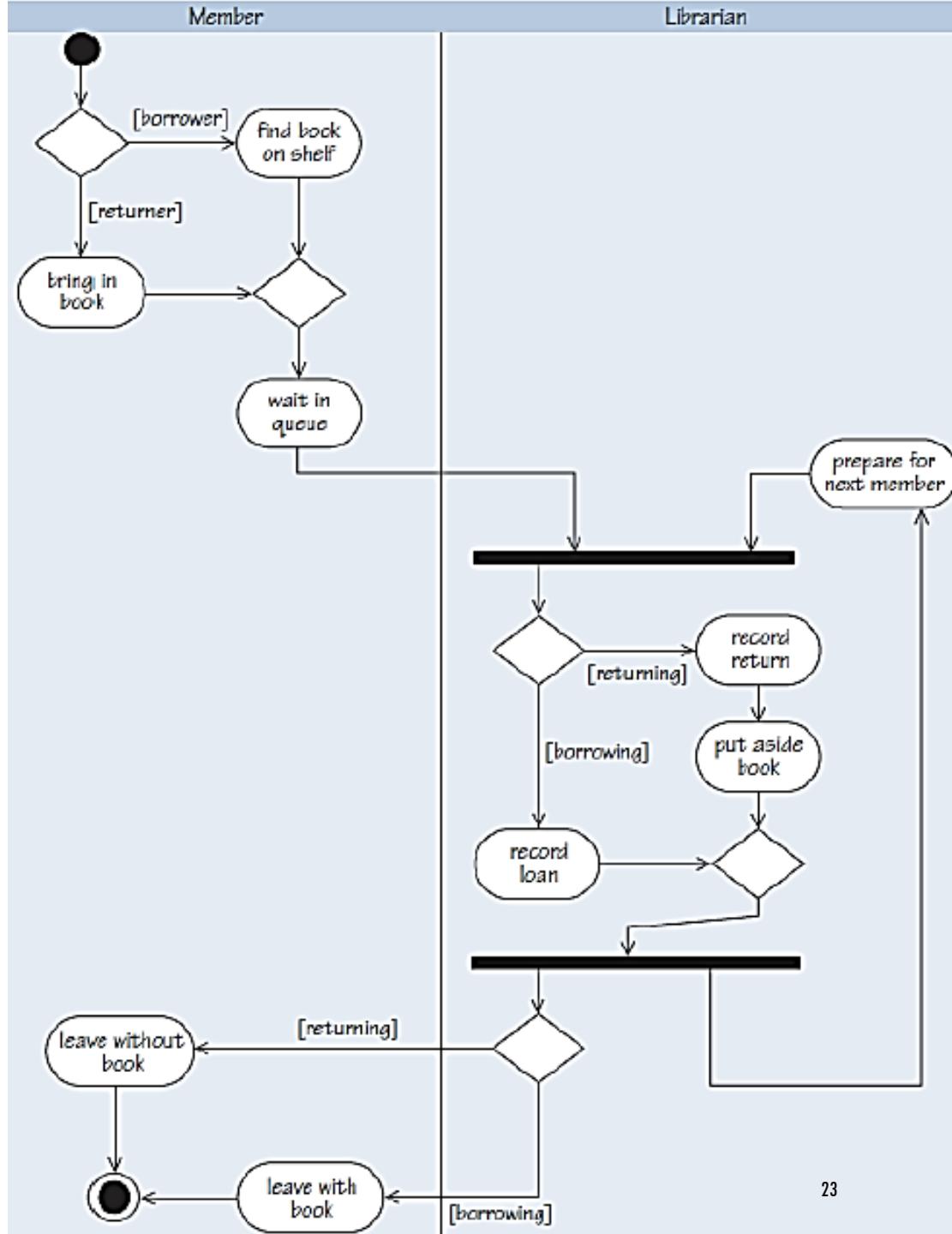


ACTIVITY DIAGRAMS .. EXAMPLE 7

Use examples 5 and 6 to produce a single activity diagram that models the borrowing and returning of books, based on the following assumptions:

1. You do not need to consider fines for overdue books.
2. The librarian places returned books onto a trolley (which will be used to carry the books back to the library's shelves at certain times during the day – an activity which need not be included at this stage).
3. There is just one queue to deal with members.

ACTIVITY DIAGRAMS .. EXAMPLE 7

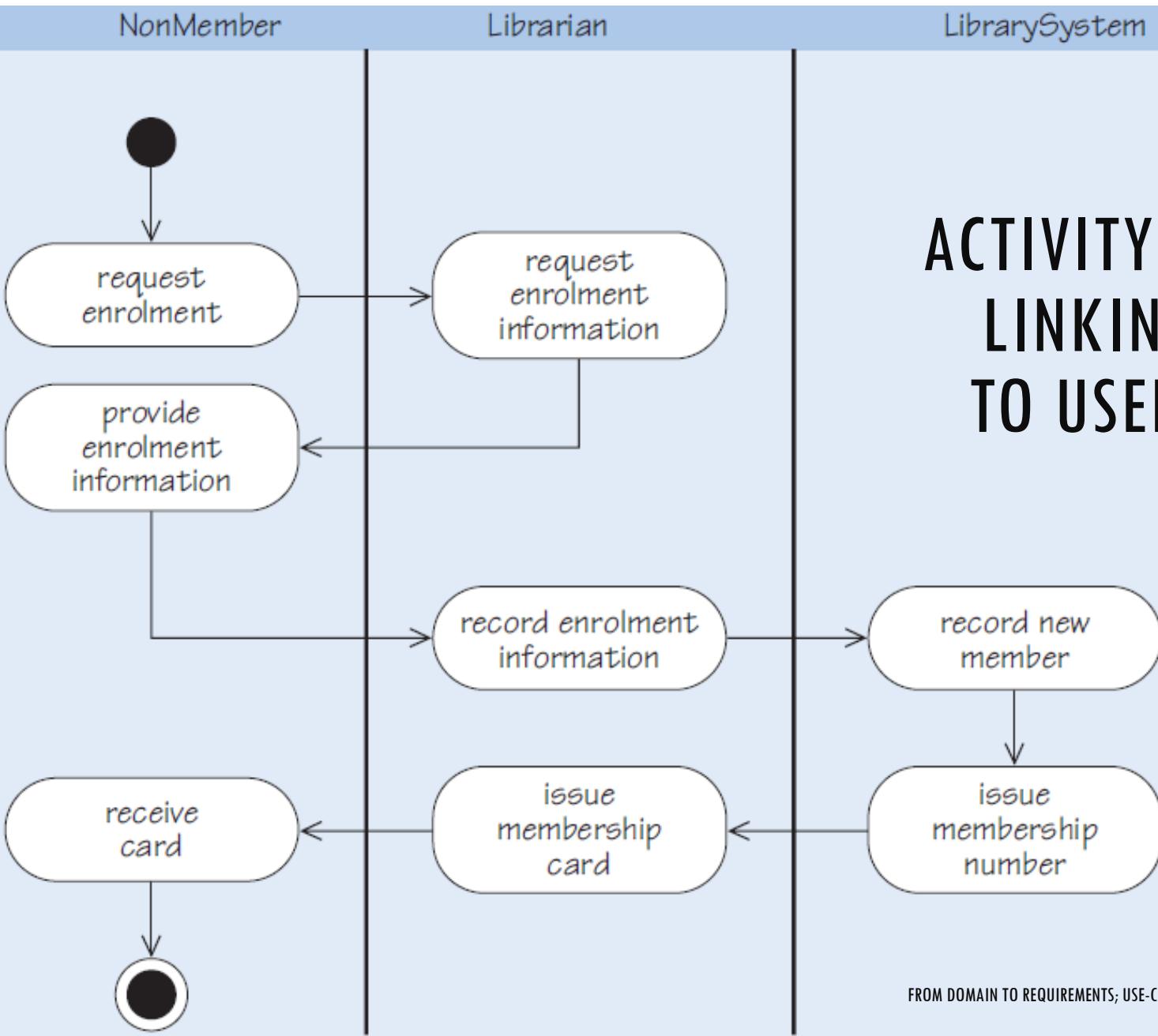


ACTIVITY DIAGRAMS .. LINKING ACTIVITIES TO USER INTERFACES

Before prototyping, the developer needs identify the appropriate interfaces from the users' requirements and determine where and when they will be needed.

- One way to achieve this goal is to use an activity diagram.
- To the users, the interface is the software system: an unacceptable interface can lead to failure.
- The user interface is the link between what the users want and what the developers produce in response

ACTIVITY DIAGRAMS .. LINKING ACTIVITIES TO USER INTERFACES



ACTIVITY DIAGRAMS .. LINKING ACTIVITIES TO USER INTERFACES

The activity diagram shows the interactions involved in enrolling a new member and the activities carried out.

- A nonmember of the library interacts with a librarian. The non-member requests to be enrolled and provides information when requested by the librarian.
- The librarian then uses the system to record the enrolment information. Therefore the librarian needs an interface with the software system for recording the information.
- This interface will be needed when a non-member request to enroll is dealt with by a librarian.
- This transaction is complete when the library system:
 - checks the information entered
 - generates a membership number
 - creates a new member record with the information entered and the membership number
 - informs the librarian of the membership number

Thus the user interface for the use case enroll new member must be available to the librarian when dealing with the enrolment request.

USE-CASE MODELS

.. introduces a means of defining what a proposed system should do from a **user's perspective**.

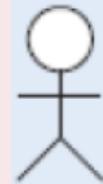
- This can provide the basis for a contract between the customer and the developer
- Your aim is to construct a software system that will meet the needs of its users. Hence you must identify 'who does what'.
- To do this, you explore the problem description, any domain models and the initial set of user requirements to determine the people involved, the work that they do and the events that trigger some work to be done.
- You are likely to identify a variety of people, some of whom may be playing a number of different roles and may be associated with different business events.
- You should ask yourself who the actual users will be and what tasks they must perform with the aid of your software system.

USE-CASE MODELS .. ADVANTAGES

- Capturing and eliciting requirements;
- Representing requirements;
- Planning iterations of development;
- Validating software systems.
- Defining the scope of a system, as they represent the interaction of a system with its environment.
- Acting as a discussion tool between developer and user, and offer a common language for agreeing on the functions of the software system.

USE-CASE MODELS .. ELEMENTS & SYMBOLS

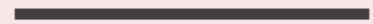
The actors, represented by stick figures;



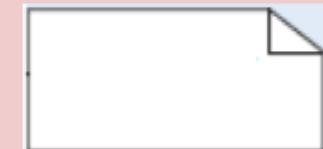
The use cases, represented by ovals, and are used to represent tasks.;



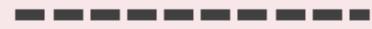
The relationships between actors and use cases, represented by lines.



Note symbol: is used to clarify an aspect or a task, it can be used with any UML diagram.



A dashed line is used to attach the note to the model element to which it refers.



USE-CASE MODELS .. DESCRIPTIONS

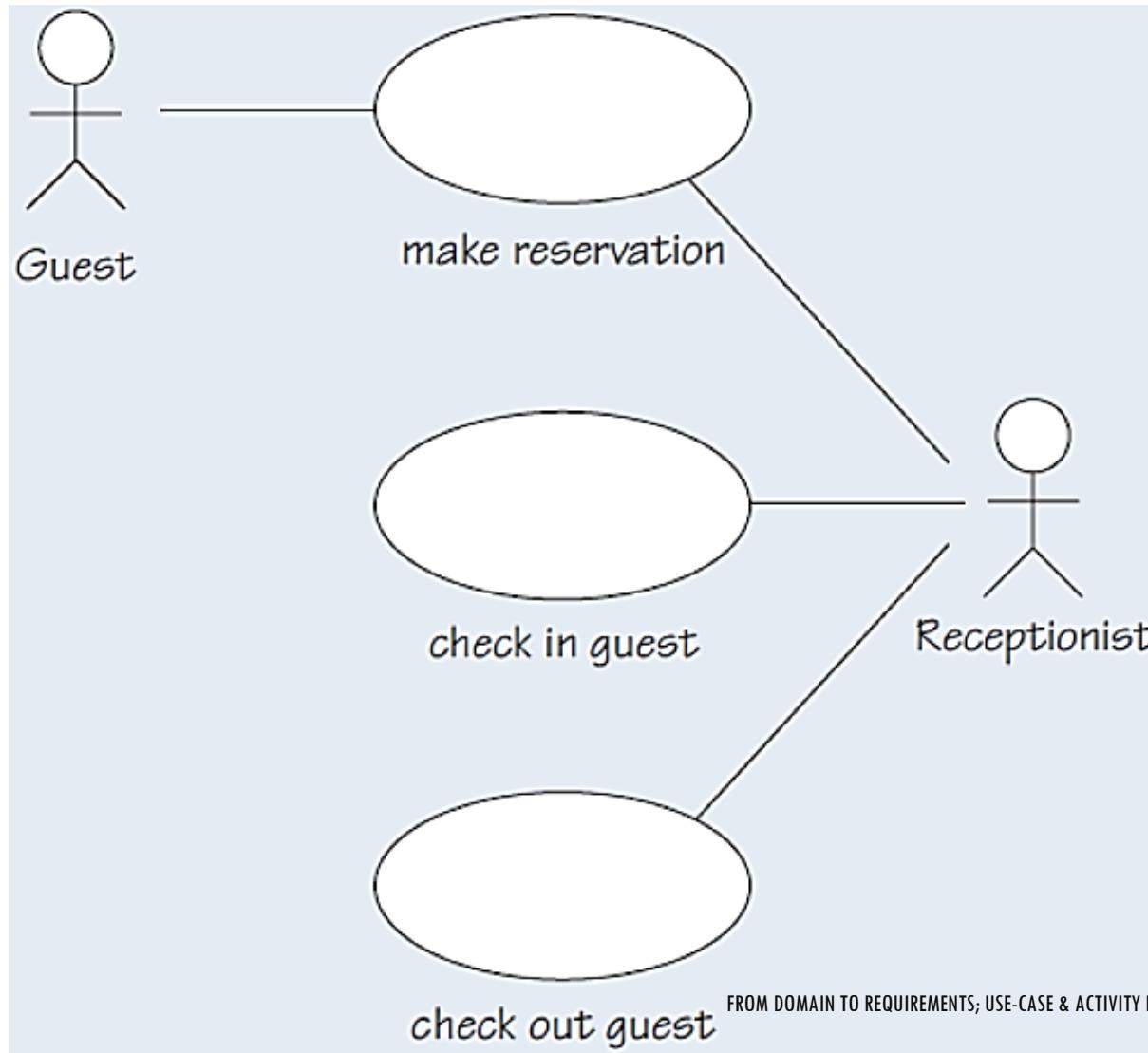
Each use case description should contain the following parts in minimum:

- **Use case identifier and name**
- **Initiator:** name of the actor who initiates the process
- **Pre-condition(s):** a condition that must hold before this use case can be carried out..
- **Post-condition(s):** a condition that must hold after the use case has been completed.
- **Main Success Scenario:** a single sequence of steps that describe the main success scenario. You can number the steps. A scenario is an instance of a use case.
- **Goal:** a short description of the goal of the use case;

Exceptions to the normal behavior for a use case are common, especially where actors decide to cancel a use case without completing it.

USE-CASE DIAGRAMS .. EXAMPLE 1

A simple use case diagram for the main tasks in a hotel system: make reservation, check-in, and check-out, where the reservation can be done by the receptionist or a guest (by Phone/Web).

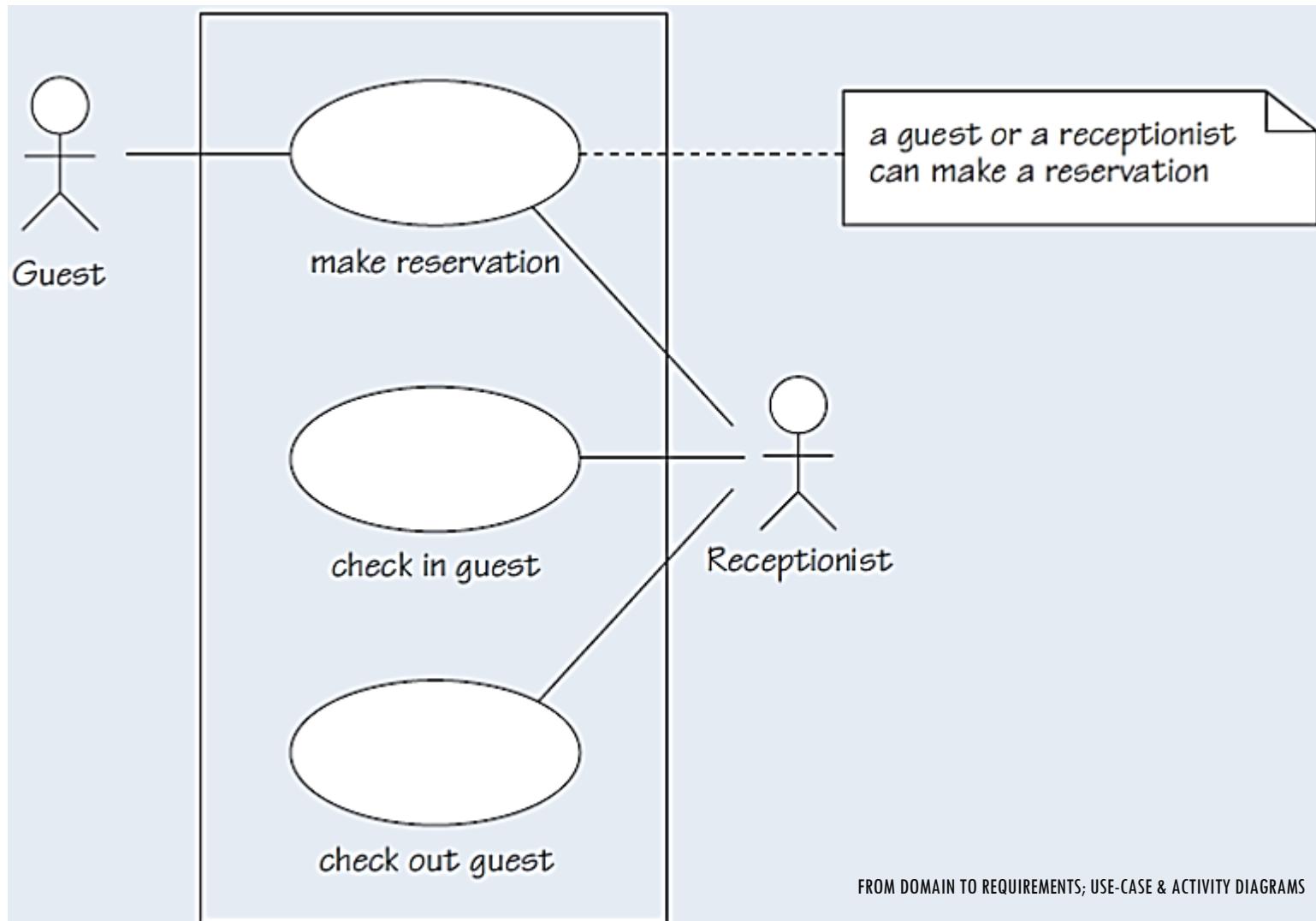


USE-CASE DIAGRAMS .. SYSTEM BOUNDARY

The **system boundary** determines the scope of the system.

- It distinguishes between internal and external components.
- The external components are **actors** and the internal components are the **use cases**.
- In UML it is optional to use system boundary notation.
- A solid box drawn around the use cases with the actors located outside it represents the system boundary.
- We use system boundary notation when the system is complex and includes several subsystems.

USE-CASE DIAGRAMS .. SYSTEM BOUNDARY



USE-CASE DESCRIPTIONS

A **scenario** describes a sequence of interactions between the system and some actors.

- In Each use case there is a set of possible scenarios. Where the main scenario is the successful scenario where nothing goes wrong and the use case is achieved.

For example: there are two scenarios for making reservation in a hotel:

Main Success Scenario:

- The guest wants to reserve a double room at the Hotel for 14 July. A double room is available for that date, and the reservation is done.

Unsuccessful Scenario:

- The guest wants to reserve a single room at the Hotel for the first week of August. There is no single room that is free for seven days in August, but there is one room available for four days and another one for the following three days. The system presents that option to the guest, who rejects it.

USE-CASE DESCRIPTIONS

What pre and post condition(s) you can obtain from the below description of a hotel check-in process?

Upon arrival, each guest provides the reservation number for his or her reservation to the hotel's receptionist, who enters it into the software system. The software system reveals the details of that reservation so that each guest can confirm them. The software system allocates an appropriate room to that guest and opens a bill for the duration of the stay. The receptionist issues a key for the room.

Precondition(s): There must be a reservation for the guest, and there must be at least one room available (of the desired type), and the guest must be able to pay for the room.

Post-condition(s): The guest will have been allocated to a room for the period identified in the reservation, the room will have been identified as being in use for a specific period, a bill will have been opened for the duration of the stay, and a key will have been issued.

USE-CASE DESCRIPTIONS

Table 1 A textual description of a use case in the hotel domain

Identifier and name	UC1 <i>make reservation</i>
Initiator	<i>Guest or Receptionist</i>
Goal	A room in a hotel is reserved for a guest.
Precondition	None (i.e. there are no conditions to be satisfied prior to carrying out this use case).
Postcondition	A room of the desired type will have been reserved for the guest for the requested period, and the room will no longer be free for that period.
Assumptions	The expected initiator is a guest (using a web browser to perform the use case) or a receptionist. The guest is not already known to the hotel's software system (see step 5).
Main success scenario	
1	The guest/receptionist chooses to make a reservation.
2	The guest/receptionist selects the desired hotel, dates and type of room.
3	The hotel system provides the availability and price for the request. (An offer is made.)
4	The guest/receptionist agrees to proceed with the offer.
5	The guest/receptionist provides identification and contact details for the hotel system's records.
6	The guest/receptionist provides payment details.
7	The hotel system creates a reservation and gives it an identifier.
8	The hotel system reveals the identifier to the guest/receptionist.
9	The hotel system creates a confirmation of the reservation and sends it to the guest.

USE-CASE SCENARIOS

For each use case there is a set of possible scenarios. A scenario is an instance of a use case. A scenario describes a sequence of interactions between the system and some actors. Here are two examples of scenarios.

- A member of a lending library wishes to borrow a book, and is allowed to do that as long as they have no outstanding loans.
- Another member wishes to borrow a book, but has exceeded the quota for the number of books that can be borrowed.

In each scenario the member wishes to borrow a book, but both the circumstances and outcomes of events are different in each instance.

A use case includes a complex set of requirements that the system must meet in order to cope with every eventuality.

The main success scenario shows the steps normally followed to achieve the stated goal of the use case. But there can be other scenarios for the same use case, each one having different outcomes depending upon circumstances.

USE-CASE MODELS .. GENERALIZATION AMONG ACTORS

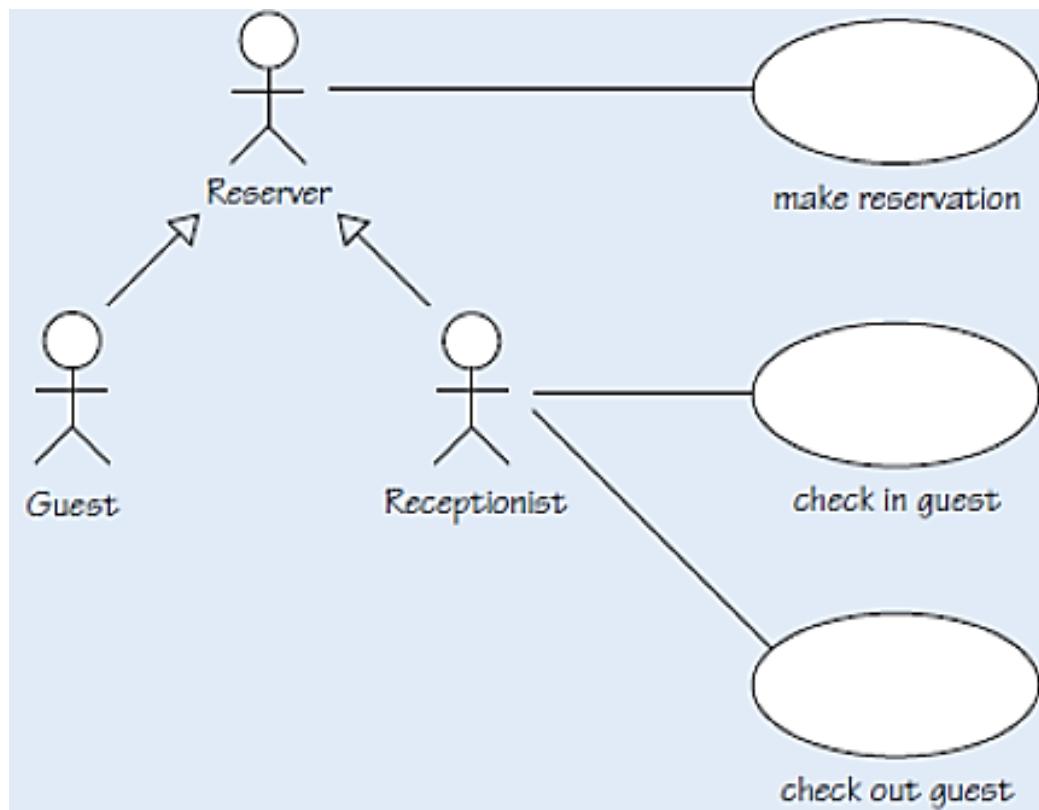
UML provides a notation to use generalisation between actors.

- When two actors share the same behaviour (interacting with the same use cases) and one of them has some extra behaviour, then the common behaviour can be associated with a generalised actor and the more specific behaviour with the specialised actor.

For example, in the hotel system both the receptionist and the guest are allowed to make reservation.

- Guest can make reservation through online system;
- the receptionist can do reservation on behalf of guest when a guest do reservation through phone or a fax.

USE-CASE MODELS .. GENERALIZATION AMONG ACTORS



We introduce a new actor called Reserver associated with the make reservation use case.

- The actors Guest and Receptionist specializes Reserver.
- Guest can do the same things that a Reserver can, but may do other things too.
- By using the open-headed arrow from Guest to Reserver, you are saying that a Guest can do the same things that a Reserver can, but may do something else that a Reserver does not. Reserver is a generalised actor and Guest a specialised one.

USE-CASE MODELS .. RELATIONSHIPS BETWEEN USE-CASES

Use cases can be related to one another. There are two very common and important forms of relationship between use cases: **inclusion** and **extension**.

Inclusion: This is when two or more use cases have a common area of functionality that can be factored out as a distinct use case.

The new use case can be used by each of the original use cases, so avoiding duplication.

Extension: This is when a use case has a main success scenario but also alternative scenarios which demand a variation on the original use case – different or additional actions.

Each variation can be separated out as a use case that is distinct from but related to the original use case.

An extension is conditional while an inclusion is not.

In UML we use «include» and «extend» **stereotypes** for representing inclusion and extension.

USE-CASE MODELS ..

«INCLUDE» .. SHARING BEHAVIOUR

In the process of eliciting and specifying requirements, you may find a certain amount of common behaviour in two or more of your use cases.

- You can record the shared behaviour in a new use case and connect it to the use cases that it came from with a dashed arrow (indicating a dependency relationship) pointing from the original use case to the new one.
- Hence the dependency arrow is labelled with the «**include**» stereotype.

USE-CASE MODELS ..

«INCLUDE» .. EXAMPLE

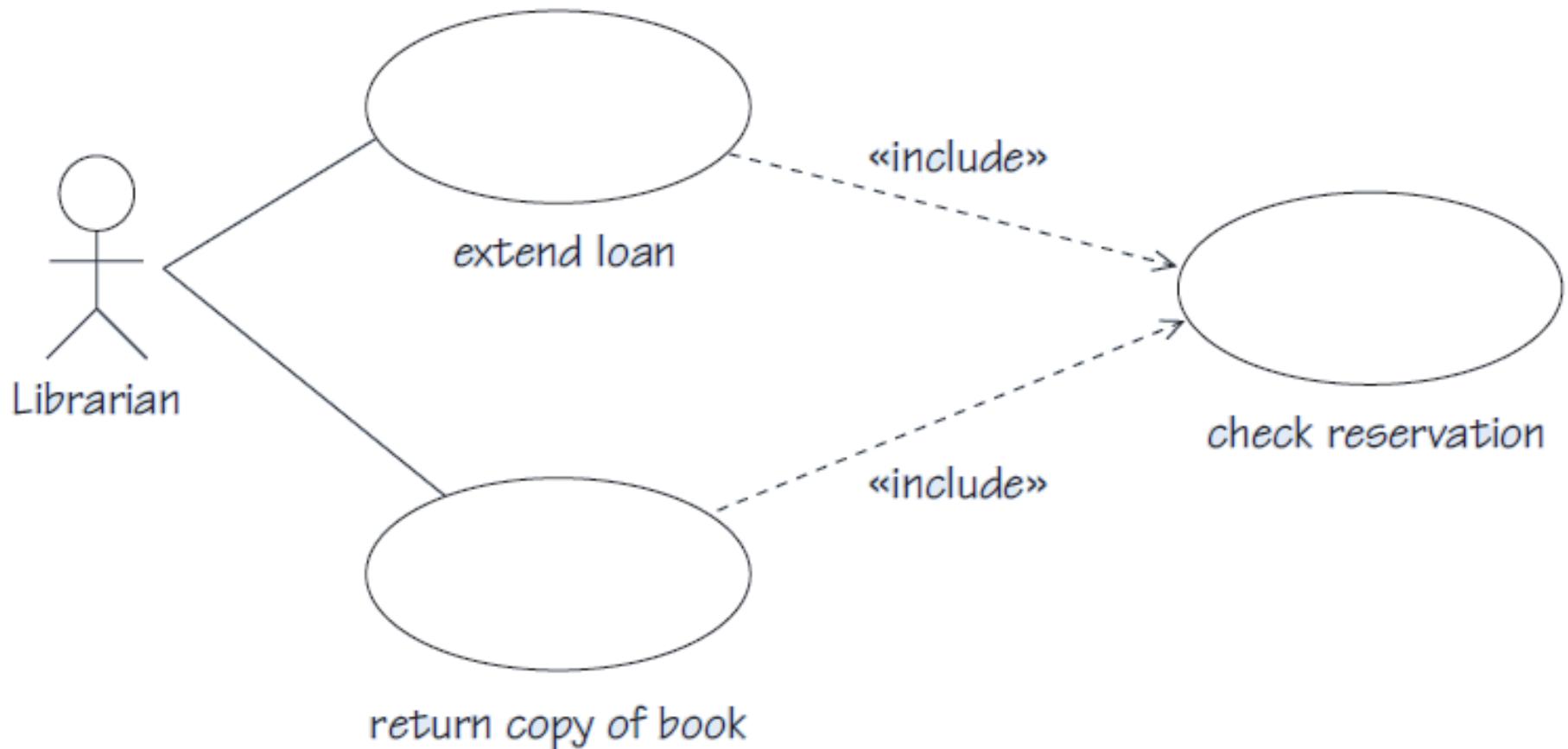
Members of a lending library can borrow a certain number of books and, for any book borrowed, renew the loan up to three times. A loan can only be renewed if there are no existing reservations for that book by other members, when all other copies are on loan.

Let us assume that when a member returns a book there must be a check to see whether there are outstanding reservations for that book by other members. In this case the book will not be returned to the shelf, but will instead be assigned to the highest priority reservation.

This same check needs to be carried out when a member extends a loan. The reservation check is a shared piece of behaviour, a common scenario, which can be developed separately as the check reservation use case.

Note that this is **unconditional behaviour** – the reservation check must be performed whenever a loan is extended or a book is returned. A dependency arrow has a source (where it comes from) and a target (where it goes to).

USE-CASE MODELS .. «INCLUDE» .. EXAMPLE



USE-CASE MODELS ..

«EXTEND» ALTERNATIVES TO MAIN SCENARIOS

The «extend» stereotype indicates a conditional extension to the original use case, known as **alternative behaviour**.

- This is used to illustrate a case where there are two or more significantly different scenarios, so that the main case and the additional **subsidiary cases** are clearly differentiated. The main purpose of this classification is to separate out a special case.
- You should add a condition to each extension – by a note for instance – to specify when the variant behaviour will be included.
- This could be done with either a note or an extension point.
- The new use case (Alternative one) depends on the original. This is a **conditional behavior**.
- The new use case (the source) is connected to the original (the target) with a dashed arrow pointing to the original and labelled with the stereotype «extend».

USE-CASE MODELS ..

«EXTEND» .. EXAMPLE

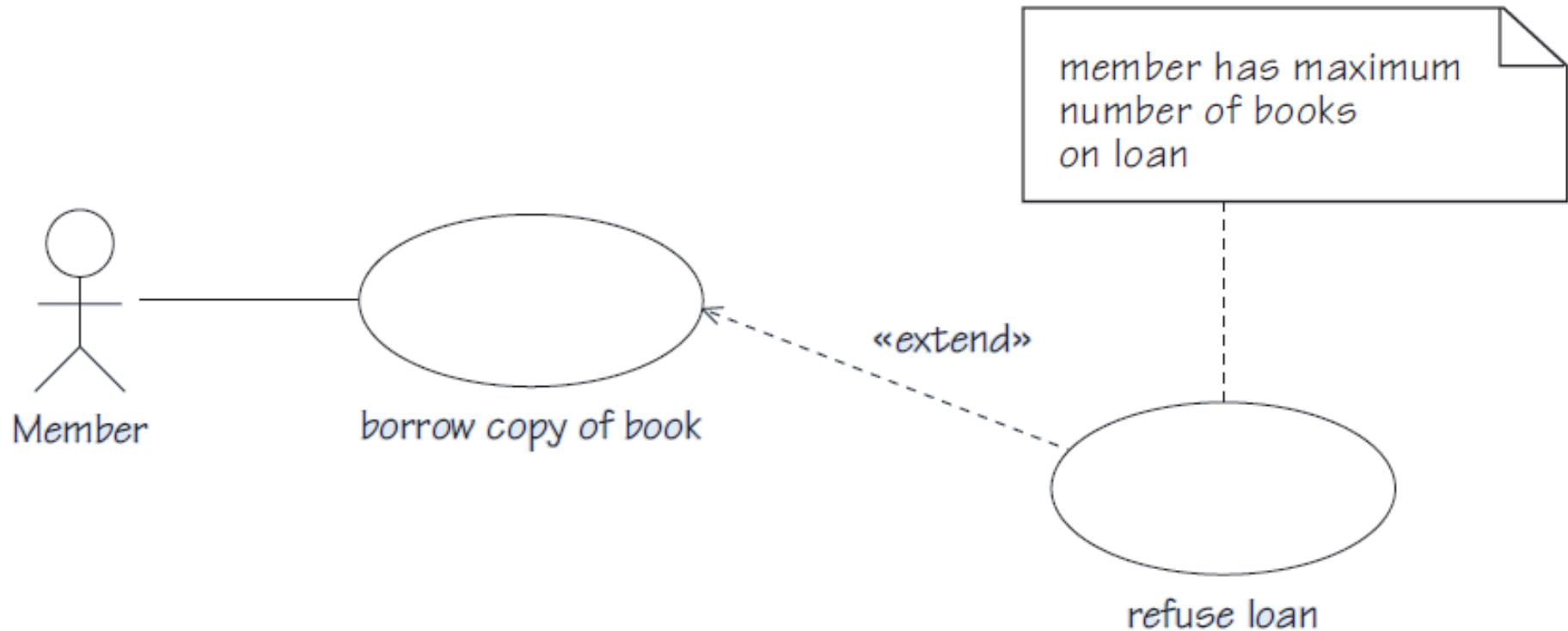
A typical lending library will set an upper limit on the number of books that its members can borrow at any one time. A librarian will not be allowed to issue a copy of a book if that limit would be exceeded.

- The next figure shows a fragment of a use case diagram that identifies a new use case, refuse loan, as it is only performed when the member would have too many books.

Note that: the refuse loan is a **conditional behavior** which is only performed when the member would have too many books.

The note symbol is used to record the event that trigger the subsidiary use case.

USE-CASE MODELS .. «EXTEND» .. EXAMPLE



USE-CASE MODELS .. EXTENDING DESCRIPTIONS

Table 3 Extending the description of a use case in the hotel domain

UC1 <i>make reservation</i>	
Identifier and name	
Initiator	<i>Reserver</i> (may be a <i>Guest</i> or a <i>Receptionist</i>)
Goal	A room in a hotel is reserved for a guest.
Precondition	None
Postcondition	A room of the desired type will have been reserved for the guest for the requested period, and the room will no longer be free for that period.
Assumptions	The expected initiator is a guest (using a web browser to perform the use case) or a receptionist. The guest is not already known to the hotel's software system (see step 5).
Main success scenario	
1	The reserver chooses to make a reservation on behalf of a potential guest.
2	The reserver selects the desired hotel, dates and type of room.
3	The hotel system provides the availability and price for the request. (An offer is made.)
4	The reserver agrees to proceed with the offer.
5	The reserver provides identification and contact details for the hotel system's records.
6	The reserver provides payment details.
7	The hotel system creates a reservation and gives it an identifier.
8	The hotel system reveals the identifier to the reserver.
9	The hotel system creates a confirmation of the reservation and sends it to the guest identified by the reserver.
Extensions	
3.a	<i>room not available</i>
3.a.1	The hotel system offers alternative dates and types of room.
3.a.2	The guest selects from the alternatives or declines the offer.
5.a	<i>guest already on record</i>
5.a.1	Resume at step 6.

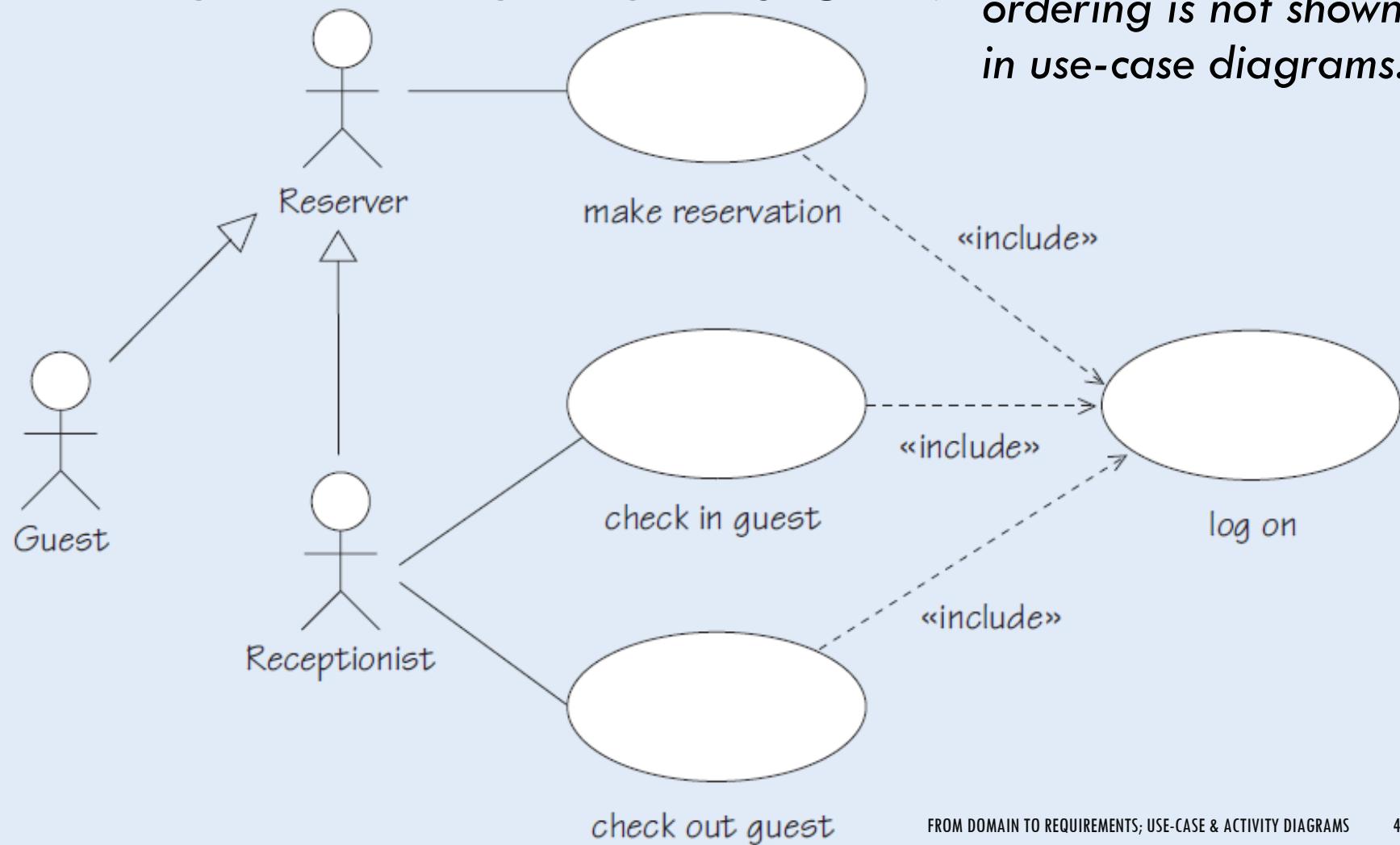
USE-CASE MODELS .. TO EXTEND OR TO INCLUDE?

We could show the log on use-case as a component of every use case that is associated with an actor.

Extend or Include?

USE-CASE MODELS .. TO EXTEND OR TO INCLUDE?

Note that: there is no implication of ordering: time ordering is not shown in use-case diagrams.



USE-CASE MODELS .. AVOIDING OVER-COMPLEX USE CASE DIAGRAMS

The general approach to managing complexity is to partition a problem into sub-problems, and use abstraction to reduce the detail and extract what is most significant to the problem.

Reduce the complexity of your use-case diagram by:

1. redrawing it at a higher level of abstraction
2. splitting it up into smaller modules.

Example: In the case of the hotel chain, we might partition our solution into three sub-problems, usually called packages (a way of grouping cases):

1. reservations, 2. checking guests in and out of their rooms, 3. system access

Each package may then be assigned to a separate developer for implementation, the project team must then deal with the dependencies between the three packages as they work towards a solution..

USE-CASE AS A PLANNING AID

One of the difficulties that developers face is planning delivery times.

The use case descriptions help the developer to:

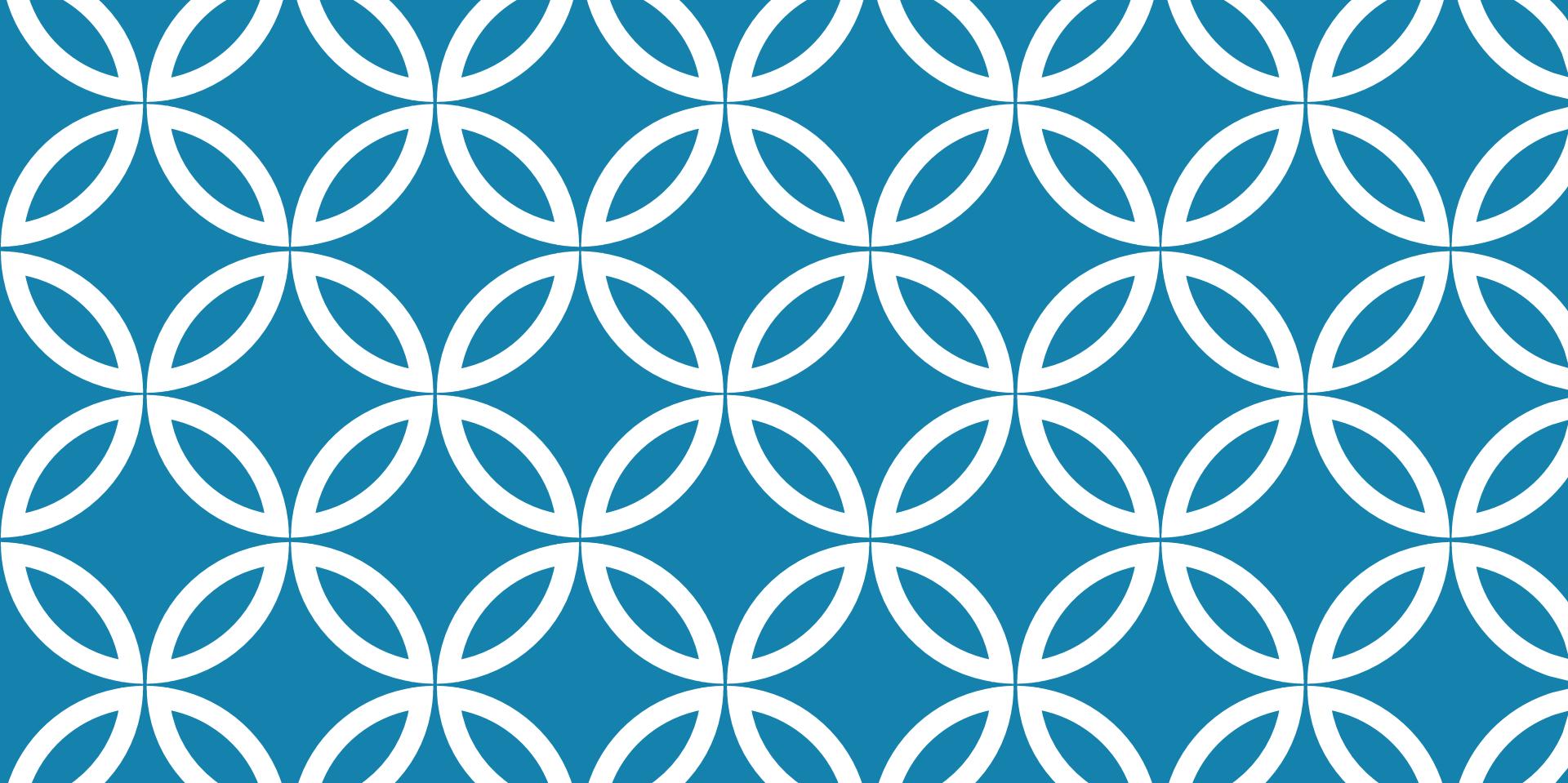
- Understand the complexity of each use case.
- Determine which actors interact with each use case and to what extent.
- Establish which use cases carry the most risk.
- Estimate how long each use case is likely to take to implement.

USE-CASE .. & ARCHITECTURE

- Use cases, as standalone chunks of system specification, dictate the sorts of functionality that need to be provided by the system and constitute an aid for identifying interfaces in an architecture.
- Use cases can also be grouped in terms of similar functionality, therefore influencing the architecture of the system.
- Scenarios can be used to check how an architecture meets non-functional requirements, in particular those that can be affected by the architecture, such as security and safety requirements.

USE-CASE .. & TESTING

- One way to validate a system is to use the walk-through technique, checking the functionality related to each use case in turn.
- The walkthrough technique can also be used to elicit system tests where each use case is required to deal with a number of scenarios – a process known as verification.



(CS251) SOFTWARE ENGINEERING I

Lecture 4
Introduction to UML, OO
Modelling, & Class
Diagrams

TOPICS COVERED

- Modelling
- Some UML Diagrams
- UML Tools
- Static Diagrams
- Class Model
- What is OO Software?
- Object Oriented “OO” Characteristics
- Identity
- Classes and Objects
- Classification
- Abstraction
- Encapsulation
- Inheritance
- How to use Inheritance?
- Polymorphism
- Generics
- Class Model
- Objects
- Values and Attributes
- Operations and Methods
- Summary of Basic Class Notation
- Sample Class Model

THE SOFTWARE PROCESS

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

Many different software processes but all involve:

- Specification – defining what the system should do;
- **Design** and implementation – defining the organization of the system and implementing the system;
- Validation – checking that it does what the customer wants;
- Evolution – changing the system in response to changing customer needs.

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

MODELLING

A model is an abstraction of a system.

Abstraction allows us to ignore unessential details

Why building models?

- To reduce complexity
- To test the system before building it
- To communicate with the customer
- To document and visualize your ideas

SOME UML DIAGRAMS

Functional diagrams

- Describe the functionality of the system from the user's point of view. It describes the interactions between the user and the system. It includes use case diagrams.

Static diagrams

- Describe the static structure of the system: Classes, Objects, attributes, associations.

Dynamic diagrams:

- **Interaction diagrams**
 - Describe the interaction between objects of the system
- **State diagrams**
 - Describe the temporal or behavioral aspect of an **individual** object
- **Activity diagrams**
 - Describe the dynamic behavior of a system, in particular the workflow.

UML TOOLS

Some UML tools can generate code once UML diagram is completed.

Some UML tools are sketching tools.

Rational Rose is one of the most popular software for UML creation (IBM).

Bouml is an open source s/w. It supports python, C++, java.

Visio is a sketching tool.

STATIC DIAGRAMS

- *Class diagrams* : show the classes and their relations
- *Object diagrams* : show objects and their relations
- *Package diagrams* : shows how the various classes are grouped into packages to simplify complex class diagrams.

CLASS MODEL

A **class model** captures the static structure of the system by characterizing

- the **classes and objects** in the system,
- the **relationships** among the objects and
- the **attributes** and **operations** for each class of objects

Class models are the **most important** OO models

In OO systems we build the system around objects not functionality

WHAT IS OO SOFTWARE?

Object-oriented software means that we organize software as a collection of discrete objects that incorporate both data structure and behavior.

The fundamental unit is the **Object**.

An object has **state (data)** and **behavior (operations)**.

In **Structured programming**, data and operations on the data were separated or loosely related.

OBJECT ORIENTED “OO” CHARACTERISTICS

- Identity
- Classification
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Generics
- Cohesion
- Coupling

IDENTITY

An object can be **concrete** like a **car**, a **file**, ...

An object can be **conceptual** like a **feeling**, a **plan**,...

Each object has its own identity even if two objects have exactly the same **state**.



Omar's car



Rana's car

IDENTITY

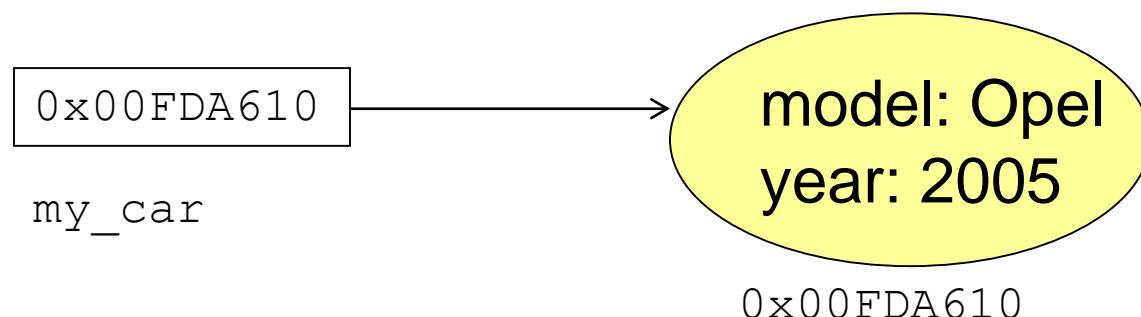
Object identity is the property by which each object can be identified and treated as a distinct software entity.

Each object has unique identity which distinguishes it from all its fellow objects. It is its memory address (or **handle**) that is referred to by one or more **object identifiers (variables)**.

IDENTITY

```
car my_car = new car("Opel", 2005);
```

The object handle is 0x00FDA610 is referenced by an object identifier (object variable) my_car



CLASSES AND OBJECTS

- Each object is an *instance* of a class
- Each object has a reference to its class (knows which class it belongs to)



abstract
into

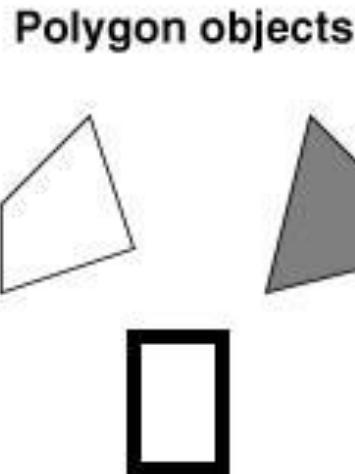
Bicycle class

Attributes

frame size
wheel size
number of gears
material

Operations

shift
move
repair



abstract
into

Polygon class

Attributes

vertices
border color
fill color

Operations

draw
erase
move

Figure 1.2 Objects and classes. Each class describes a possibly infinite set of individual objects.

CLASSIFICATION

Classification means that objects with the same data structure (**attributes**) and behavior (**operations**) belong to the same **class**.

A class is an **abstraction** that describes the properties important for an application

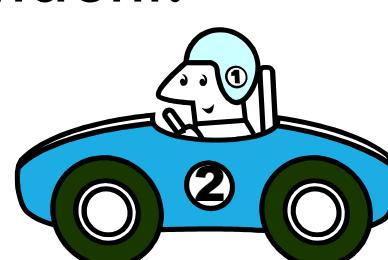
The choice of classes is arbitrary and application-dependent.



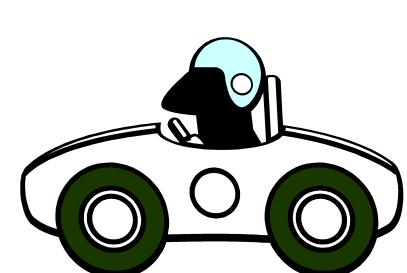
Mina's car



Ali's car



Samir's car



A Car

CLASSIFICATION



Objects in a class share a common ***semantic*** purpose in the system model.

Both car and cow have ***price*** and ***age***.

If both were modeled as pure ***financial assets***, they both can belong to the same class.

If the application needs to consider that:

- Cow eats and produces milk
- Car has speed, make, manufacturer, etc.

then model them using separate classes.

So the semantics depends on the application

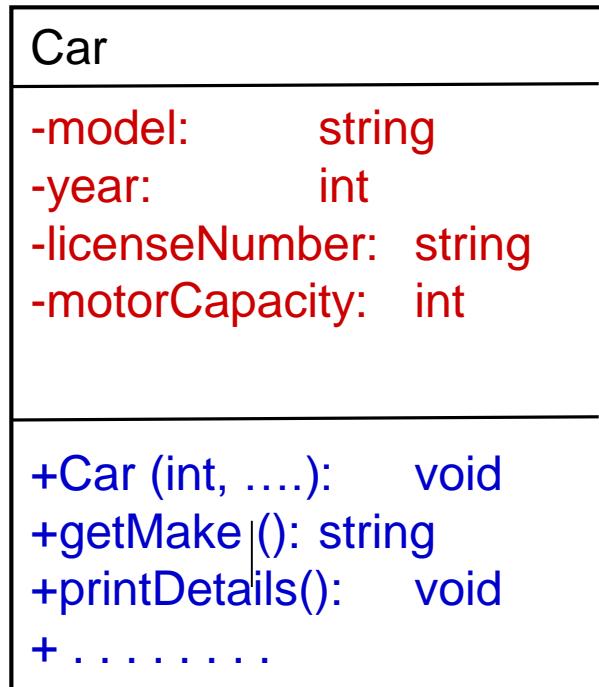
ABSTRACTION

Abstraction is the selective examination of certain aspects of a problem.

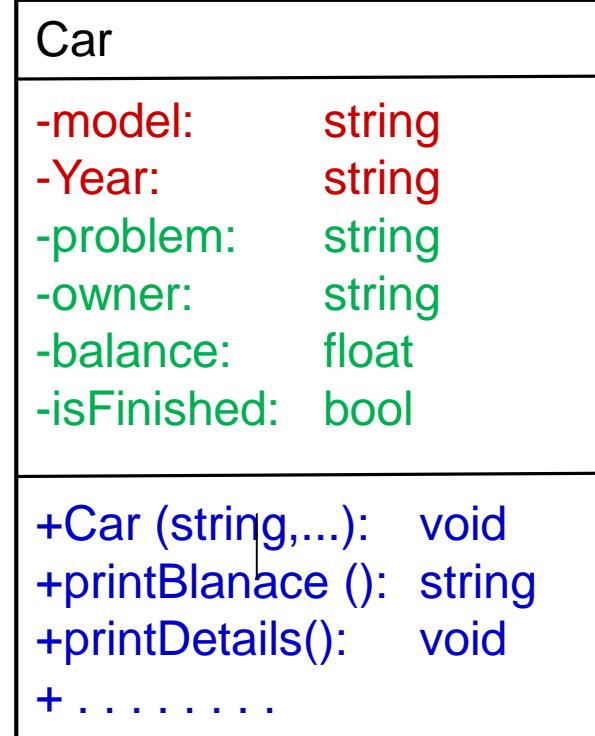
Abstraction aims to isolate the aspects that are important for some purpose and suppress the unimportant aspects.

The purpose of abstraction determines what is important and what is not.

ABSTRACTION



In Department of
Motor Vehicles



At the
mechanic

ENCAPSULATION

Encapsulation separates the external aspects of an object, that are accessible to other objects, from the internal implementation details that are hidden from other objects.

Encapsulation reduces **interdependency** between different parts of the program.

You can **change the implementation** of a class (to enhance performance, fix bugs, etc) **without affecting** the applications that use objects of this class.

ENCAPSULATION

Data hiding. information from within the object cannot be seen outside the object.

Implementation hiding. implementation details within the object cannot be seen from the outside.

ENCAPSULATION

List

- items: int []
- length: int

+ List (array):void
+ search (int): bool
+ getMax (): int
+ sort(): void

```
void sort () { // Bubble Sort
    int i, j;
    for (i = length - 1; i > 0; i--) {
        for (j = 0; j < i; j++) {
            if (items [j] > items [j + 1]) {
                int temp = items [j];
                items [j] = items [j + 1];
                items [j + 1] = temp;
            }
        }
    }
}
```

```
void sort () { // Quick Sort
    .....
}
```

INHERITANCE

Inheritance is the sharing of **features** (attributes and operations) among classes based on a hierarchical relationship.

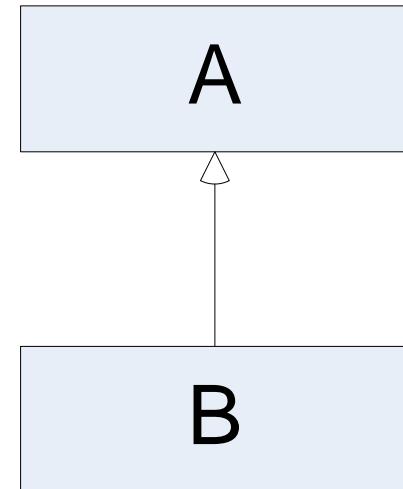
A **superclass** (also **parent** or **base**) has general features that **subclasses** (**child** or **derived**) inherit, and may refine some of them.

Inheritance is one of the strongest features of OO technology.

INHERITANCE

Inheritance is the facility by which objects of a class (say B) may use the methods and variables that are defined only to objects of another class (say A), as if these methods and variables have been defined in class B

Inheritance is represented as shown in **UML** notation.

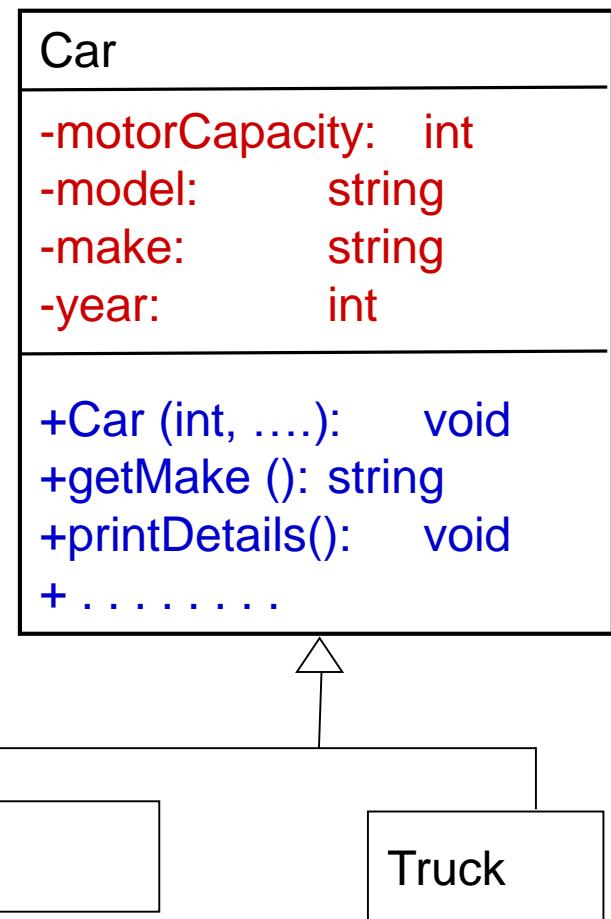


HOW TO USE INHERITANCE?

Inheritance helps building software incrementally:

First; build classes to cope with the most straightforward (or general) case,

Second; build the special cases that inherit from the general base class. These new classes will have the same features of the base class plus their own.



POLYMORPHISM

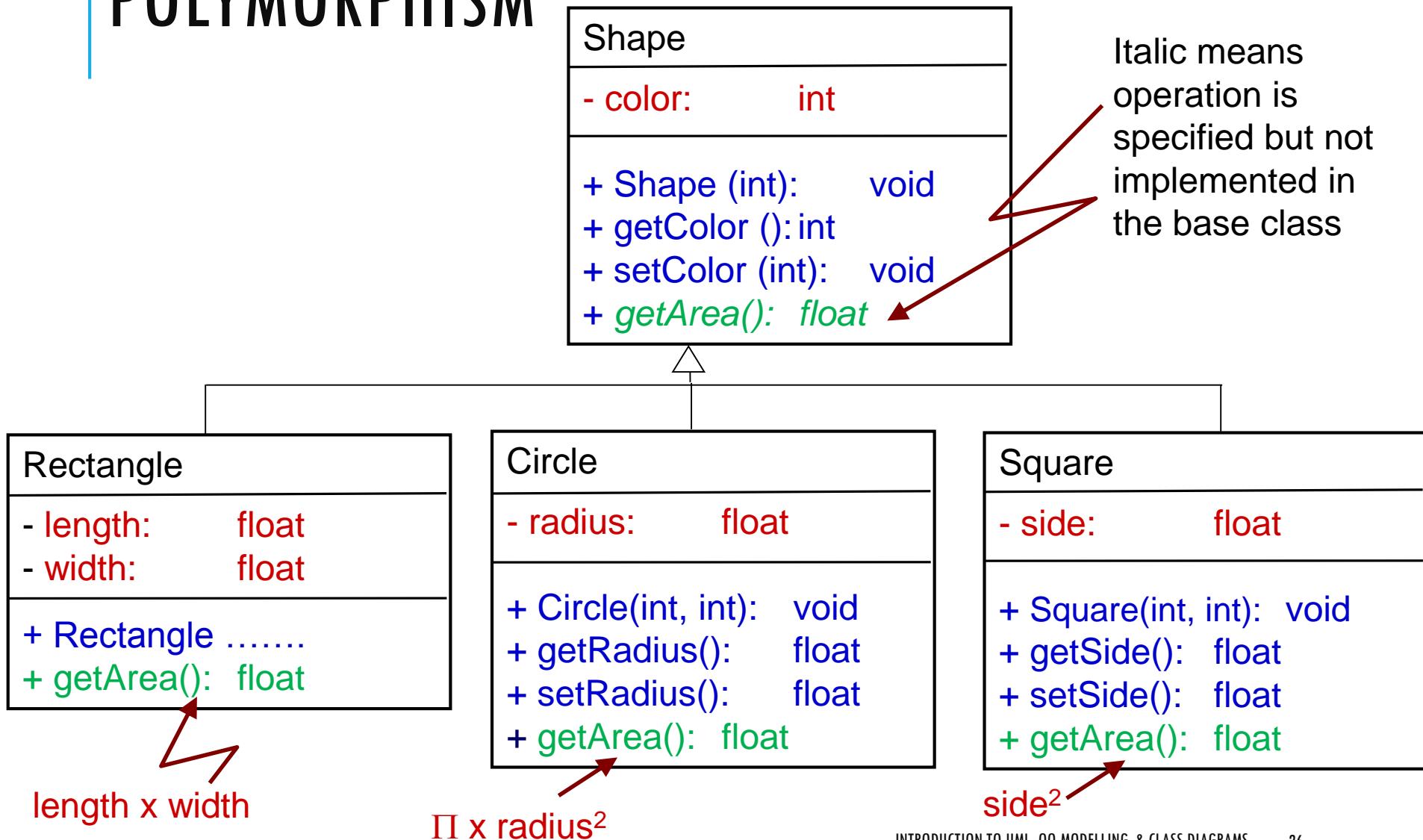
Polymorphism means that the same **operation** may behave differently for different classes.

An **operation** is a procedure or a transformation that the object performs or is subject to.

An implementation of an operation by a specific class is called a **method**.

Because an OO operation is polymorphic, it may have more than one method for implementing it, each for a different class.

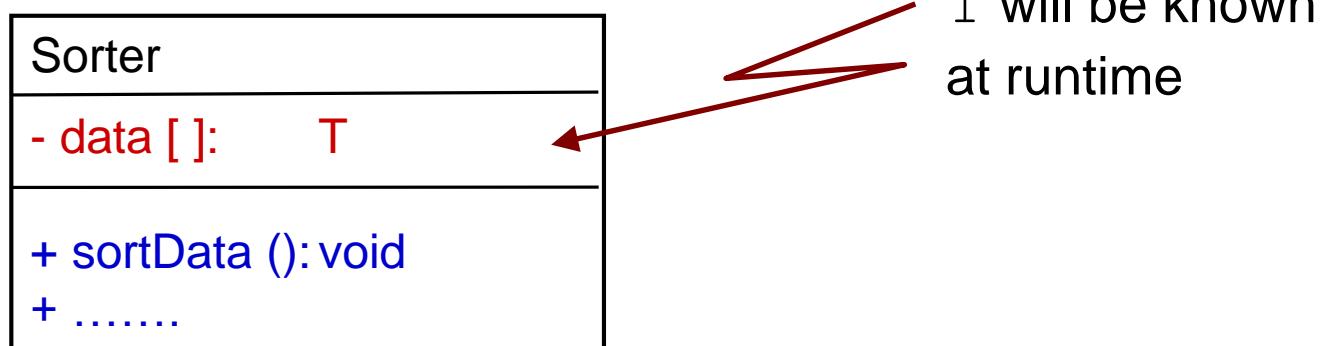
POLYMORPHISM



GENERICCS

Generic Class ; the data types of one or more of its attributes are supplied at run-time (at the time that an object of the class is instantiated).

This means that the class is **parameterized**, i.e., the class gets a parameter which is the name of another type.



CLASS MODEL

A **class model** captures the static structure of the system by characterizing

- the **classes and objects** in the system,
- the **relationships** among the objects and
- the **attributes** and **operations** for each class of objects

Class models are the **most important** OO models

In OO systems we build the system around objects not functionality

OBJECTS

Objects often appear as **proper nouns** in the problem description or discussion with the customer.

Some object correspond to **real world entities** (BUE, MIT, Omar's car)

Some objects correspond to **conceptual entities** (the formula for solving an equation, binary tree, etc.)

The choice of objects depends on the analyst's judgment and the problem in hand. There can be **more than one correct representation**.

CLASS

An object is an **instance of** a class

A class describes a group of objects with the same

- Properties (attributes)
- Behavior (operations)
- Kinds of relationships

Person, Company and Window are all classes

Classes often appear as **common nouns** and **noun phrases** in problem description and discussion with customers or users

CLASS MODEL

Provides a graphical notation for modeling classes and their relationships, thereby describing possible objects

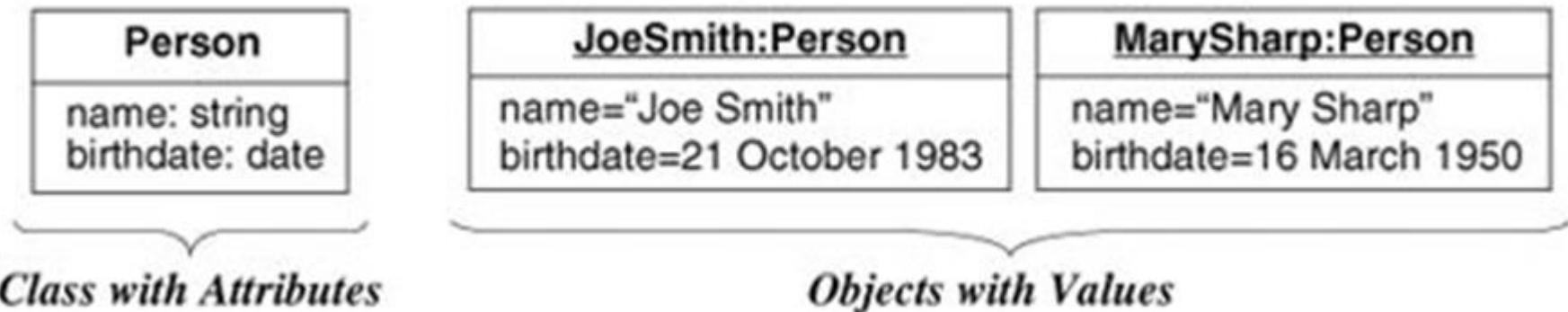
Class diagram is useful for:

- *Designing and Implementing the programs*



Figure 3.1 A class and objects. Objects and classes are the focus of class modeling.

VALUES AND ATTRIBUTES



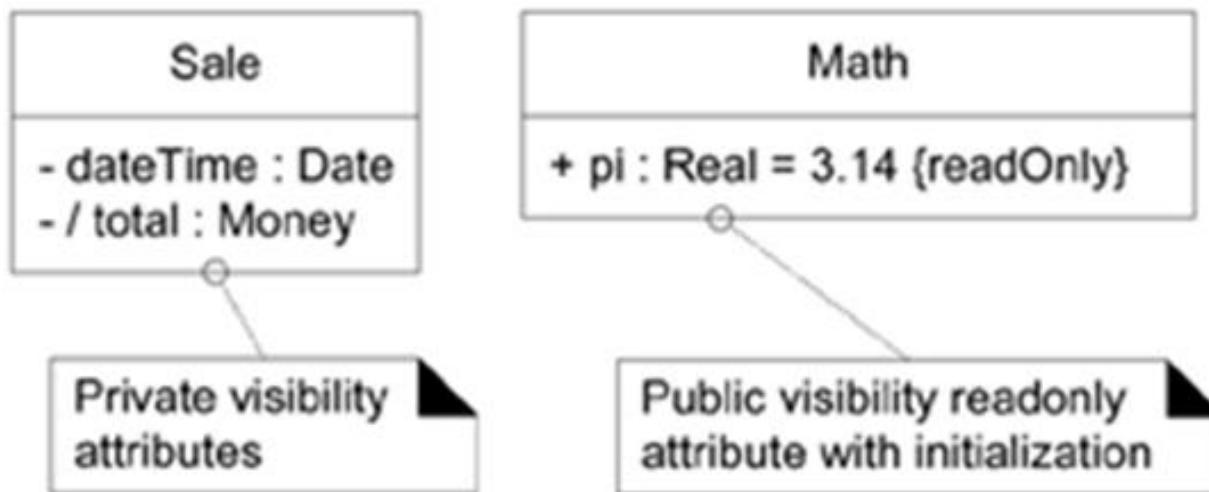
An **attribute** is a named property of class that describes a value held by each object of that class.

Attribute name is unique per class.

Several classes may have the same attribute name.

A **value** is a piece of data assigned to an attribute.

MORE ON ATTRIBUTES



OPERATIONS AND METHODS

An **operation** is a function or procedure that may be applied to or by objects of a class.

A **method** is the implementation of an operation for a class.

An operation is **polymorphic** if it takes different forms in different classes.

All objects of the same class have the same operations.

Financial Asset

-type: int
-age: float
-currentValue: float
- . . .

+getCurrentValue(): int
+printDetails(): void
+

SUMMARY OF CLASS NOTATION

The attribute and operation compartments are optional.

You may show them or not depending on the level of abstraction you want.

A missing attribute compartments means that the attributes are not specified yet.

But empty compartment means that the attributes are specified but there are none.

SUMMARY OF BASIC CLASS NOTATION

ClassName

attributeName1 : dataType1 = defaultValue1
attributeName2 : dataType2 = defaultValue2
• • •

operationName1 (argumentList1) : resultType1
operationName2 (argumentList2) : resultType2
• • •

Figure 3.5 Summary of modeling notation for classes. A box represents a class and may have as many as three compartments.

A SAMPLE CLASS MODEL

Model the classes in a system that represents flights. Each city has at least an airport. Airlines operate flights from and to various airports. A flight has a list of passengers, each with a designated seat. Also a flight uses one of the planes owned by the operating airline. Finally a flight is run by a pilot and a co-pilot.

A SAMPLE CLASS MODEL

Model the classes in a system that represents **flights**. Each **city** has at least an **airport**. **Airlines** operate flights from and to various airports. A flight has a **list** of **passengers**, each with a designated **seat**. Also a flight uses one of the **planes** owned by the operating airline. Finally a flight is run by a **pilot** and a **co-pilot**.

A SAMPLE CLASS MODEL

- *Flights*
- *List of Passengers*
- *Pilot* and a *Co-Pilot*
- *City*
- *Seat*
- *Airlines*
- *Planes*

City

Airline

Pilot

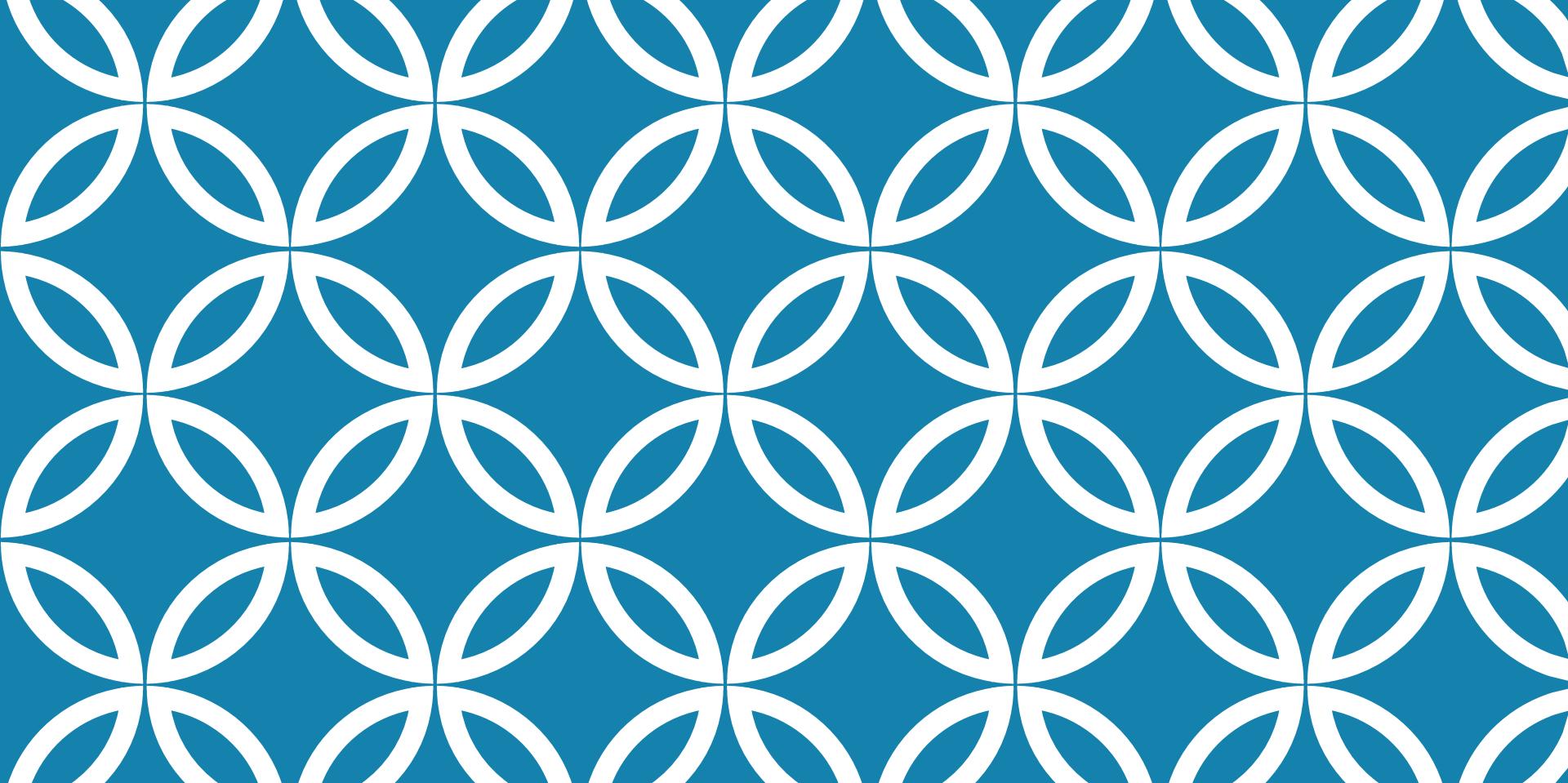
Plane

Passenger

Airport

Flight

Seat



(CS251) SOFTWARE ENGINEERING I

Lecture 5
OO Modelling, UML Static
Diagrams (Class, Object, &
Package Diagrams)

TOPICS COVERED

- Continuing Class Diagrams
- Associations
- Multiplicity
- Links and Associations
- Roles
- Self-Associations
- Association classes vs. Ordinary classes
- Qualified Association
- A sample class model
- Aggregation & Composition Relation
- Generalization, Specialization & Inheritance
- Overriding Features
- Abstract Class
- Object Diagrams
 - Elements & An Example
- Class Modelling Tips
- Divide and Conquer: Modularization
- Cohesion
 - Types of Cohesion
 - Coincidental Cohesion
 - Functional Cohesion
- Coupling
 - Examples
 - Consequences of Coupling
- Package Diagrams

ASSOCIATIONS

An association is a relationship between classes that indicate some meaningful and interesting relationship.

It's represented by a line with a name.

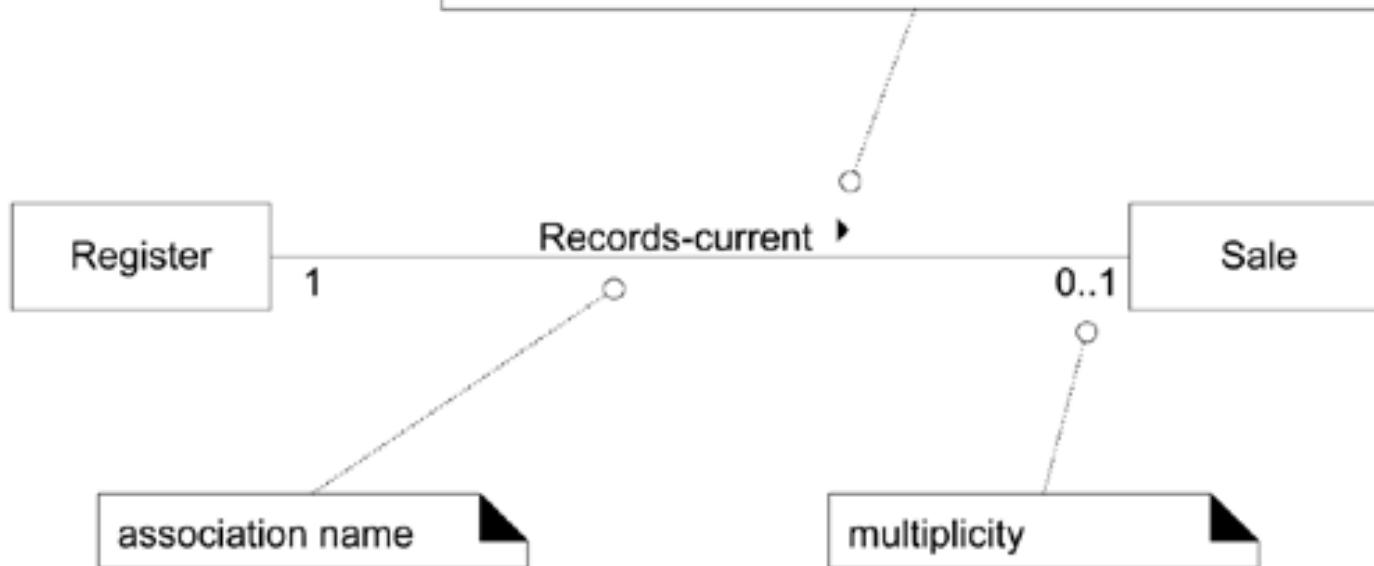
Properly naming associations is important to enhance understanding: **use verb phrase.**



ASSOCIATIONS

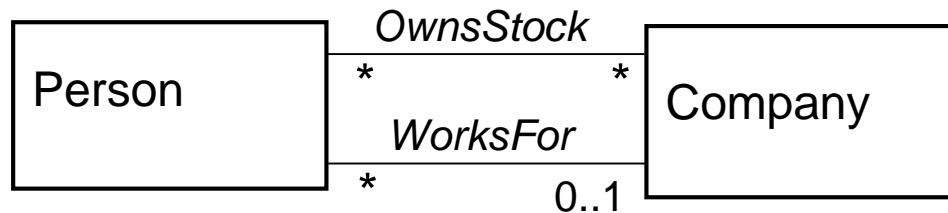
An optional "reading direction arrow" indicates the direction to **read** the association name; *it does not indicate direction of visibility or navigation*

- "reading direction arrow"
- it has no meaning except to indicate direction of reading the association label
- often excluded



ASSOCIATIONS

There may be more than one associations between classes (this is not uncommon).



ASSOCIATIONS

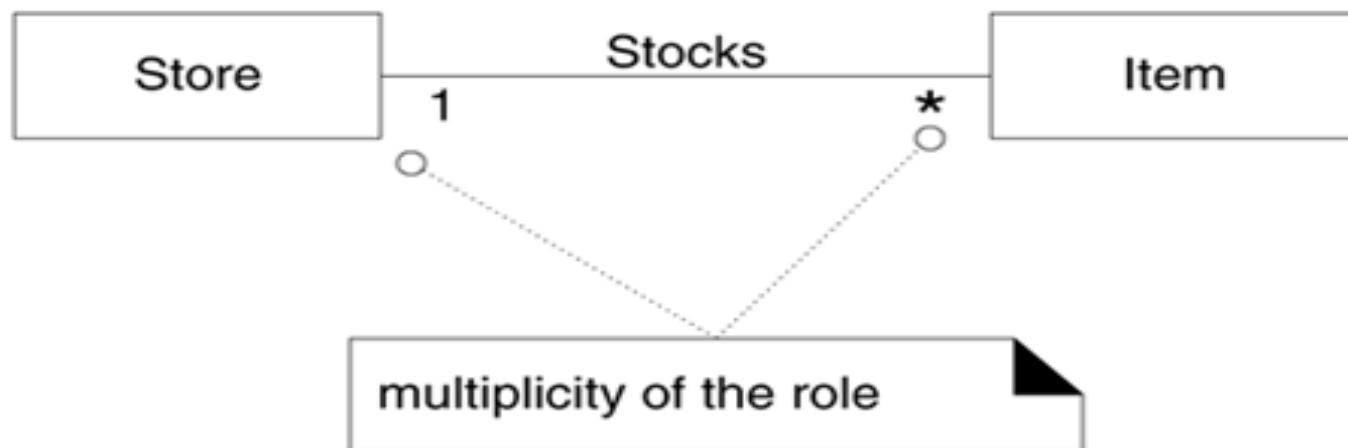
Associations are usually implemented by a **reference** from an object to another.

Associations are inherently bidirectional. They can be traversed in either direction. A person *WorksFor* a company and a company *Employs* a person.

Associations could be unidirectional.

MULTIPLICITY

Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.



MULTIPLICITY

Multiplicity exposes hidden assumptions in the model

For example, if a person **WorksFor** a company, can he work for more than one company? In other words, is it **one-to-one** or **one-to-many** association?

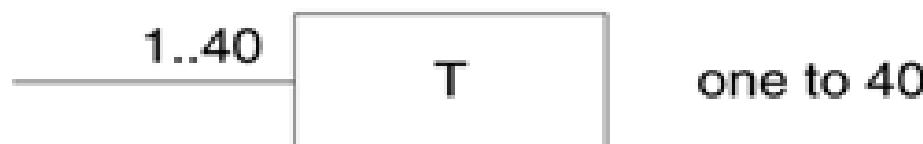
MULTIPLICITY VALUES



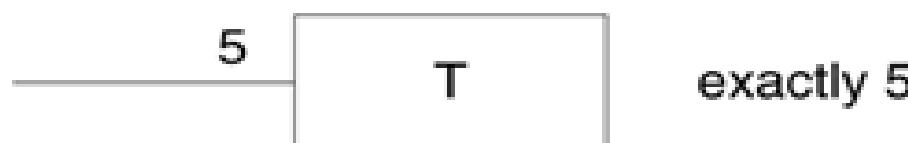
zero or more;
"many"



one or more



one to 40



exactly 5



exactly 3, 5, or 8

LINKS AND ASSOCIATIONS

A **link** is a physical or conceptual connection among objects

- For example John works for GE company.

An **association** is a relationship between classes and represents group of links.

- For example, a person `WorksFor` a company.

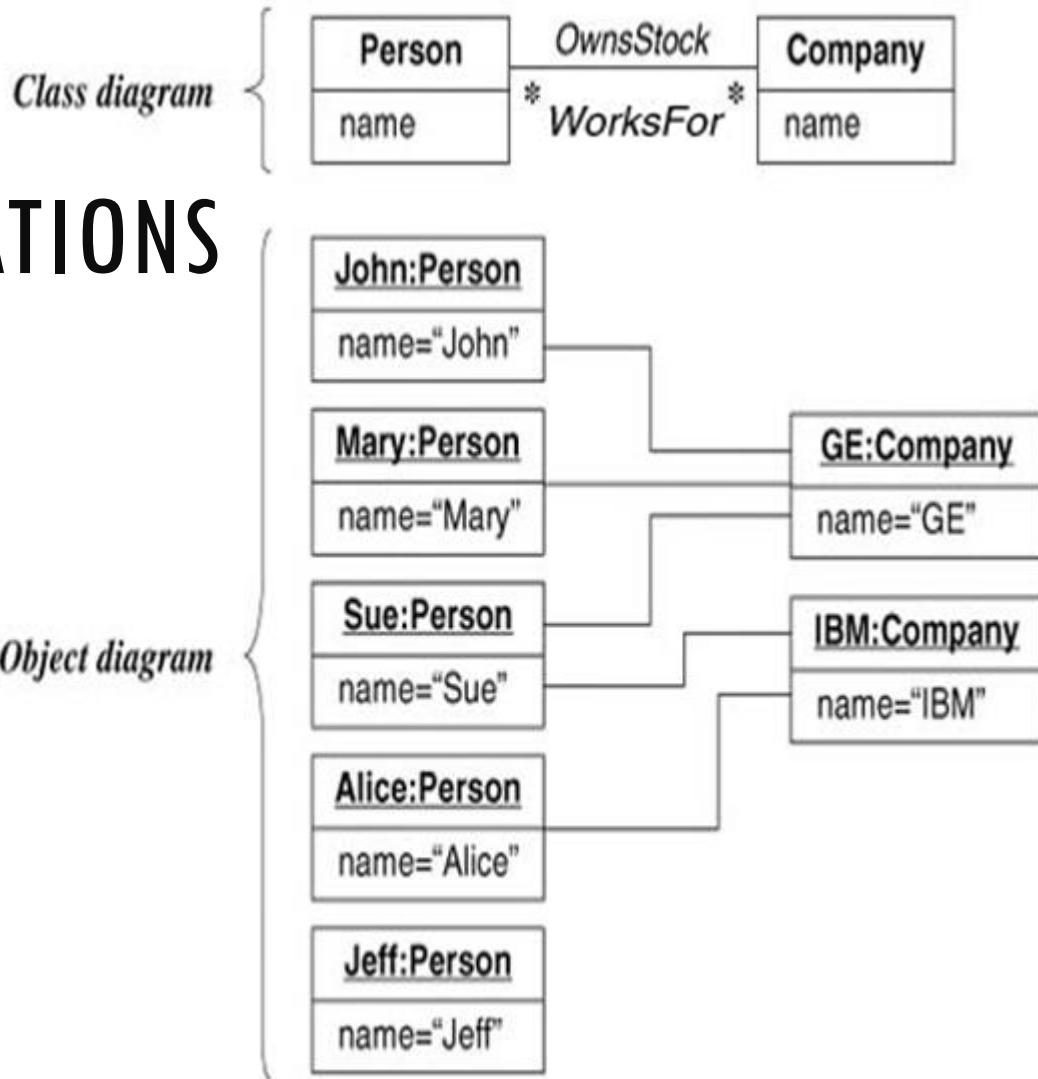


Figure 3.7 Many-to-many association. An association describes a set of potential links in the same way that a class describes a set of potential objects.

LINKS AND ASSOCIATIONS

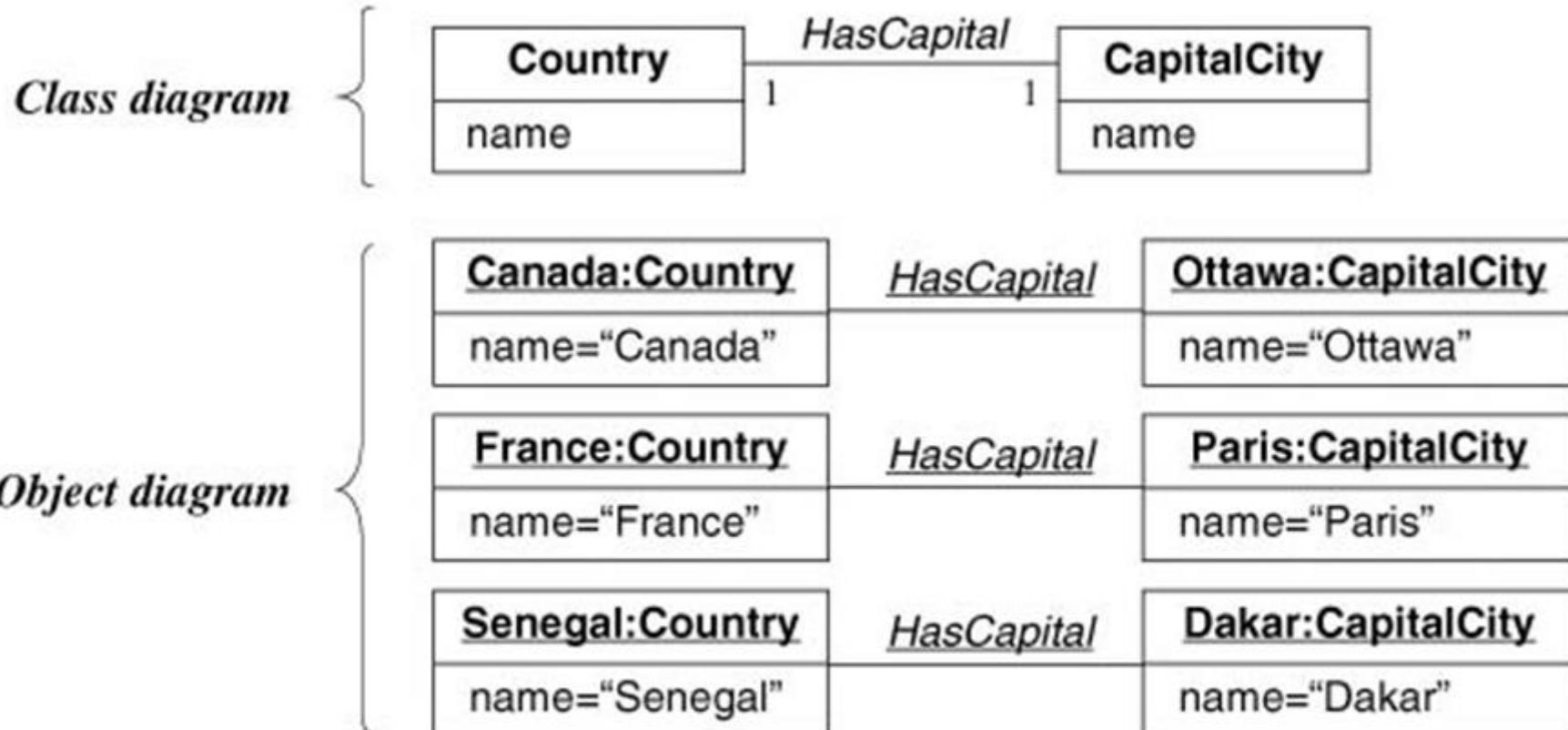


Figure 3.8 One-to-one association. Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

LINKS AND ASSOCIATIONS

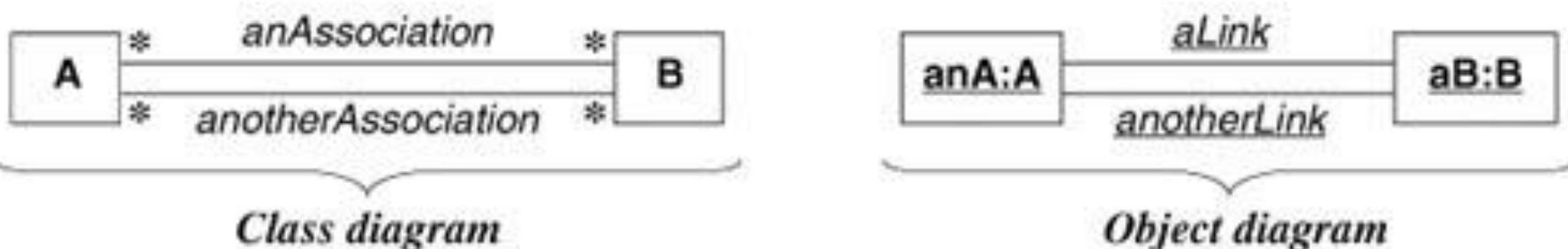


Figure 3.11 Association vs. link. You can use multiple associations to model multiple links between the same objects.

Object -Oriented Modeling and Design with UML, Second Edition by Michael Blaha and James Rumbaugh. ISBN 0-13-1-015920-4. © 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

ROLES

Each **association End** can be labeled by a **role**.

This makes understanding associations easier.

They are especially important for ***Self-associations*** between objects of the same class.

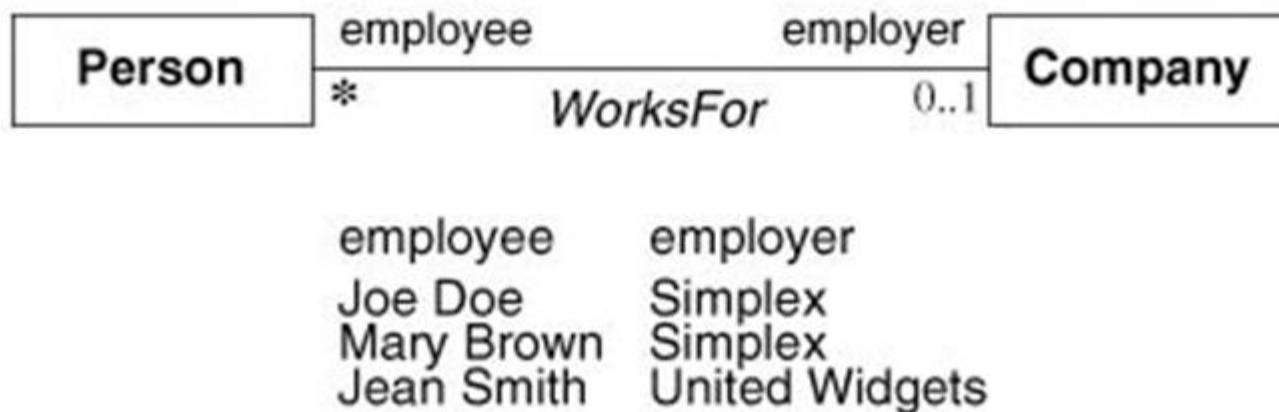


Figure 3.12 Association end names. Each end of an association can have a name.

SELF ASSOCIATIONS

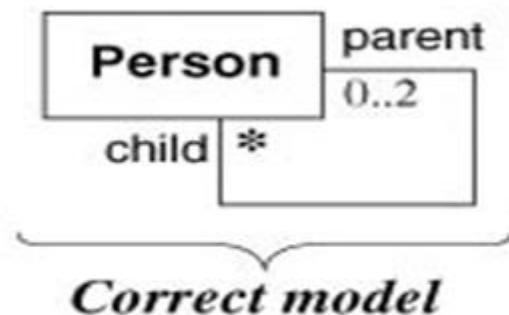
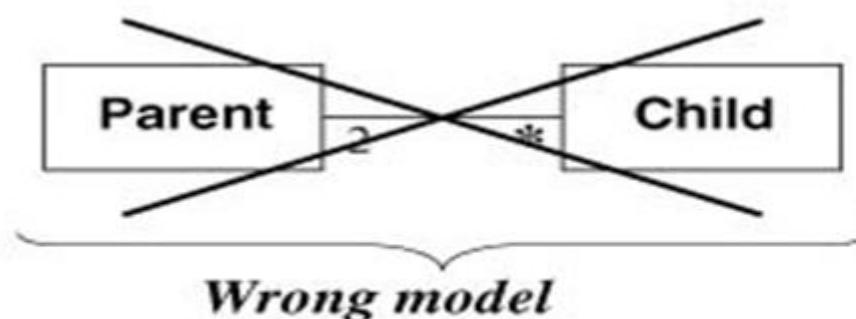
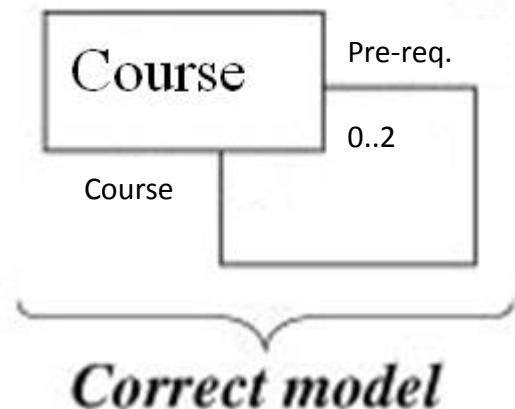
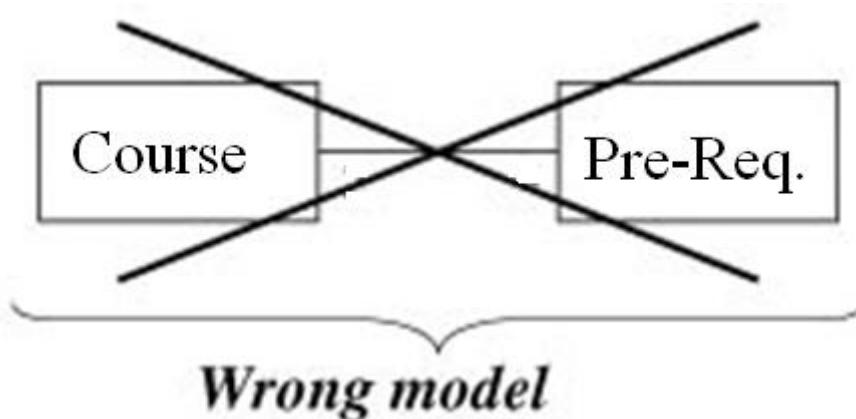


Figure 3.14 Association end names. Use association end names to model multiple references to the same class.

ASSOCIATION CLASSES

An **Association class** is an association that is also a class (has attributes and operations)

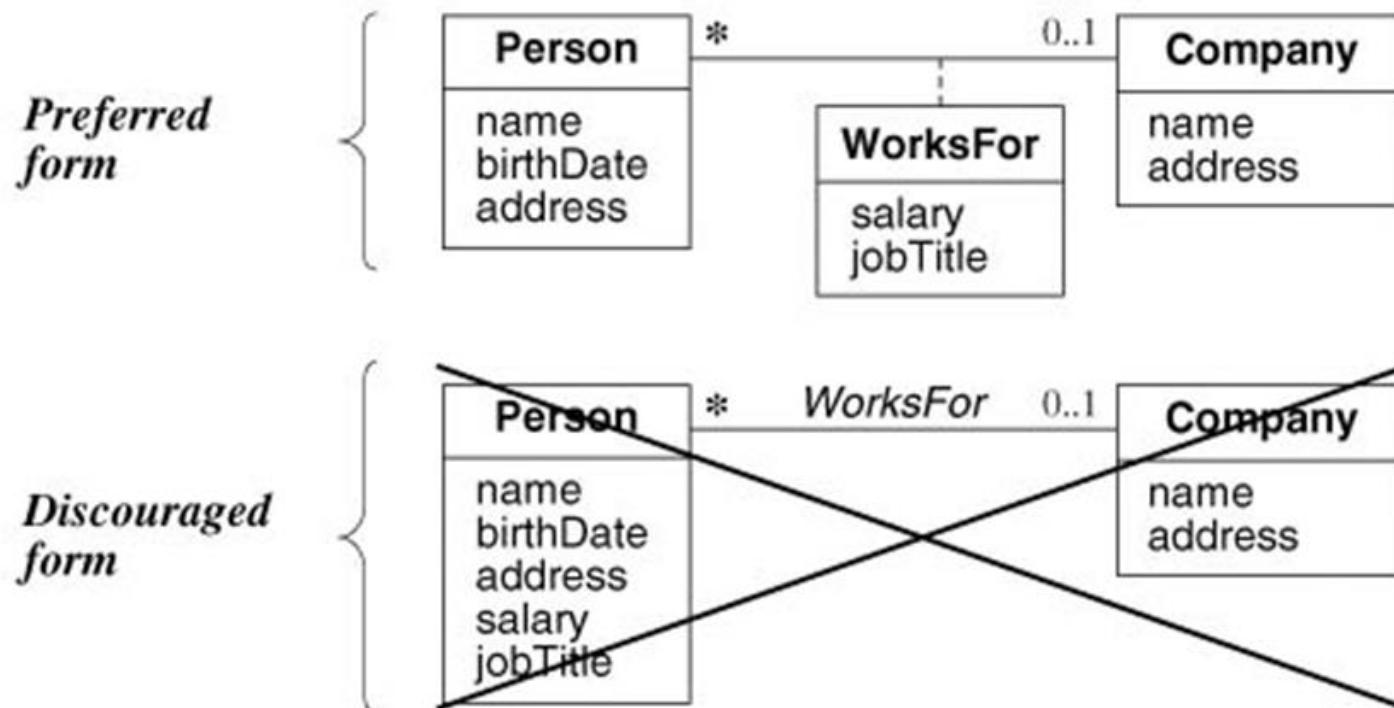
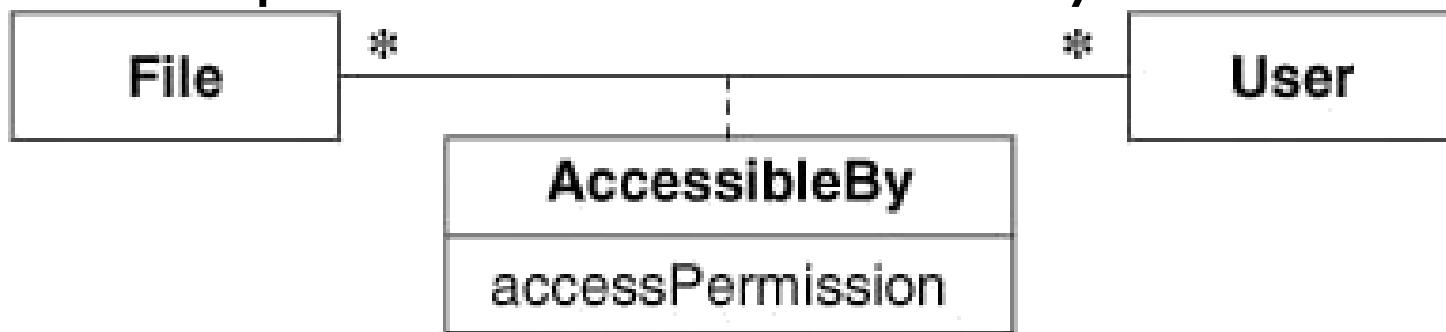


Figure 3.19 Proper use of association classes. Do not fold attributes of an association into a class.

ASSOCIATION CLASSES

Many-to-many associations provide compelling rationale for association classes.

accessPermission is a joint property of File and User and cannot be placed in one of them only.



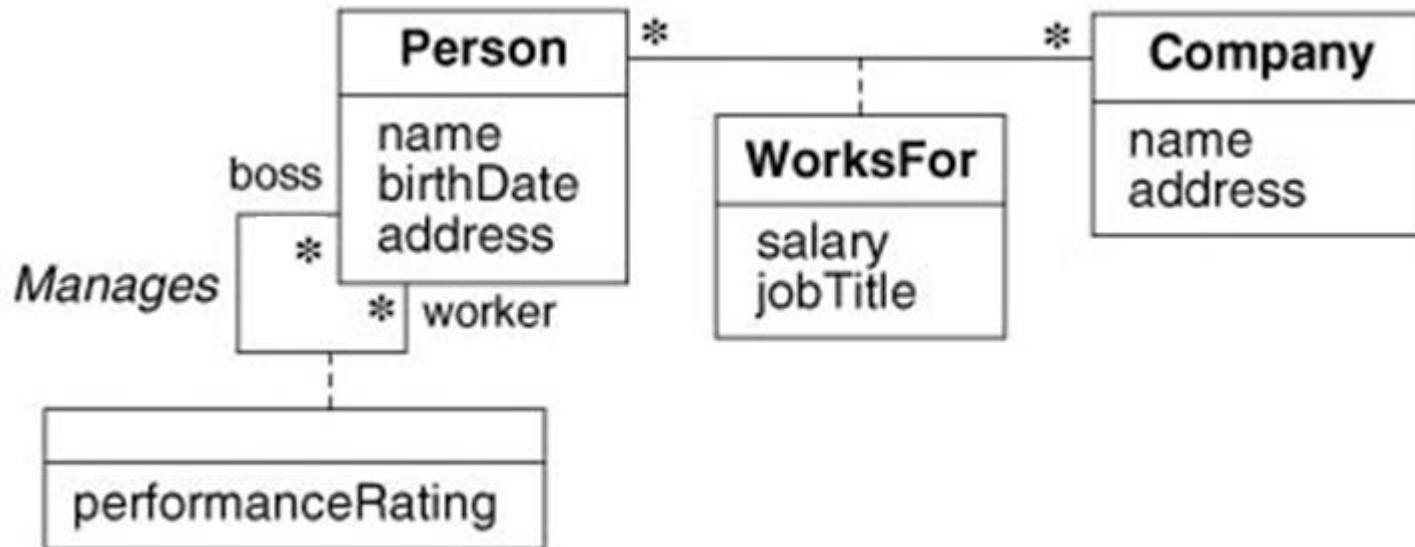
/etc/termcap	read	John Doe
/etc/termcap	read-write	Mary Brown
/usr/doe/.login	read-write	John Doe

Figure 3.17 An association class. The links of an association can have attributes.

EXERCISES

Sketch a class diagram for the following:

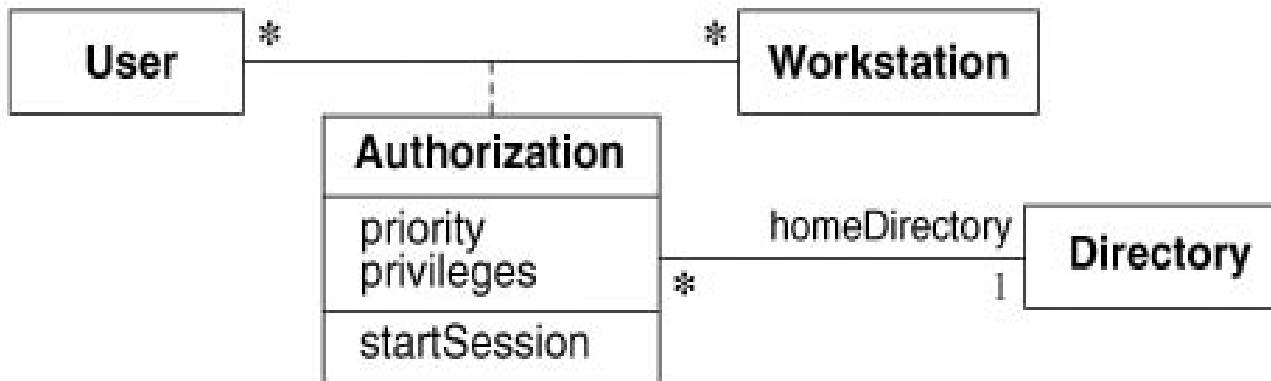
Each person works for a company receives a salary and has a job title. The person can work for more than one company. The boss evaluates the performance of each worker.



EXERCISES

Sketch a class diagram for the following:

A user may be authorized on many workstations. Each authorization has a priority and access privileges. A user has a home directory for each authorized workstation, but several workstations and users can share the same home directory.



ASSOCIATION CLASSES VS. ORDINARY CLASSES

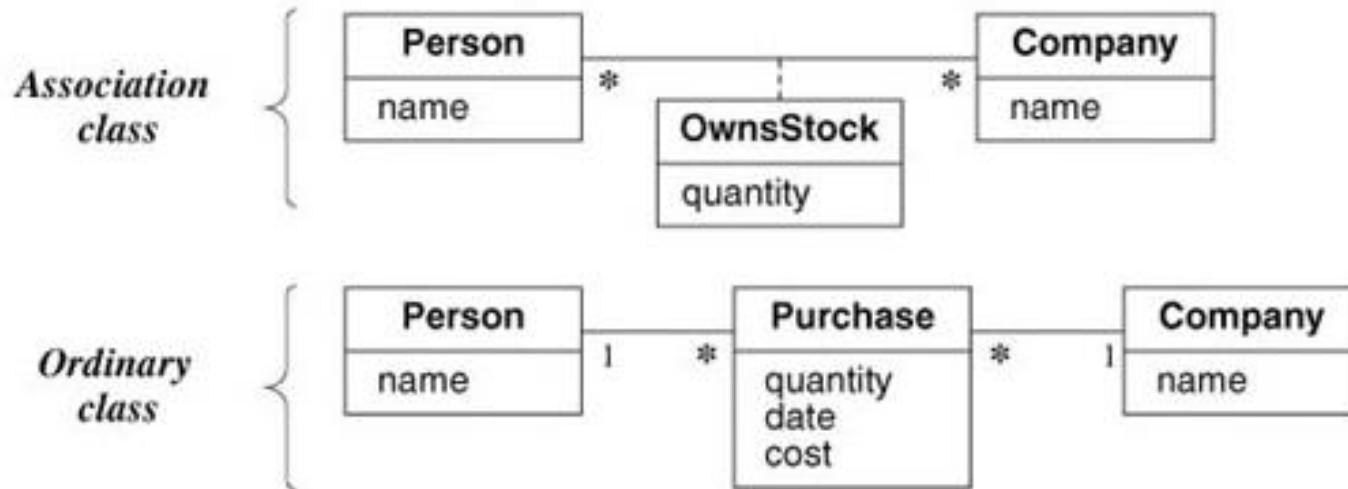


Figure 3.21 Association class vs. ordinary class. An association class is much different than an ordinary class.

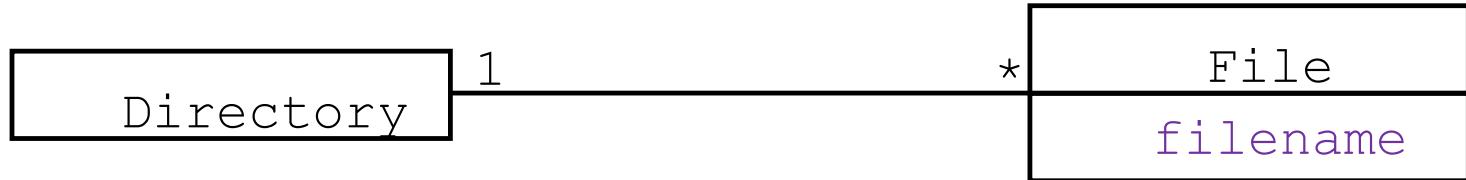
Association class **OwnsStock** has only one occurrence for each pair of person and company (i.e. only one link exists between pair of person and company objects).

In contrast there can be any number of **purchase** objects for each pair of person and company objects.

QUALIFIED ASSOCIATION

A **qualified association** is an association in which an attribute called the **qualifier** is used to reduce a **many** relation to **one** relation on the other end.

Not qualified



qualified



QUALIFIED ASSOCIATION

A bank has many accounts [0..*]

But a bank + account number gives only one account

Account Number is attribute of account class.

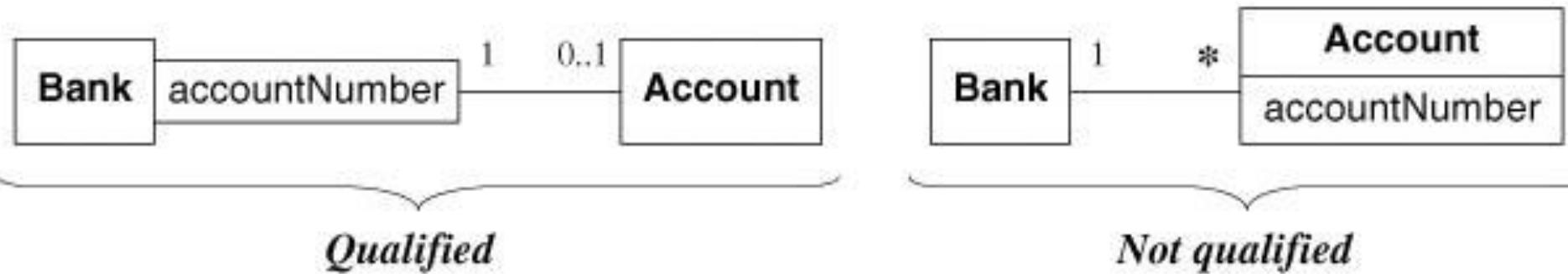


Figure 3.22 Qualified association. Qualification increases the precision of a model.

A SAMPLE CLASS MODEL

Model the classes in a system that represents flights. Each city has at least an airport. Airlines operate flights from and to various airports. A flight has a list of passengers, each with a designated seat. Also a flight uses one of the planes owned by the operating airline. Finally a flight is run by a pilot and a co-pilot.

The order of modeling is:

- Define classes
- Define associations
- Define multiplicity

A SAMPLE CLASS MODEL

Model the classes in a system that represents **flights**. Each **city** has at least an **airport**. **Airlines** operate flights from and to various airports. A flight has a **list** of **passengers**, each with a designated **seat**. Also a flight uses one of the **planes** owned by the operating airline. Finally a flight is run by a **pilot** and a **co-pilot**.

A SAMPLE CLASS MODEL

- *Flights*
- *List of passengers*
- *Pilot* and a *Co-Pilot.*
- *City*
- *Seat*
- *Airlines*
- *Planes*

City

Airline

Pilot

Plane

Airport

Flight

Seat

Passenger

A SAMPLE CLASS MODEL

Model the classes in a system that represents flights. Each city **has** at least an airport. Airlines **operate** flights **from** and **to** various airports. A flight **has** a list of passengers, each with a designated seat. Also a flight **uses** one of the planes **owned** by the operating airline. Finally a flight is **run** by a pilot and a co-pilot.

A SAMPLE CLASS MODEL

City **has** airport(s)

Flight **to** airport

Flight **uses** airplane

Flight **run by** pilot & co-pilot

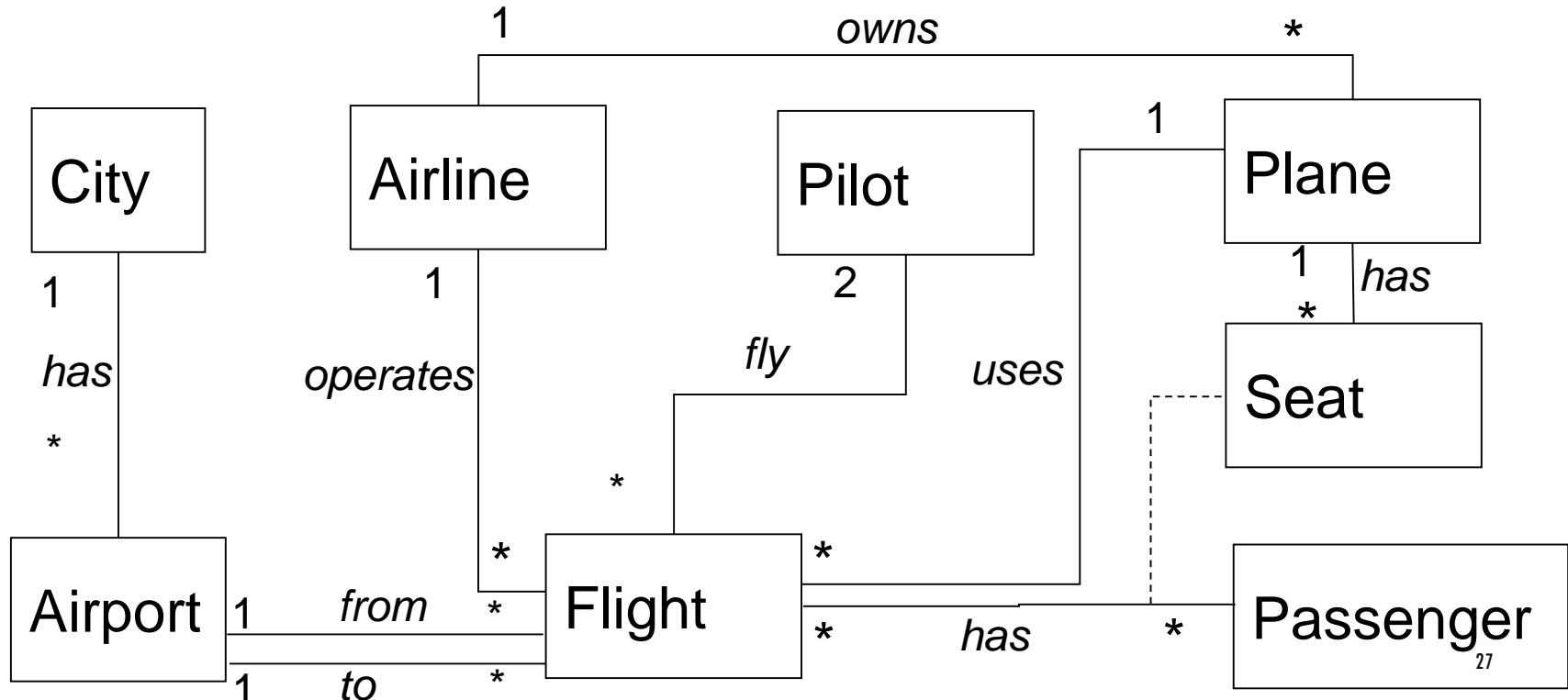
Airline **operates** flights

Flight **has** passengers

Airplane **owned by** airline

Flight **from** airport

Passenger **has** seat



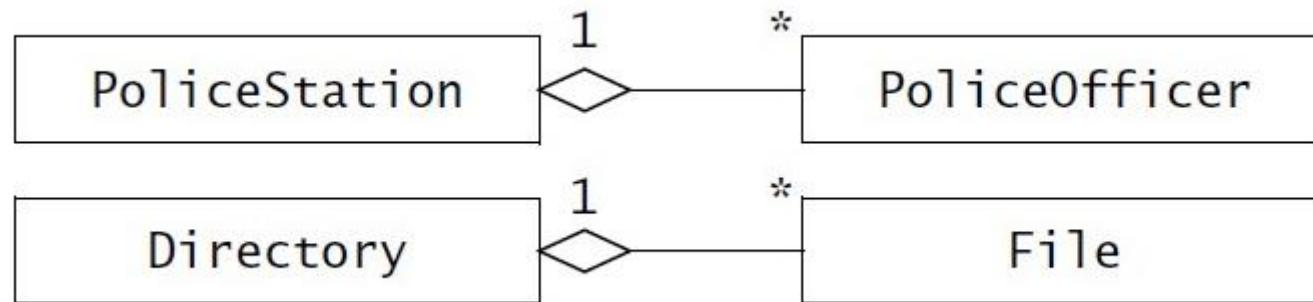
A SAMPLE CLASS MODEL

Add some suitable attributes and operations to the classes of flight system.

FURTHER CLASS CONCEPTS

AGGREGATION RELATION

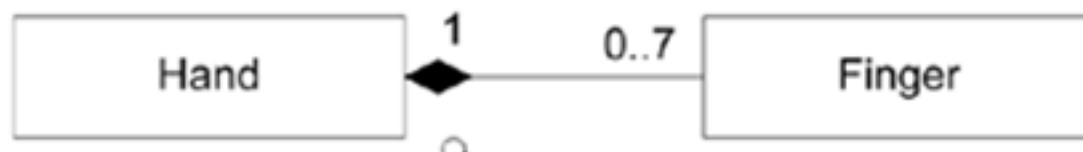
Aggregation is a special kind of association where an aggregate object **contains** constituent parts. It's **has-a** relationship



FURTHER CLASS CONCEPTS

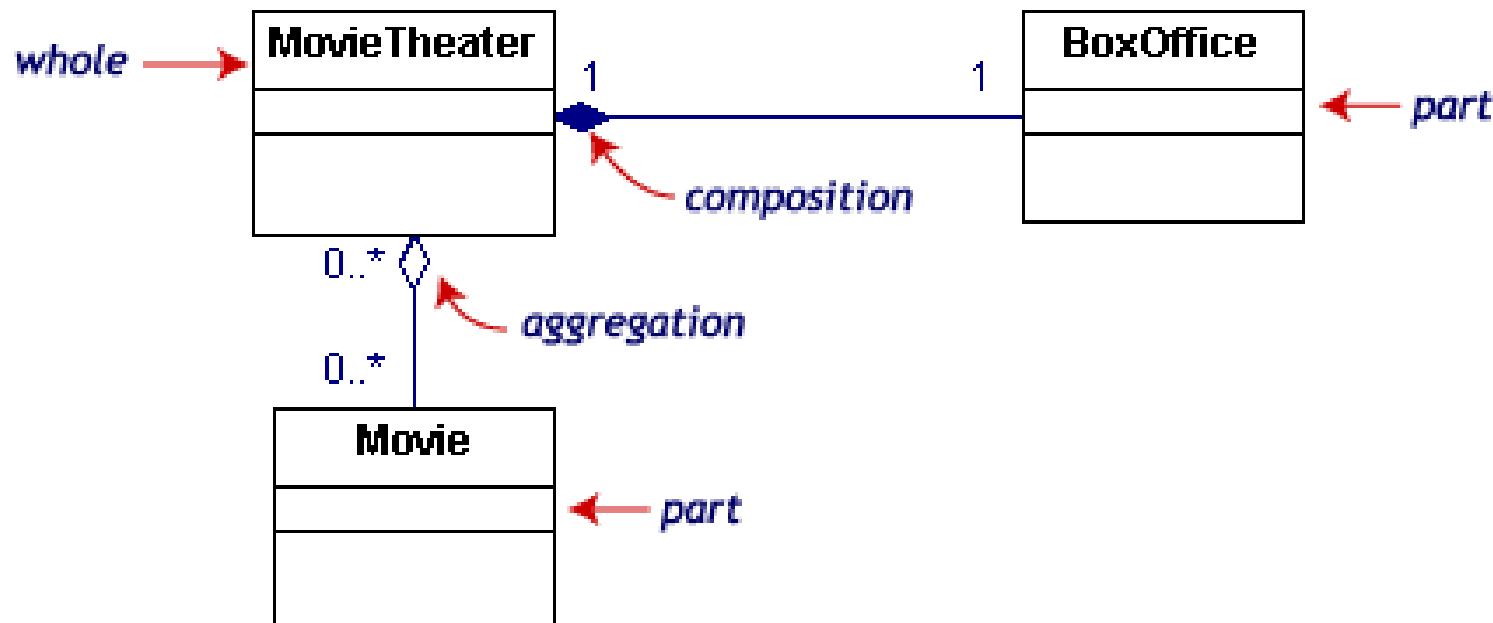
COMPOSITION RELATION

Composition is a special kind of association in which a constituent part belongs at most to one assembly and exists only if the assembly exists.



FURTHER CLASS CONCEPTS

AGGREGATION & COMPOSITION RELATION



FURTHER CLASS CONCEPTS

GENERALIZATION, SPECIALIZATION AND INHERITANCE

Generalization is the relationship between class (the superclass) and one or more variations of the class (the subclass).

The superclass holds common attributes, operations and associations.

The subclasses add specific attributes, operations and associations.

Each subclass is said to **inherit** the features of the its superclass.

Inheritance is called “**is-a**” relationship.

GENERALIZATION, SPECIALIZATION AND INHERITANCE

The terms **generalization**, **specialization** and **inheritance** refer to aspects of the same idea, they are used in place of one another .

Generalization refers to the super-class generalizing its subclasses

Specialization refers to the fact that the subclass specializes or refines the super-class,

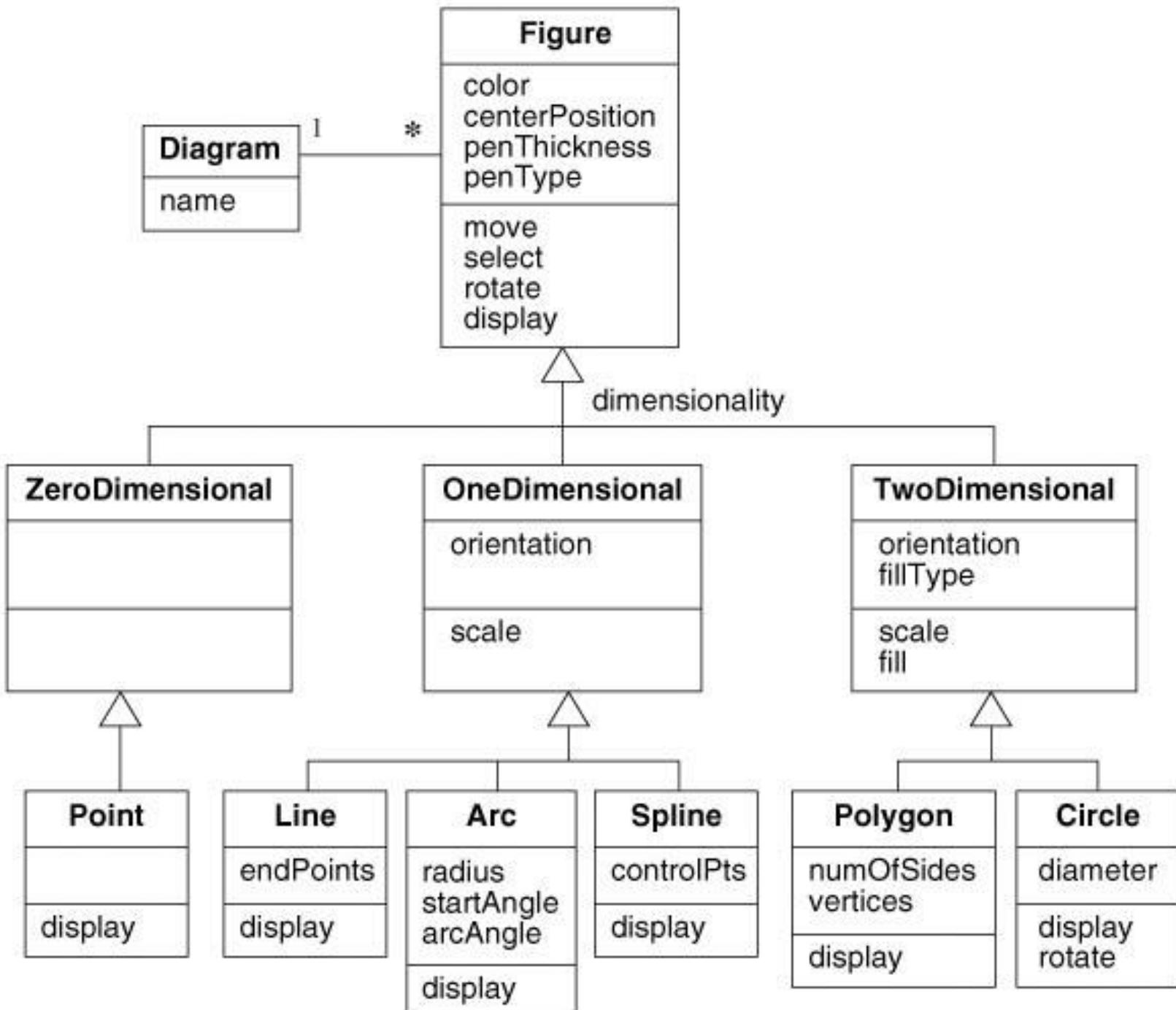
Inheritance refers to the mechanism for implementing generalization / specialization.

GENERALIZATION, SPECIALIZATION AND INHERITANCE

Generalization is **transitive** across an arbitrary number of levels.

An **ancestor** is parent or grandparent class of a class.

A **descendant** is a child or grandchild of a class.



USE OF GENERALIZATION

Generalization serves three purposes.

1. *Supporting polymorphism.*
2. *Structuring the description of objects* and their relation to each other based on their similarity and differences.
3. *Reusing code.* You inherit code from super-classes or from a class library automatically with inheritance. Reuse is more productive than repeatedly writing code from scratch. When reusing code, you can adjust the code if necessary to get the exact desired behavior.

OVERRIDING FEATURES

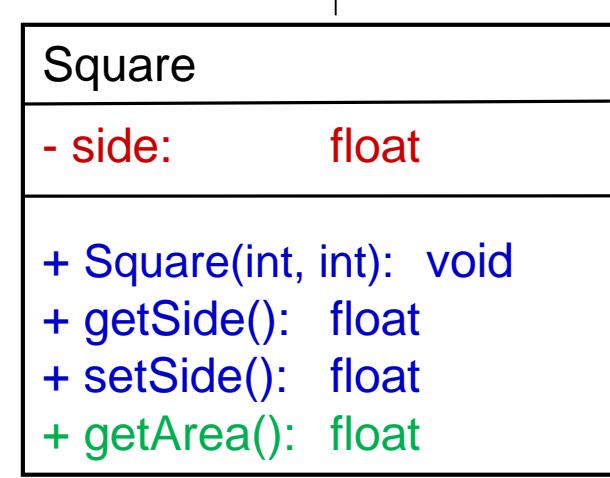
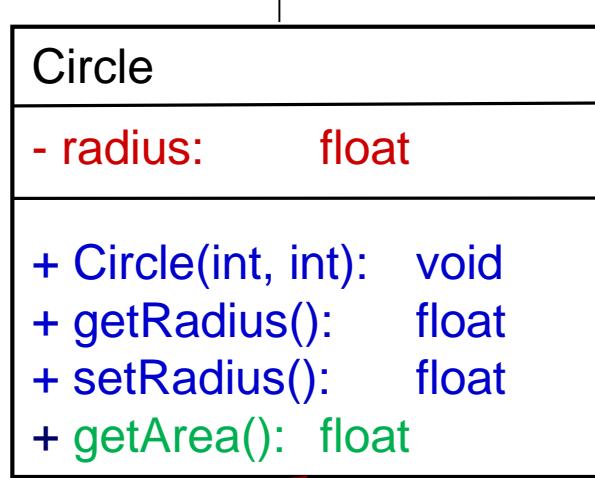
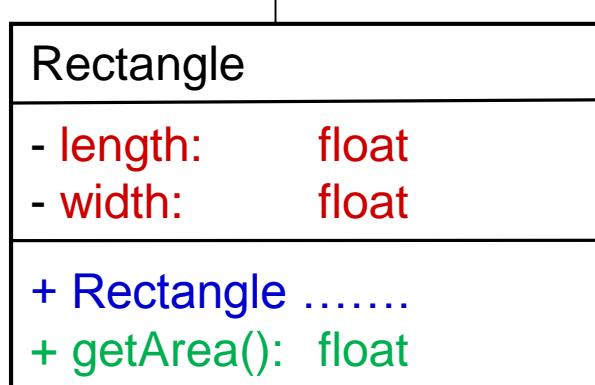
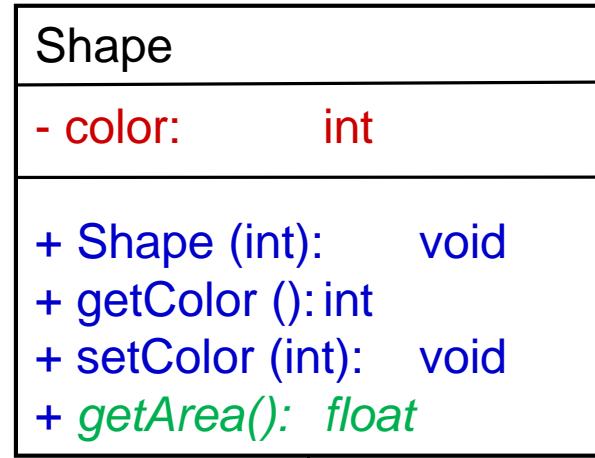
A subclass may **override** a superclass feature by defining a feature with the same name.

The overriding feature (in the sub-class) replaces the overridden feature (in the super-class)

Never override a feature so that it is inconsistent with the original inherited feature.

Overriding should preserve the attribute type, number and type of arguments of an operation and its return type.

GENERALIZATION, SPECIALIZATION AND INHERITANCE



length x width

$\pi \times \text{radius}^2$

side^2

FURTHER CLASS CONCEPTS *ABSTRACT CLASS*

An incomplete class that cannot be instantiated.

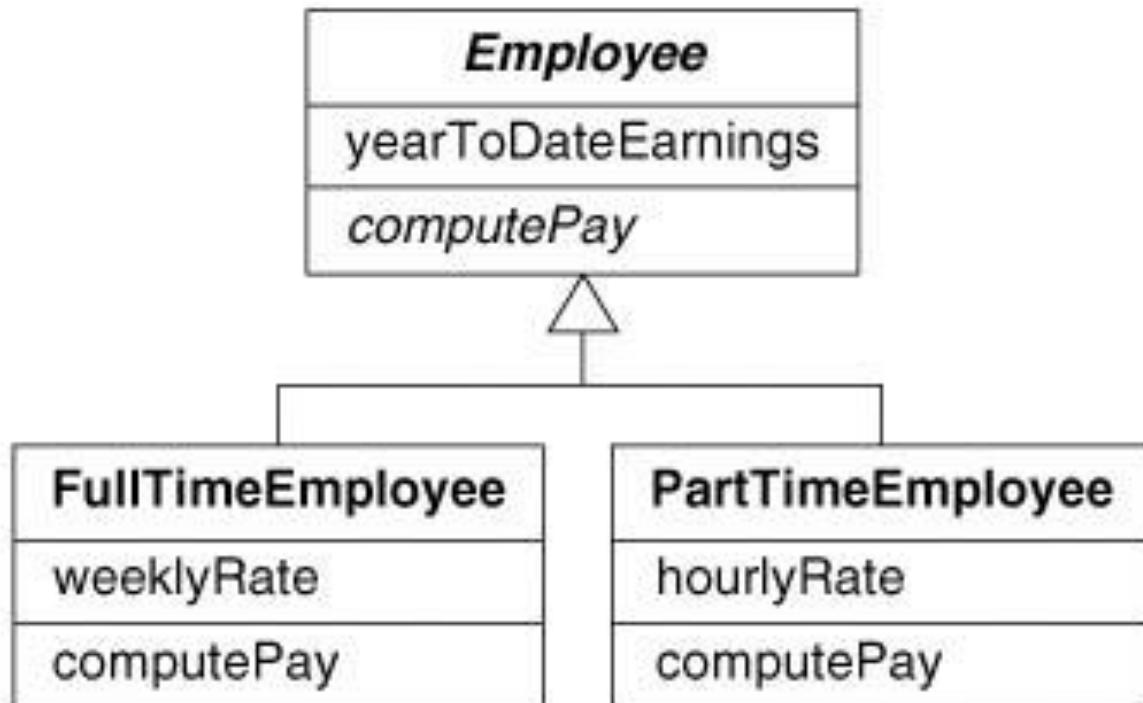


Figure 4.13 Abstract class and abstract operation. An abstract class is a class that has no direct instances.

OBJECT DIAGRAMS

An object model: is a collection of objects, where every object in the model is an instance of a class in the class model.

- it represents a snapshot of the system at one instant in time.
- no object can have attributes or relationships that are not in the class model.

An object model may be thought of as an instantiation of a class model.

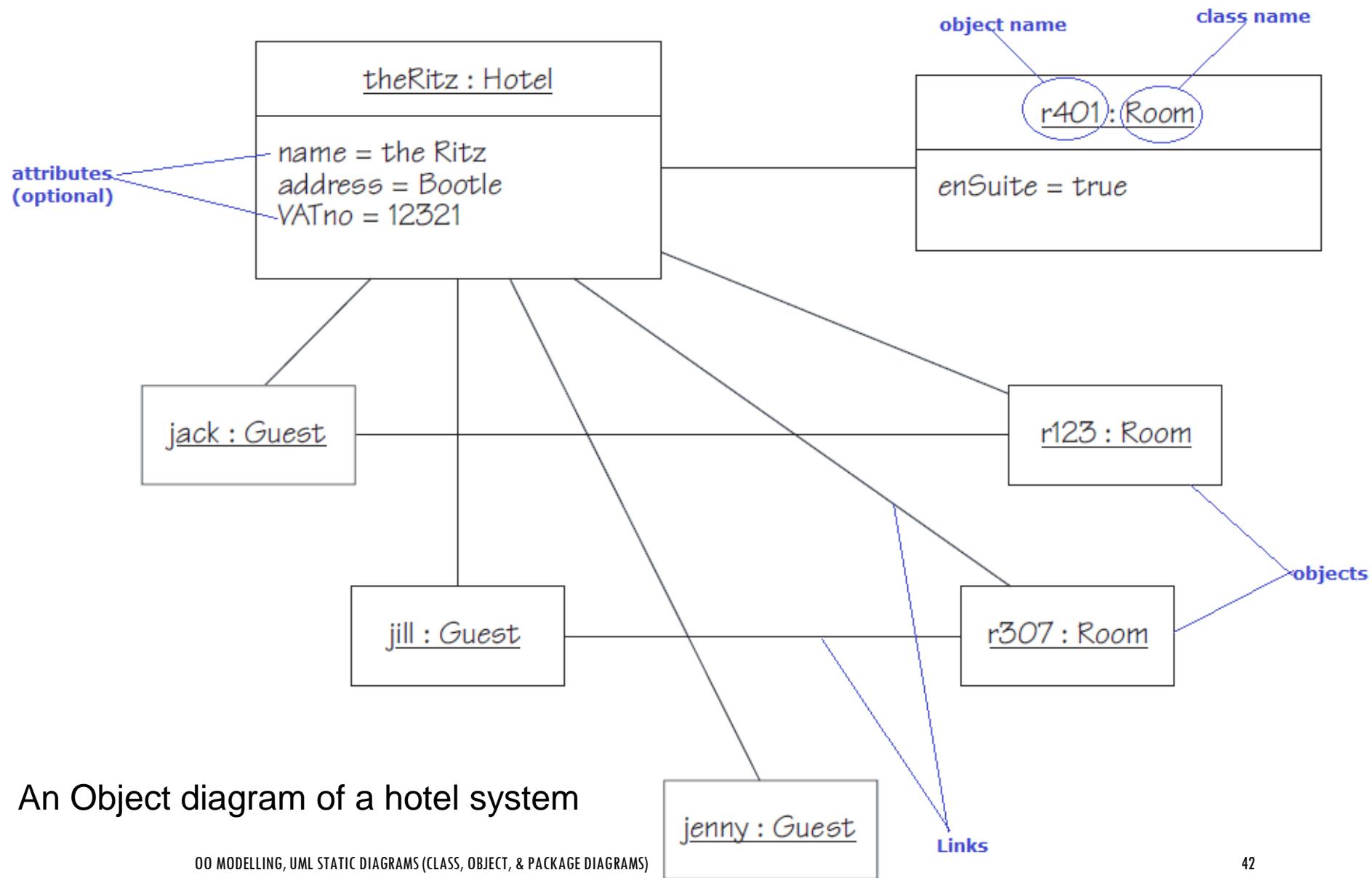
- It shows concrete instances of relevant classes & relationships.
- It indicates what connections do exist at that instant in time.
- Is visually represented using an object diagram.

OBJECT DIAGRAMS .. ELEMENTS

An **object diagram**: is a visual representation of an object model, showing a **snapshot** of the system at some point in time. The elements of this object diagram are:

- ❖ Rectangular boxes representing **objects**.
- Inside each box you add:
 - ObjectName : ClassName which are underlined.
 - Names and values of useful attributes in a second compartment (Optional)
- Lines drawn between objects are called **links**.
 - They represent particular cases of one object ‘knowing about’ another object.
 - Each link is an instance of an association (a relationship between classes).

OBJECT DIAGRAMS .. AN EXAMPLE



CLASS MODELING TIPS

The following categories are **useful sources of relevant objects:**

- **Tangible objects:** the physical things in the domain such as rooms, bills, books and vehicles;
- **Roles:** the roles played by people in the domain, such as employees, guests and members;
- **Events:** the circumstances, episodes, interactions, happenings and significant incidents, such as room reservations, vehicle registrations, orders, deliveries and transactions;
- **Organizational units:** the groups to which people belong, such as accounts departments, production teams and maintenance crews.

CLASS MODELING TIPS

Scope: A model represents the relevant aspects of the problem. *Exercise judgment in deciding which objects to show and which ones to ignore.*

Simplicity: *Make the model as simple as possible.* Simpler models are easier to understand and develop.

Diagram Layout: *Draw your diagram in a way that is easy to understand.* Avoid cross lines. Position important classes where they are visually prominent.

Names: *Carefully choose class names.* Choose descriptive names and use singular nouns to name classes.

DIVIDE & CONQUER: MODULARIZATION

The only way till now to reduce the complexity of systems is **modularization**.

Modularization means decompose system or problem to a smaller sub-system with separate boundaries (modules).

Examples of modules are:

- Whole programs or applications;
- Software libraries;
- Procedures & functions in classical PLs;
- Classes, in an OO PL such as Java.

DIVIDE & CONQUER: MODULARIZATION

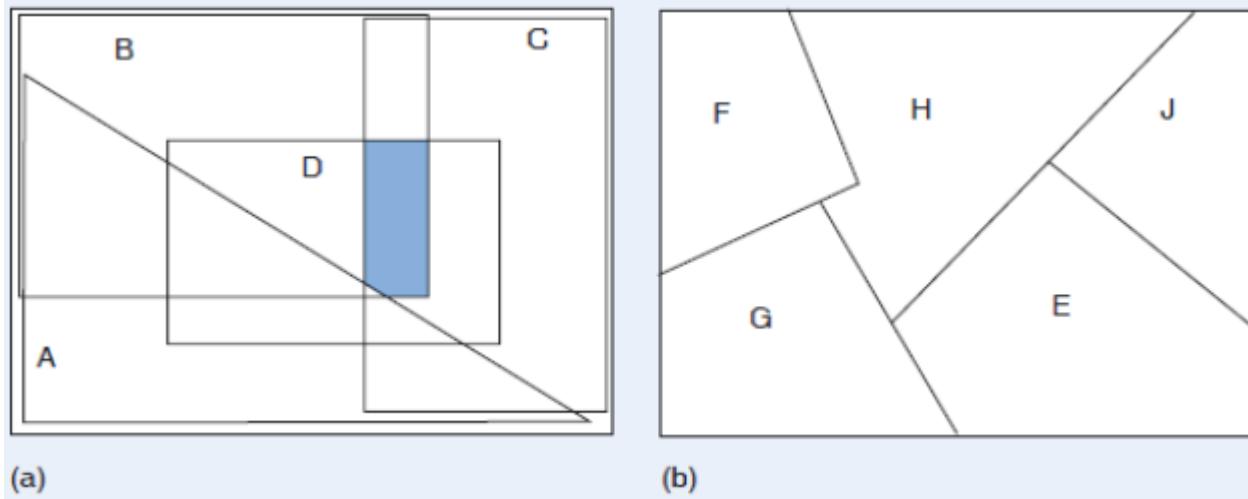
Modules in themselves are not “good” ..

Must design them to have good properties ..

Good modularization must have:

- Maximal relationships within modules (Cohesion)
- Minimal relationships between modules (Coupling)

DIVIDE & CONQUER: MODULARIZATION



There are two forms of decomposition:

- (a) Projection:** dependent modules, common elements between modules.
- (b) Partitions:** independent modules, which are easy to use.

Note: Usually even if the modules sound separate there are relationships between them which must be taken into consideration through the software development and during maintenance.

COHESION

Definition

- The degree to which all elements of a component are directed towards a single task.
- The degree to which all elements directed towards a task are contained in a single component.
- The degree to which all responsibilities of a single class are related.

Internal glue with which component is constructed

All elements of component are directed toward and essential for performing the same task.

TYPES OF COHESION

- | | | |
|----|--|--------|
| 7. | {
Informational cohesion
Functional cohesion | (Good) |
| 5. | Communicational cohesion | |
| 4. | Procedural cohesion | |
| 3. | Temporal cohesion | |
| 2. | Logical cohesion | |
| 1. | Coincidental cohesion | (Bad) |

Functional

Sequential

Communicational

Procedural

Temporal

Logical

Coincidental

Low

COINCIDENTAL COHESION

Def: Parts of the component are unrelated (unrelated functions, processes, or data) .. Module performs multiple, completely unrelated actions ..

- Parts of the component are only related by their location in source code.
- Elements needed to achieve some functionality are scattered throughout the system.
- Accidental
- Worst form

FUNCTIONAL COHESION

Def: Every essential element to a single computation is contained in the component .. Module performs exactly one action ..

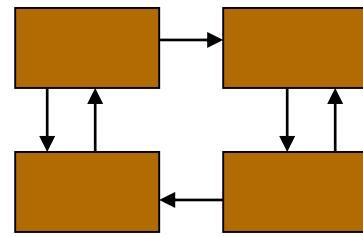
- Every element in the component is essential to the computation.
- Ideal situation
- What is a functionally cohesive component?
 - One that not only performs the task for which it was designed but
 - it performs only that function and nothing else.

COUPLING

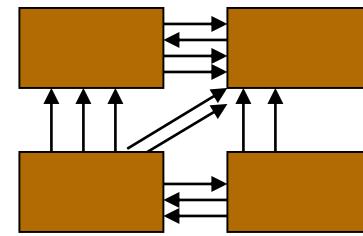
- The degree of dependence such as the amount of interactions among components.



No dependencies



Loosely coupled
some dependencies



Highly coupled
many dependencies

COUPLING .. EXAMPLES

A component **depends** on another if a change to one requires a change to another.

Examples:

Circular Dependency; a relation between two or more modules which either directly or indirectly depend on each other to function properly. Such modules are also known as mutually recursive. Suppose we have a class called A which has class B's Object (*in UML terms: A HAS B*). At the same time class B is also composed of an Object of class A (*in UML: terms B HAS A*). Obviously this represents a circular dependency because while creating an object of A, the compiler must know of B .. On the other hand, while creating an object of B, the compiler must know of A.

Chain Dependency; Module A depends on B and B depends on C (chain of interdependent modules) ..

COUPLING .. EXAMPLES

- One component modifies another.
- More than one component share data such as global data structures.
- Component passes a data structure to another component that does not have access to the entire structure.

CONSEQUENCES OF COUPLING

High coupling

- Components are difficult to understand in isolation
- Changes in component ripple to others
- Components are difficult to reuse
 - Need to include all coupled components
 - Difficult to understand

Low coupling

- May incur performance cost
- Generally faster to build systems with low coupling

PACKAGE DIAGRAM

- Packages let you organize large models so that they are more understandable.

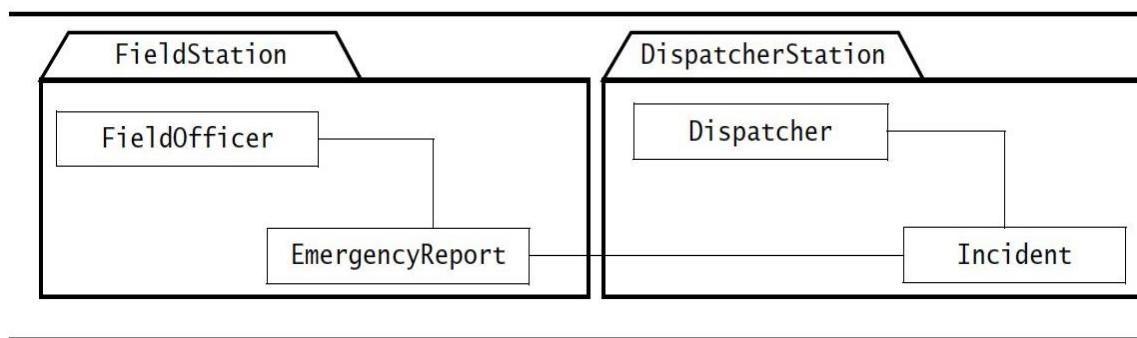
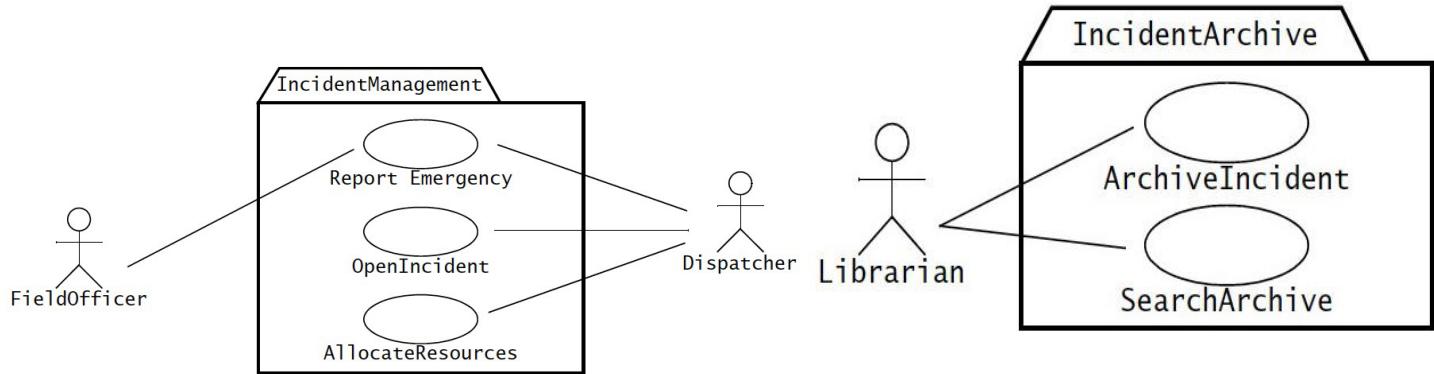
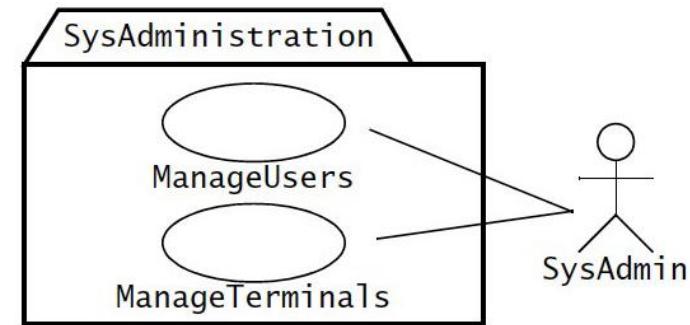


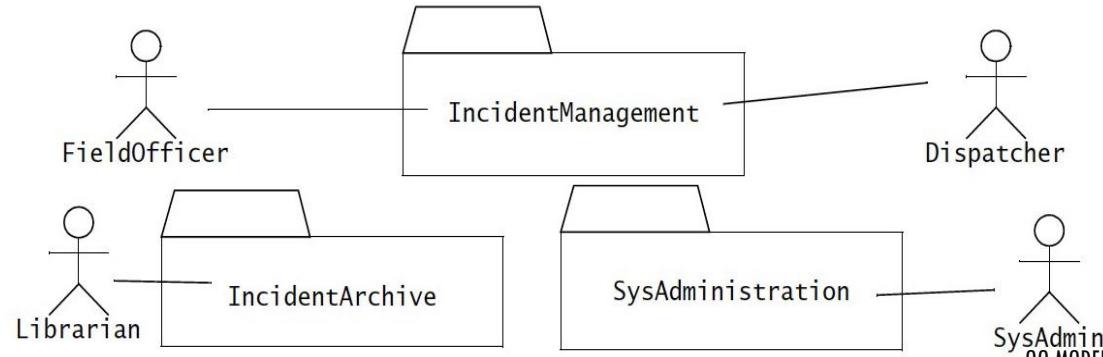
Figure 2-47 Example of packages. The `FieldOfficer` and `EmergencyReport` classes are located in the `FieldStation` package, and the `Dispatcher` and `Incident` classes are located on the `DispatcherStation` package.

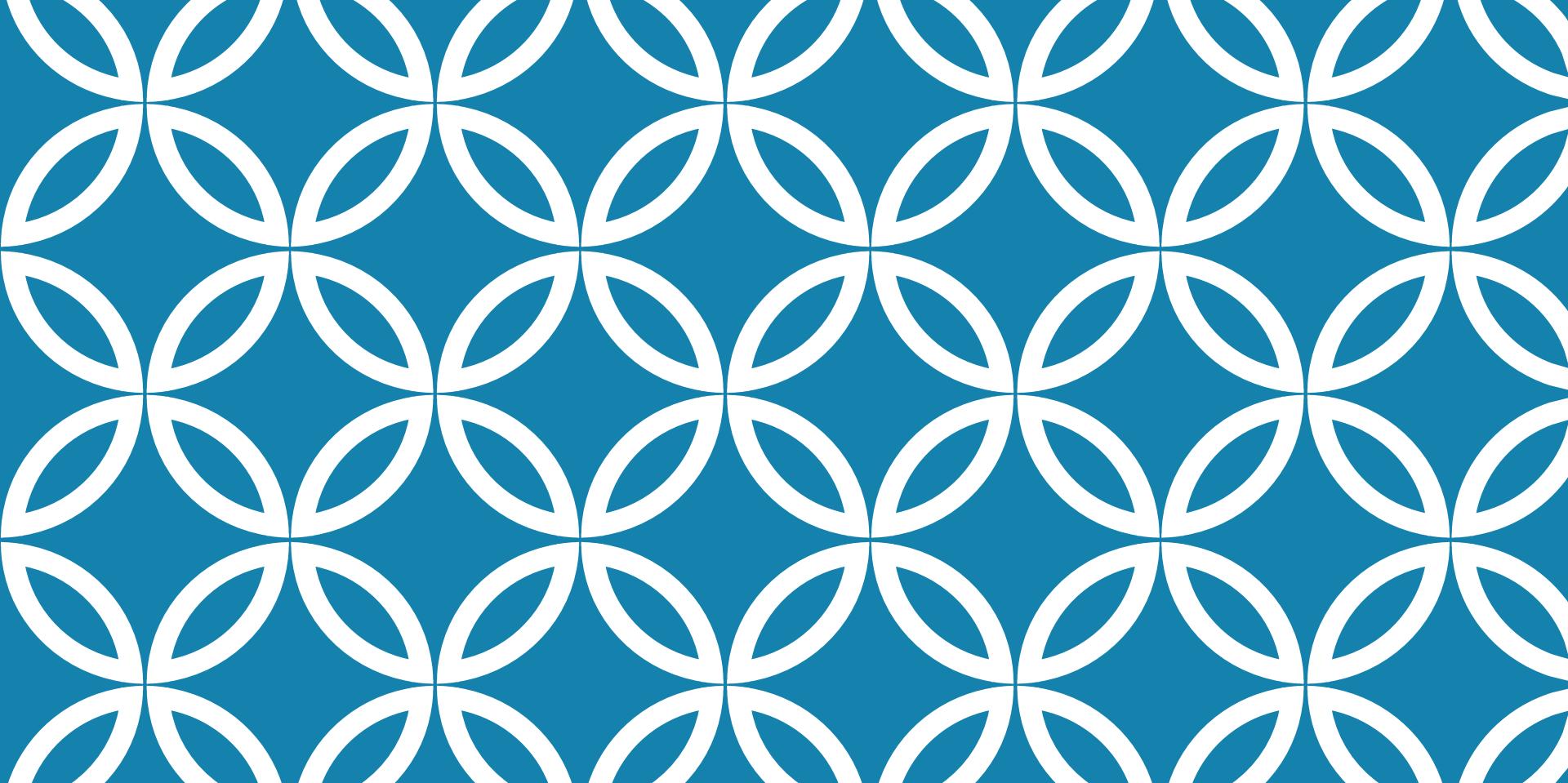
Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

PACKAGE DIAGRAM



Example of packages: Use-Cases are organized by actors





(CS251) SOFTWARE ENGINEERING I

Lecture 6
OO Modelling, Interaction
Diagrams (Sequence,
Collaboration, & System
Sequence Diagrams)

LECTURE OBJECTIVES

1. Learn the notation of interaction diagrams.
2. Learn how to visualize system-user interaction using system sequence diagrams, collaboration diagrams, system sequence diagrams.
3. Learn how to decide upon issues such as fork & cascade, or how to implement use-cases.



UML DIAGRAMS

Functional Diagrams (*Use case diagrams*)

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

Static Diagrams (*Class, Object, & Package diagrams*)

- Describe the static structure of the system: Objects, attributes, associations.

Dynamic Diagrams:

- Interaction diagrams (Sequence, & Collaboration diagrams)
- Describe the interaction between objects of the system.
- State diagrams
- Describe the dynamic behavior of an individual object.
- Activity diagrams
- Describe the dynamic behavior of a system, in particular the workflow.

UML DYNAMIC MODELLING .. WHY?

UML static modeling involves **use cases, activity diagrams** and **class/object modeling**:

- **Use cases** describe what a system should do (*the required behavior/operations*). It did not say which classes should be responsible for which parts of computation
- An initial **Class model** identifies classes and its relationship but do not contain complete list of attributes and operations.
- Thus, a link is required between use cases and class modeling, which leads us to what is known as **dynamic modeling**.
- **Dynamic Modelling** shows how the objects interact by sending messages to implement the required functionality of the software system **(to realize a Use-Case)**.

UML DYNAMIC MODELLING: INTERACTION DIAGRAMS

Interaction diagrams are UML notations representing dynamic modeling that are used to help make and record decisions relating to the behavior (*operations*) defined for each class.

There are two types of interaction diagrams:

1. Sequence Diagram ..
2. Communication (Collaboration) Diagram ..

Both are semantically equivalent to each other, and used for quite similar purposes, but each has different features as described next.

UML INTERACTION DIAGRAMS

SEQUENCE VERSUS COLLABORATION DIAGRAMS

	Sequence Diagrams	Collaboration (Communication) Diagrams
Definition	Shows object interaction arranged in time sequence. That is, a 2-dimensional diagram, horizontally the objects participating in the interaction are depicted, while vertical dimension represents time.	Shows how cooperating objects dynamically interact with each other by sending and receiving messages.
Strengths	Time relationship extremely clear.	Objects' interactions extremely clear.
Weaknesses	Not easy to see the overall pattern of message flow.	Actual sequence of messages is difficult to see.

SEQUENCE DIAGRAMS

A **sequence diagram** shows the participants (objects) in an interaction (a use case or part of it) and the sequence of messages among them.

Each Use Case requires one or more sequence diagrams to describe its behavior.

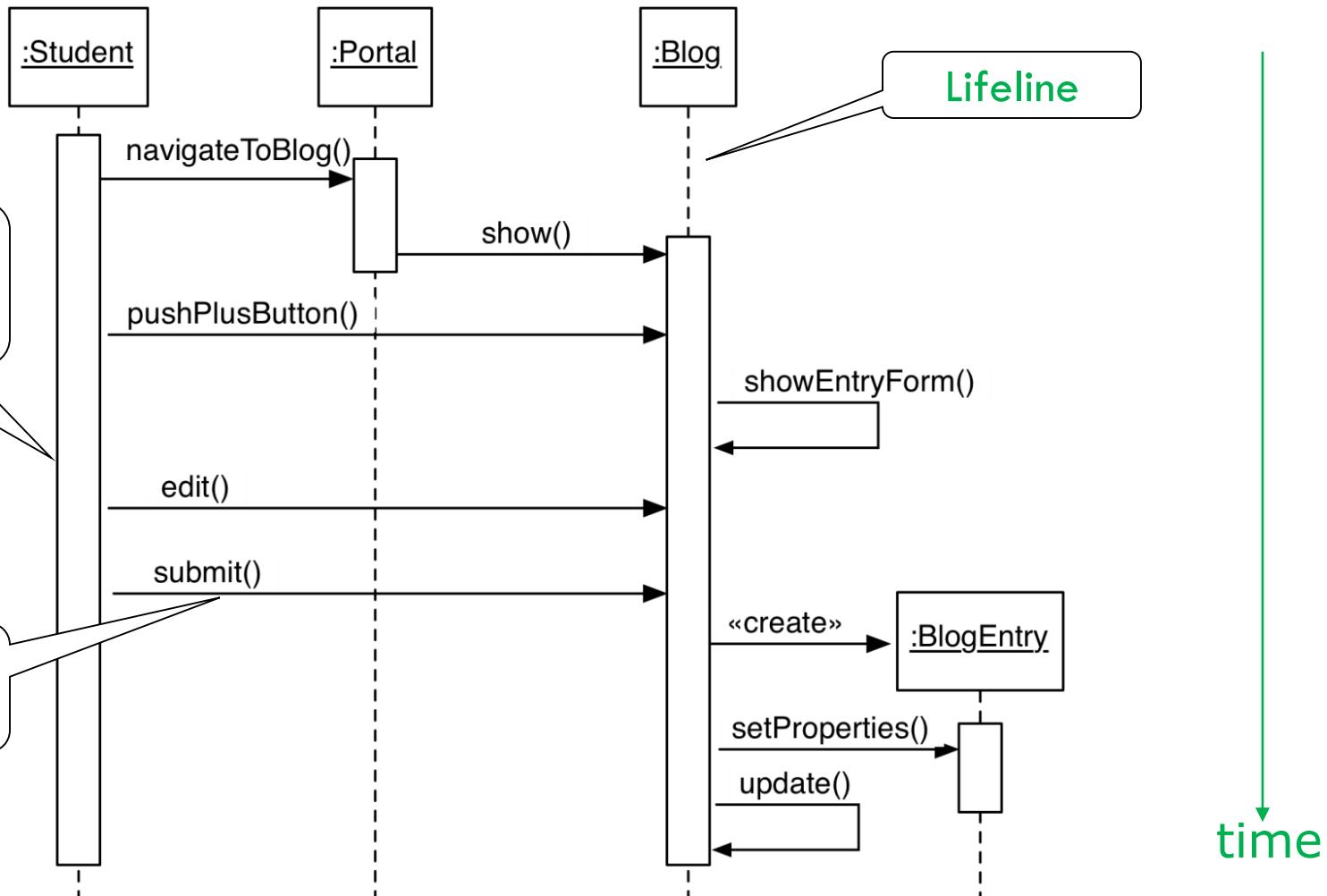


SEQUENCE DIAGRAMS

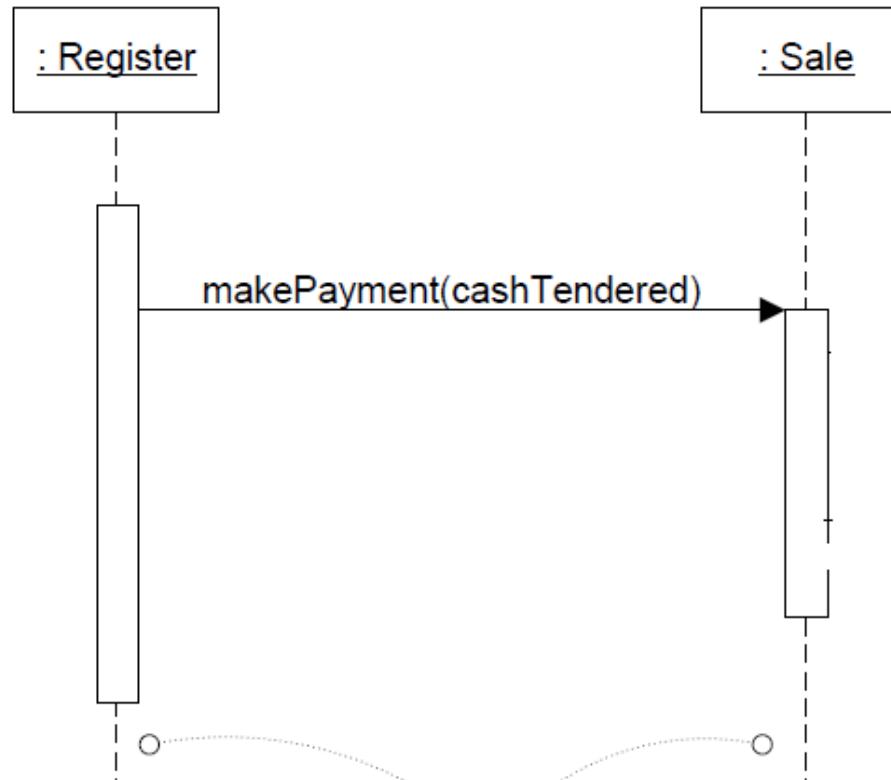
Starting by building a sequence diagram:

- Begin by looking at **interactions implied by use cases**. Consider a **use case scenario** to be modeled.
- You need to know where the **first message** in the use case originates. (*for example, a user interface*).
- You need to consider the **message flow**;
 - To where the initial message should be sent?
 - This implies which class should have this method in its protocol.
- Use the **post-condition(s)** of the use cases which reflect how the system must change.
 - Take into consideration the **main scenario and the alternative ones**.
 - It may be helpful to **use a pair of object diagrams**, showing the states before and after the operation in question.

SEQUENCE DIAGRAM: BASIC NOTATION SUMMARY

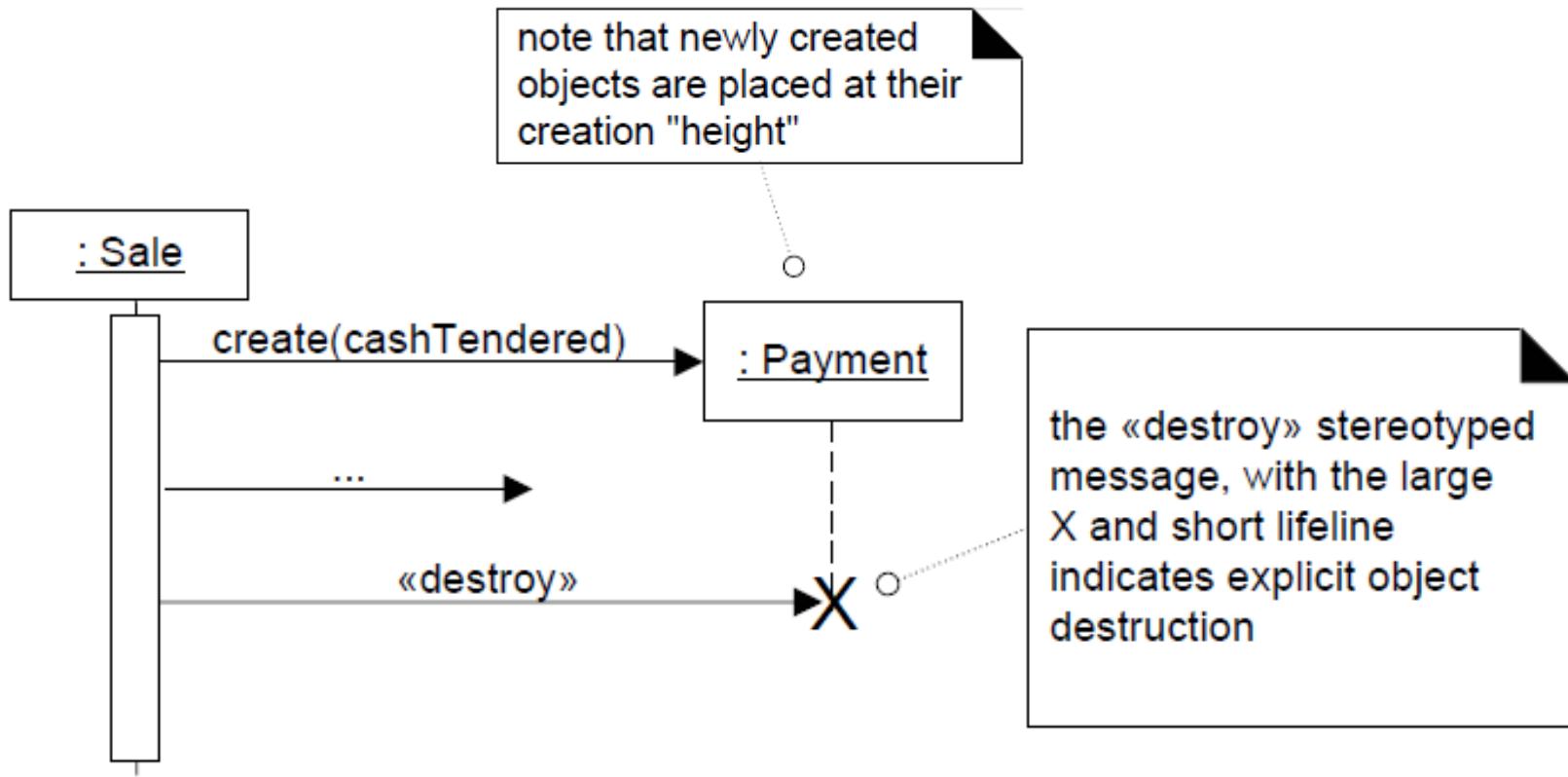


OBJECT LIFELINES

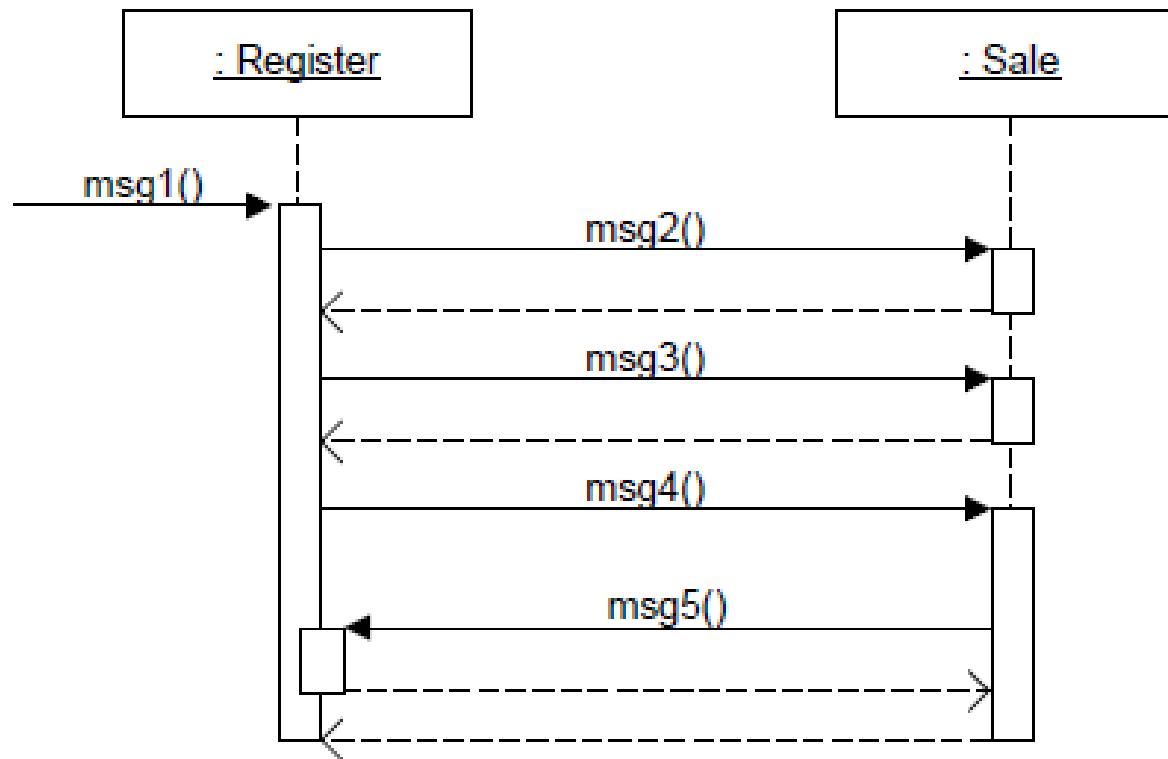


an object lifeline shows the extent of
the life of the object in the diagram

OBJECTS CREATION AND DESTRUCTION



MESSAGES, RETURNS & ACTIVATION BOXES



CONDITIONAL MESSAGES

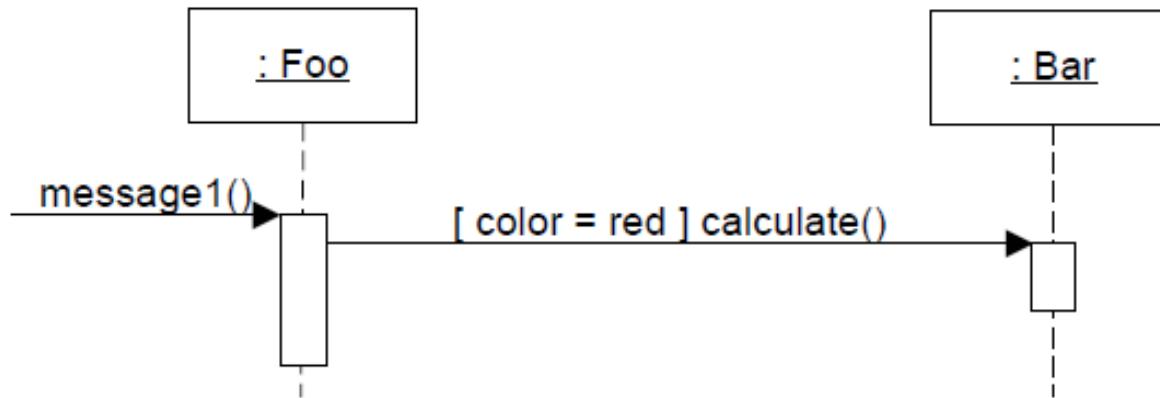
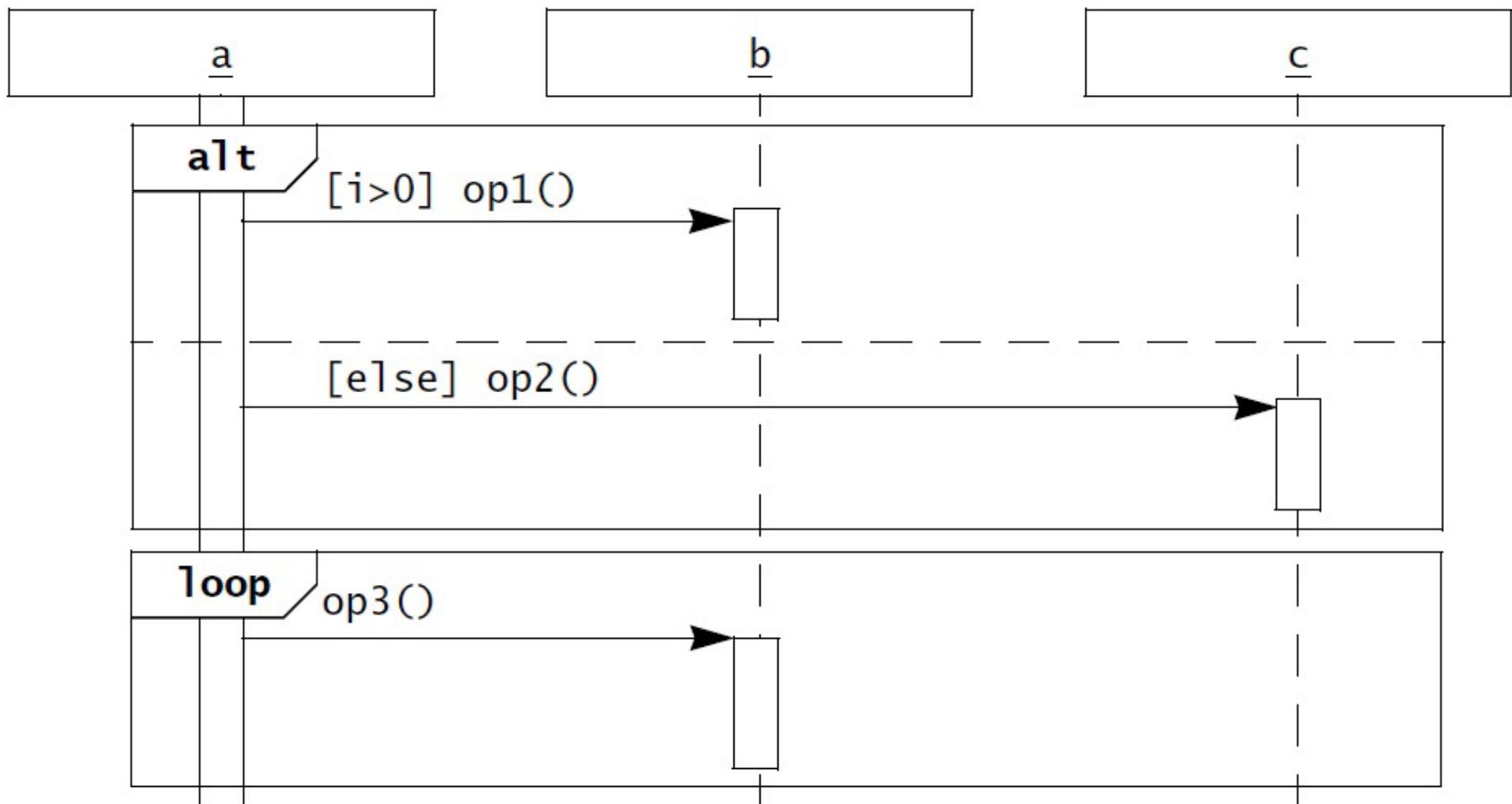
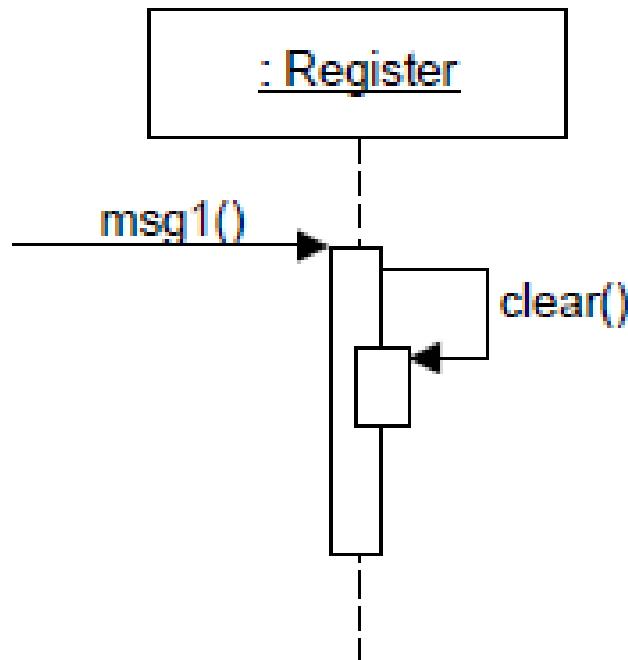


Figure 15.22 A conditional message.

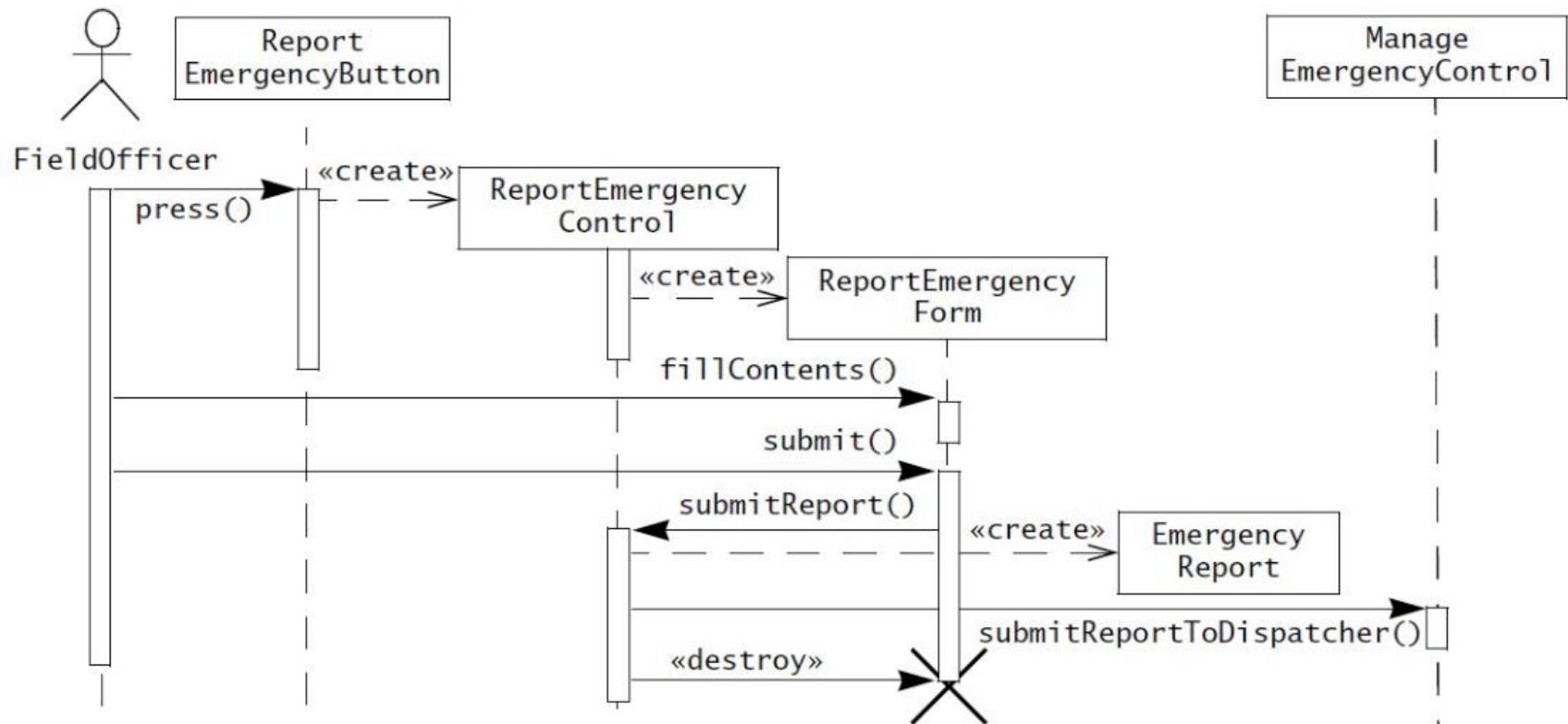
CONDITIONS AND ITERATIONS



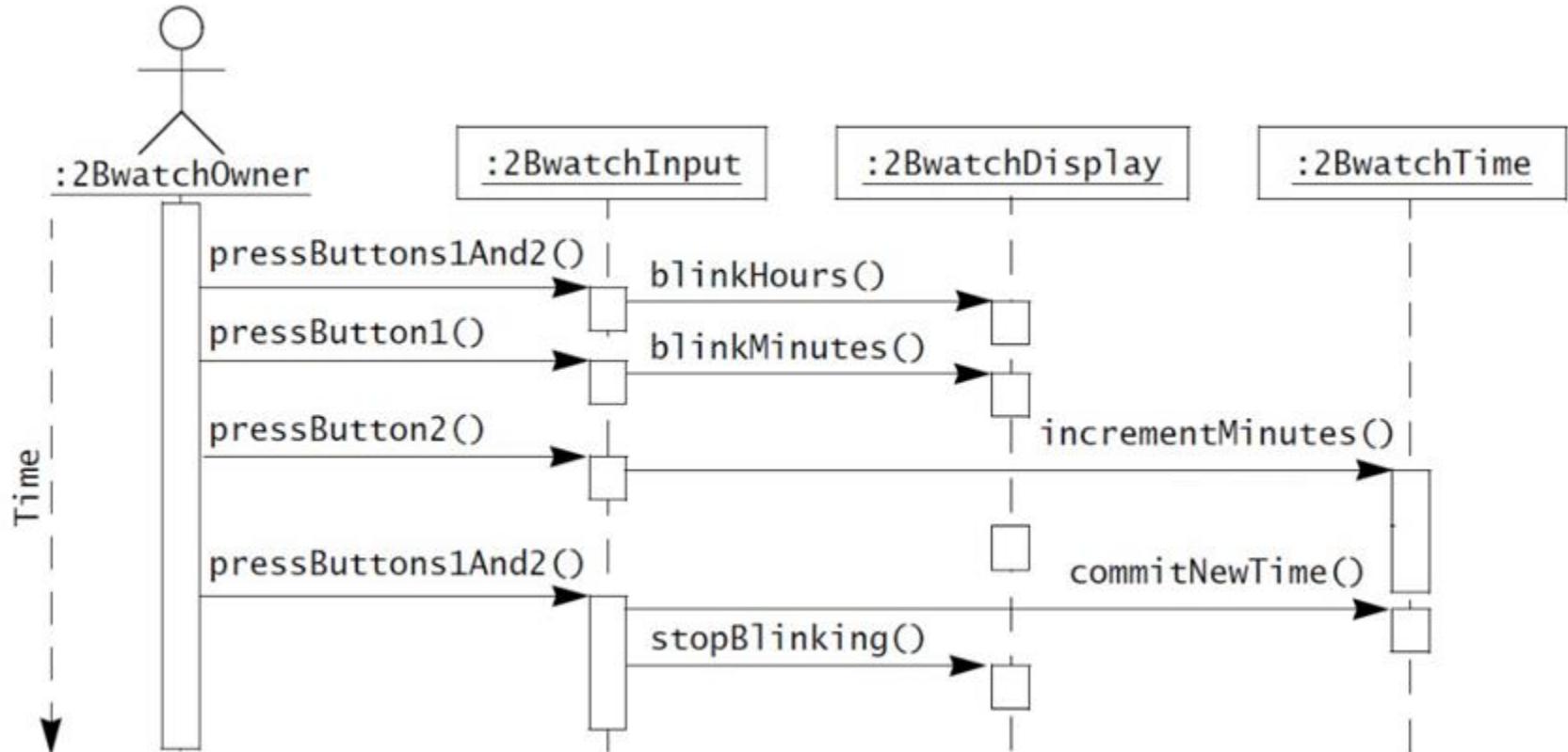
MESSAGES TO "SELF" OR "THIS"



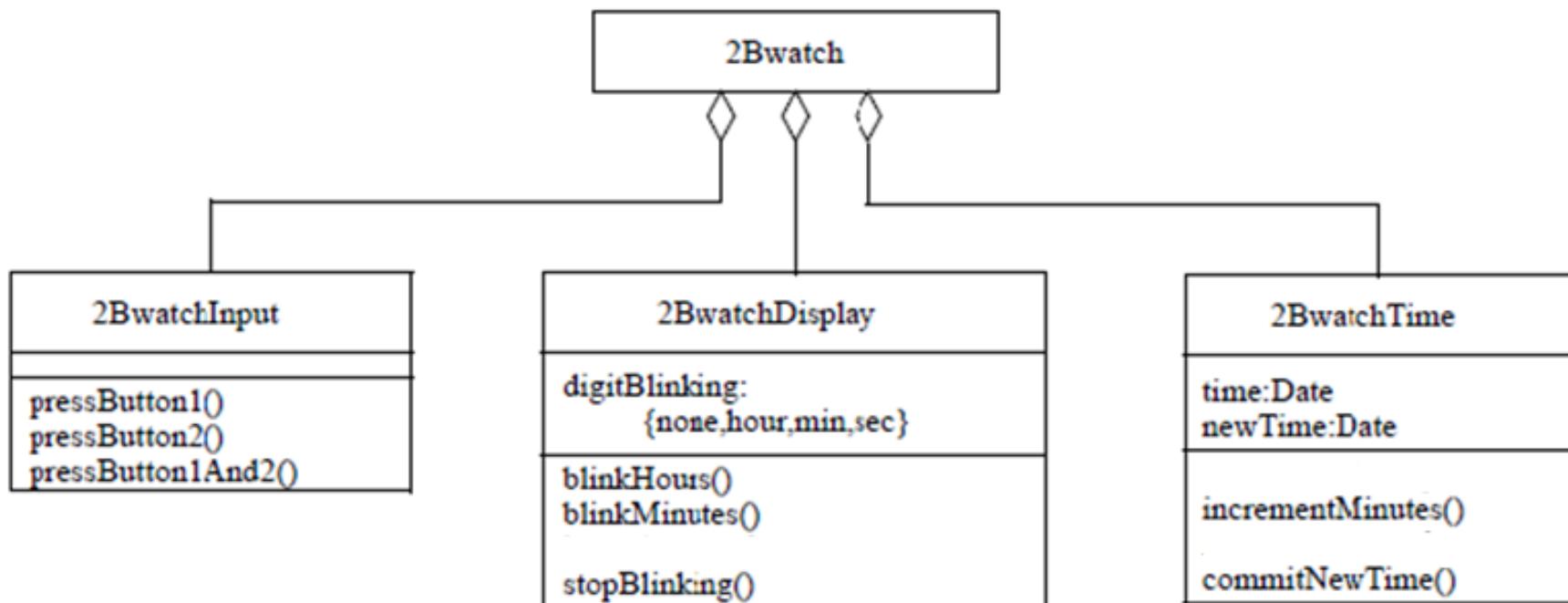
SEQUENCE DIAGRAM: EXAMPLE 2

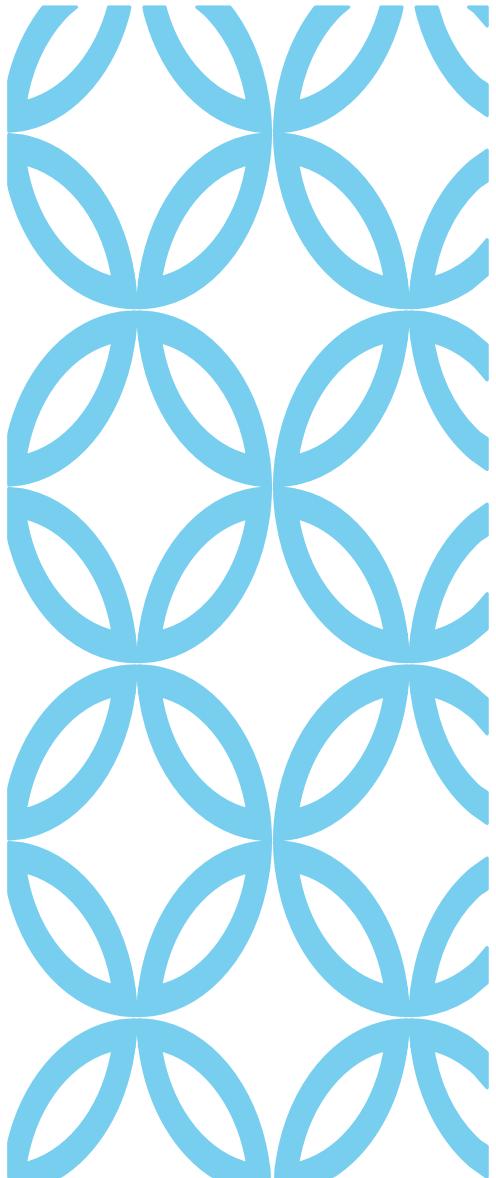


SEQUENCE DIAGRAM: EXAMPLE 3



SEQUENCE DIAGRAM: EXAMPLE 3 (SEQUENCE TO CLASS DIAGRAM)





A **communication (collaboration) diagram** is an object diagram with added message-sends.

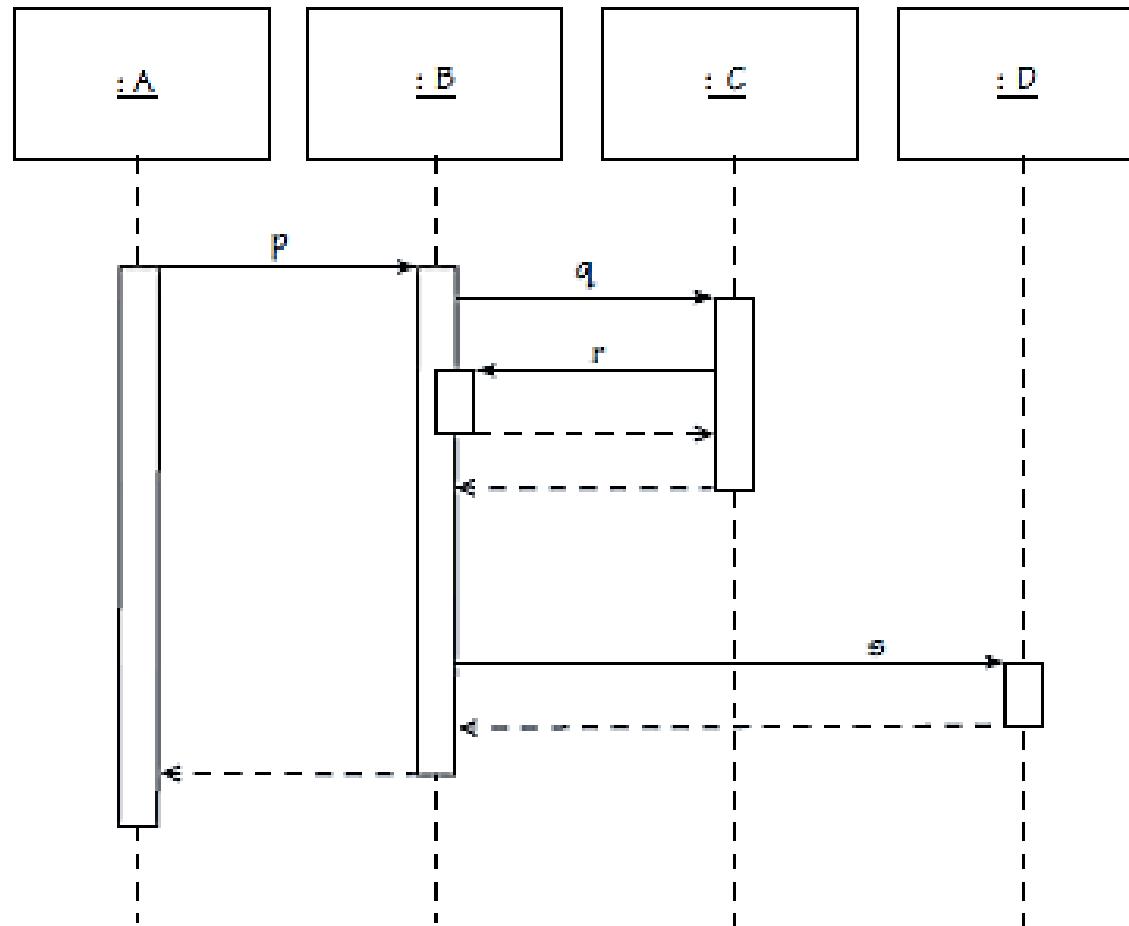
Every message has a multi-stage number (1.1).

The numbers specify the sequencing, replacing vertical position in a sequence diagram.

COLLABORATION DIAGRAMS

COLLABORATION DIAGRAMS

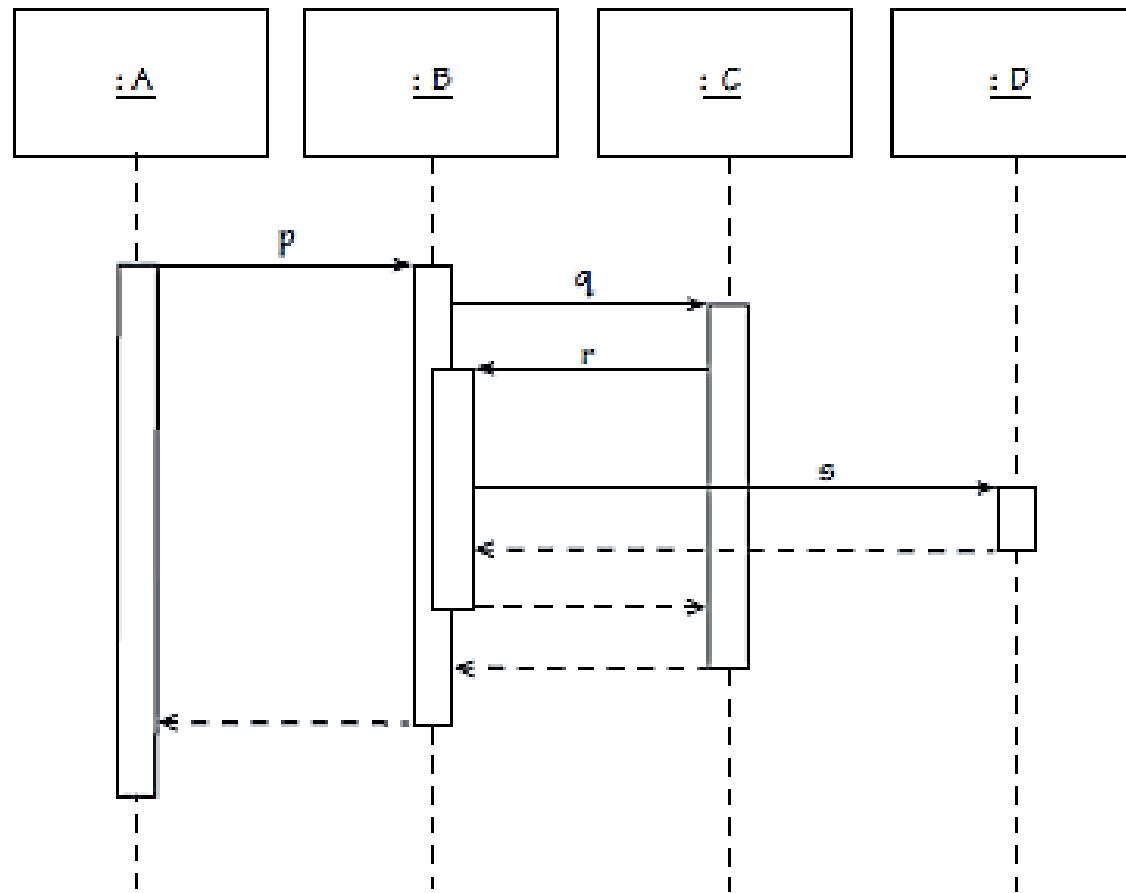
CONVERSION FROM A SEQUENCE DIAGRAM



(a) first message
sequence

COLLABORATION DIAGRAMS

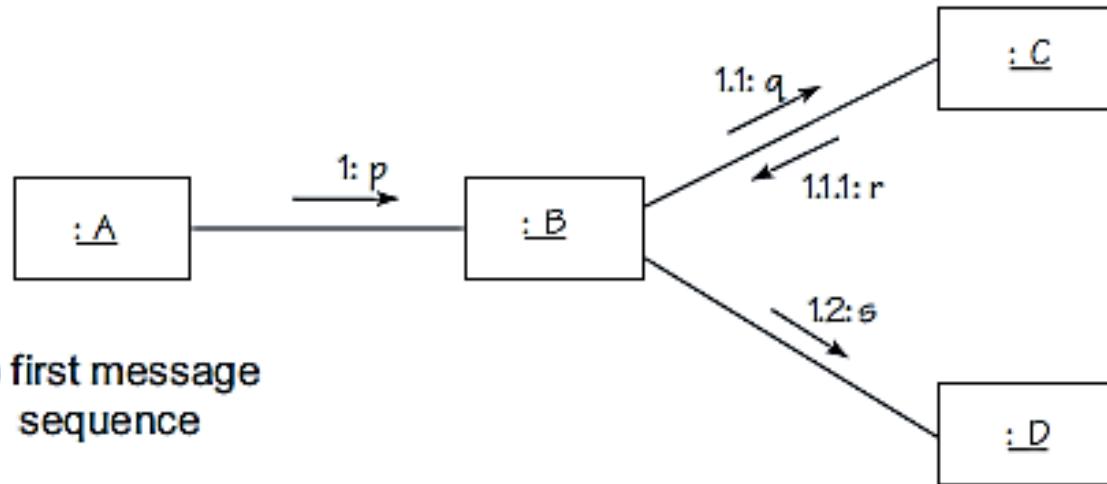
CONVERSION FROM A SEQUENCE DIAGRAM



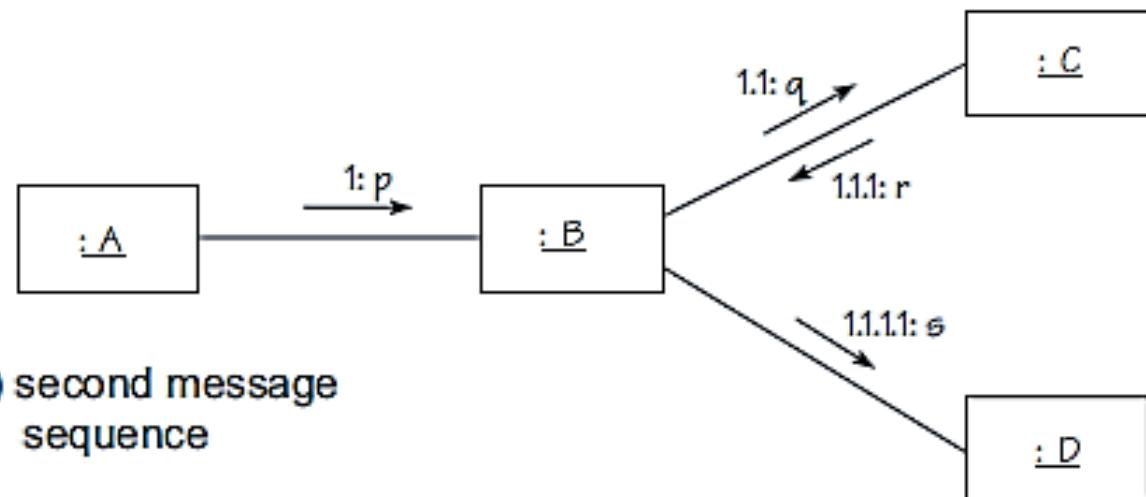
(b) second message sequence

COLLABORATION DIAGRAMS

CONVERSION FROM A SEQUENCE DIAGRAM



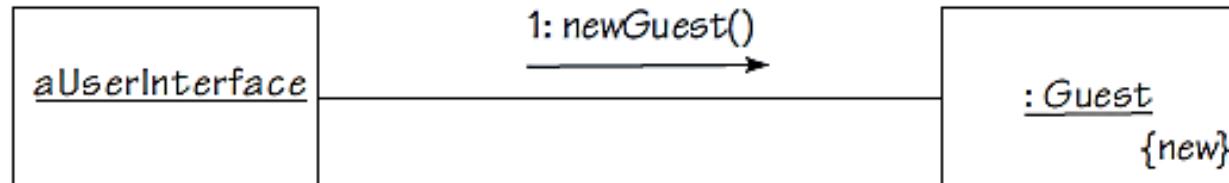
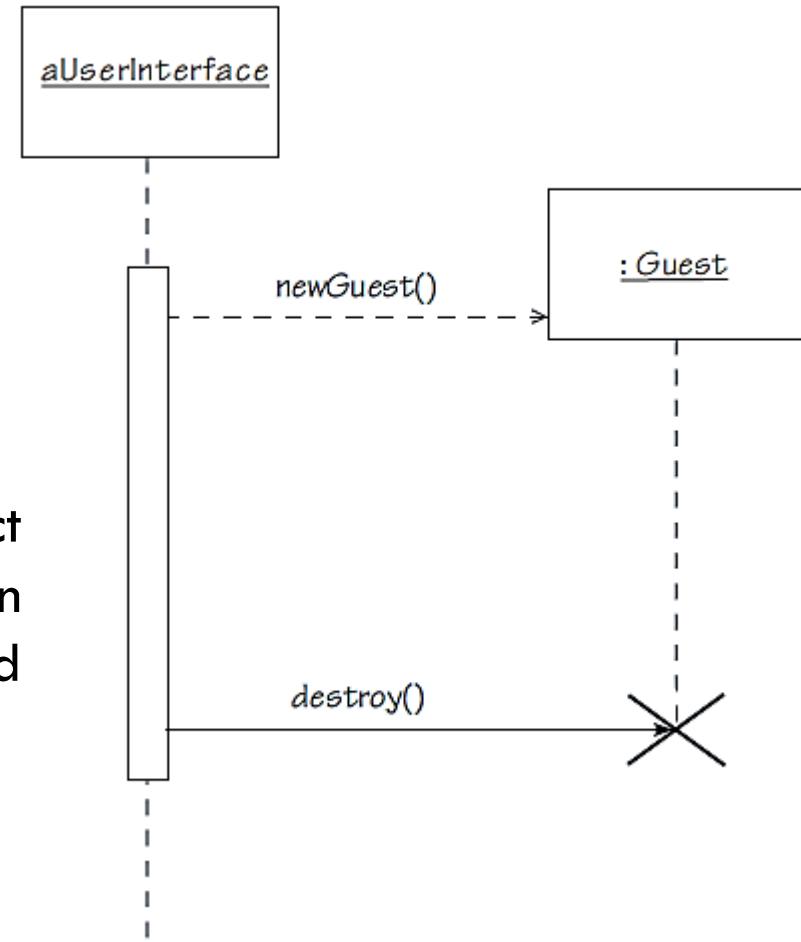
(a) first message sequence



(b) second message sequence

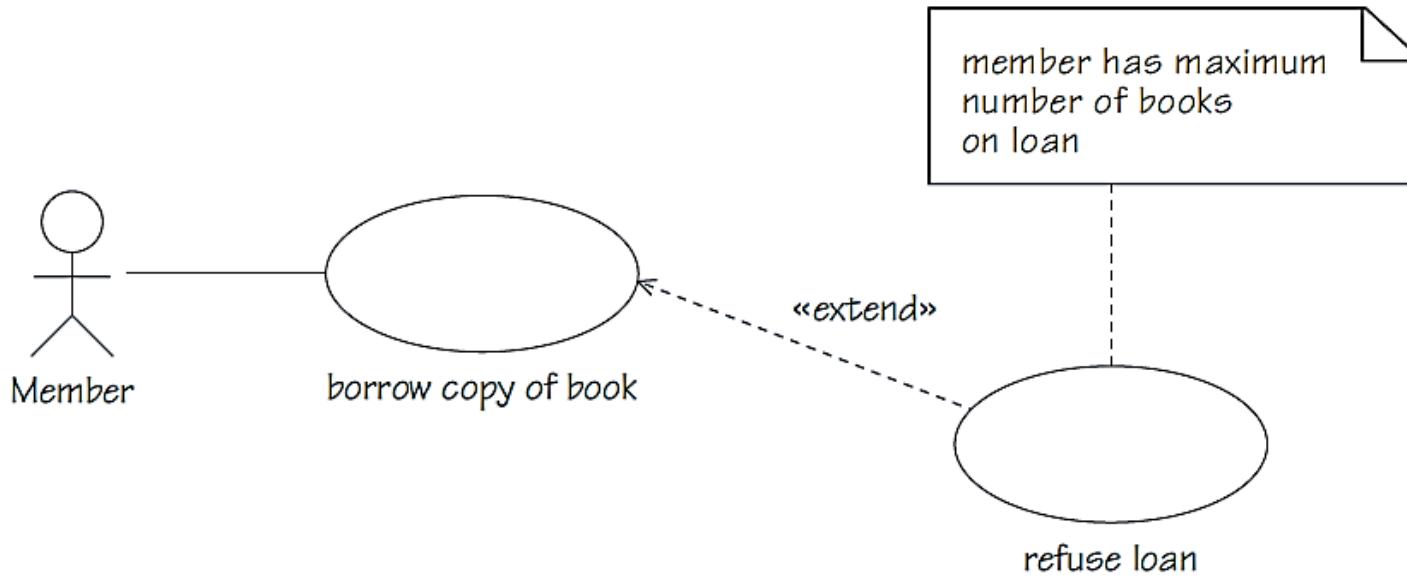
OBJECTS CREATION AND DESTRUCTION

In a communication diagram: object creation and destruction are shown by the special constraints {new} and {destroyed} respectively.

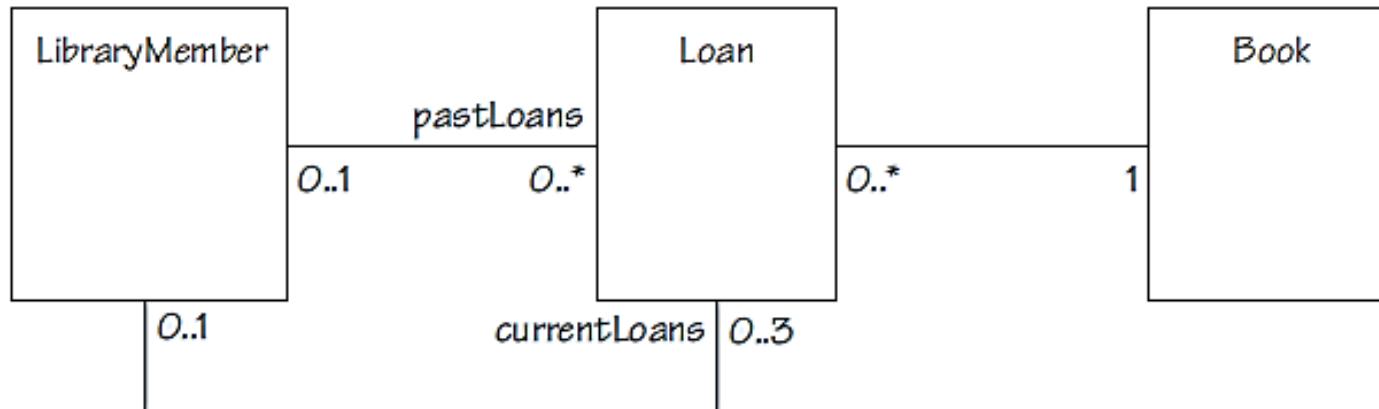


COLLABORATION DIAGRAMS

AN EXAMPLE .. USE-CASE & CLASS DIAGRAMS



- A class model for lending books, where there is a requirement to record both past and current loans



COLLABORATION DIAGRAMS

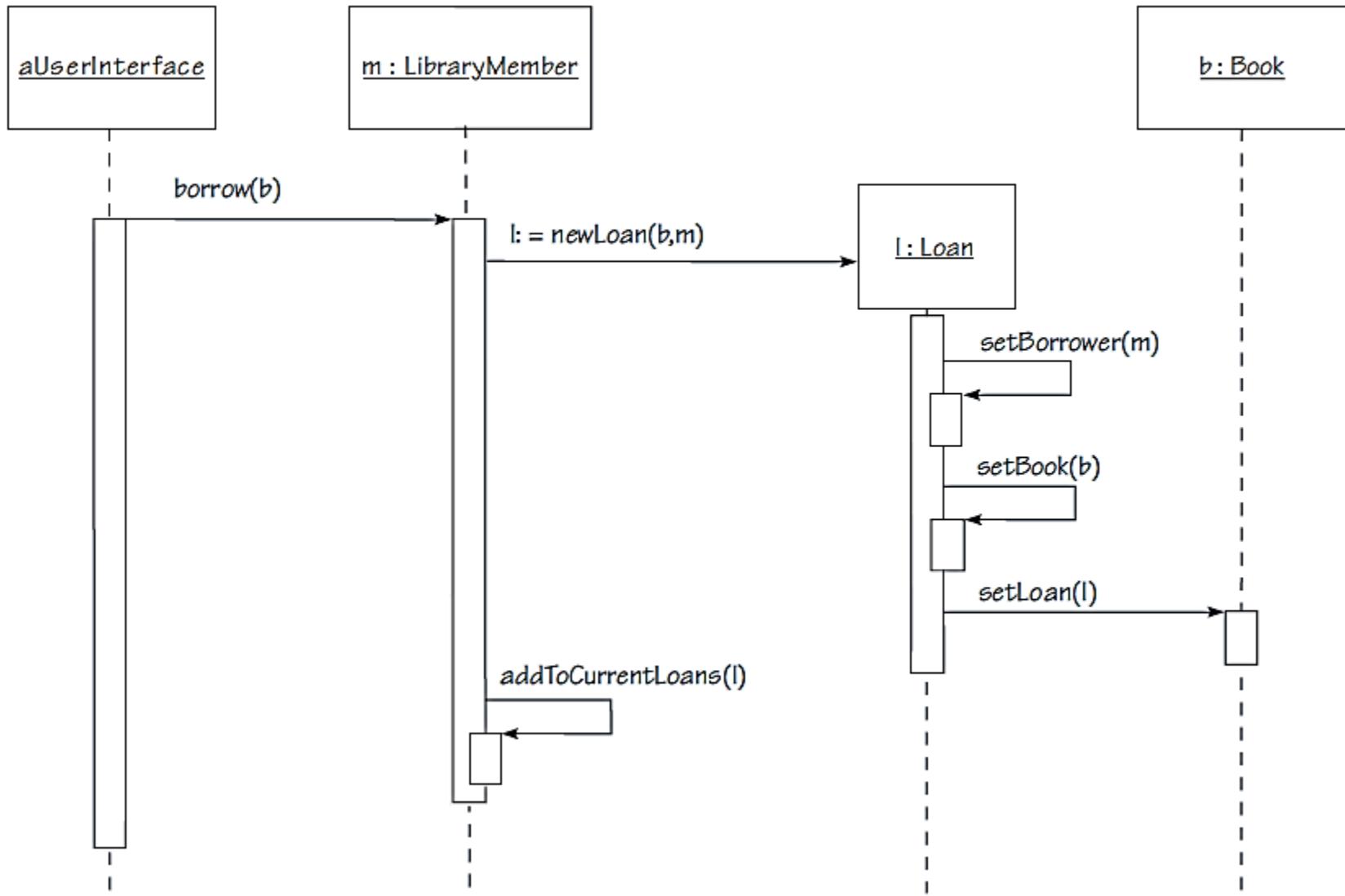
AN EXAMPLE .. FROM STATIC TO DYNAMIC MODELLING

Represent the interaction for the borrowing of a book.

- Draw both a sequence diagram and a communication diagram that sends the message `borrow(b)` to an instance of `LibraryMember` from `aUserInterface`, where `b` is a reference to the object representing the book that the library member `m` wants to borrow.
- *At this point we are not concerned about where the object `b` came from.*

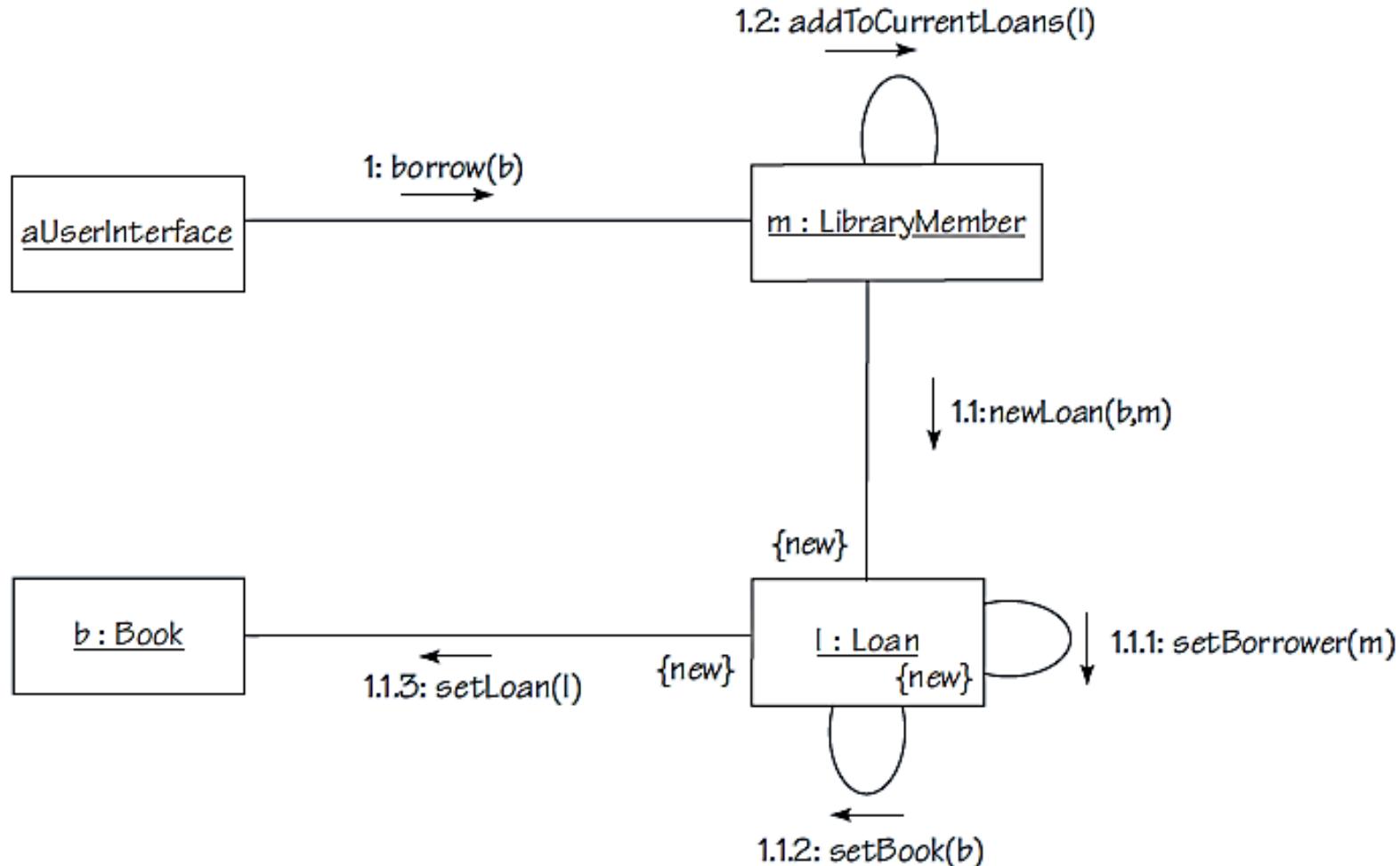
COLLABORATION DIAGRAMS

AN EXAMPLE .. THE SEQUENCE DIAGRAM



COLLABORATION DIAGRAMS

AN EXAMPLE .. THE COLLABORATION DIAGRAM



UML INTERACTION DIAGRAMS

SEQUENCE VERSUS COLLABORATION DIAGRAMS

There are two main differences between communication diagrams and sequence diagrams:

1. A communication diagram shows in one place all the links of interest between objects, whereas a sequence diagram does not.
2. The time-ordering of messages is clear in a sequence diagram (Time is viewed as running vertically downwards). However, some form of numbering is needed in a communication diagram to show the time-ordering of messages.

COMMUNICATION DIAGRAM NOTATION SUMMARY

The rectangles represent the various objects involved that make up the application.

The same notation for classes and objects used on UML sequence diagrams are used on UML communication diagrams

The lines between the classes represent the relationships (associations, composition, dependencies, or inheritance) between them.

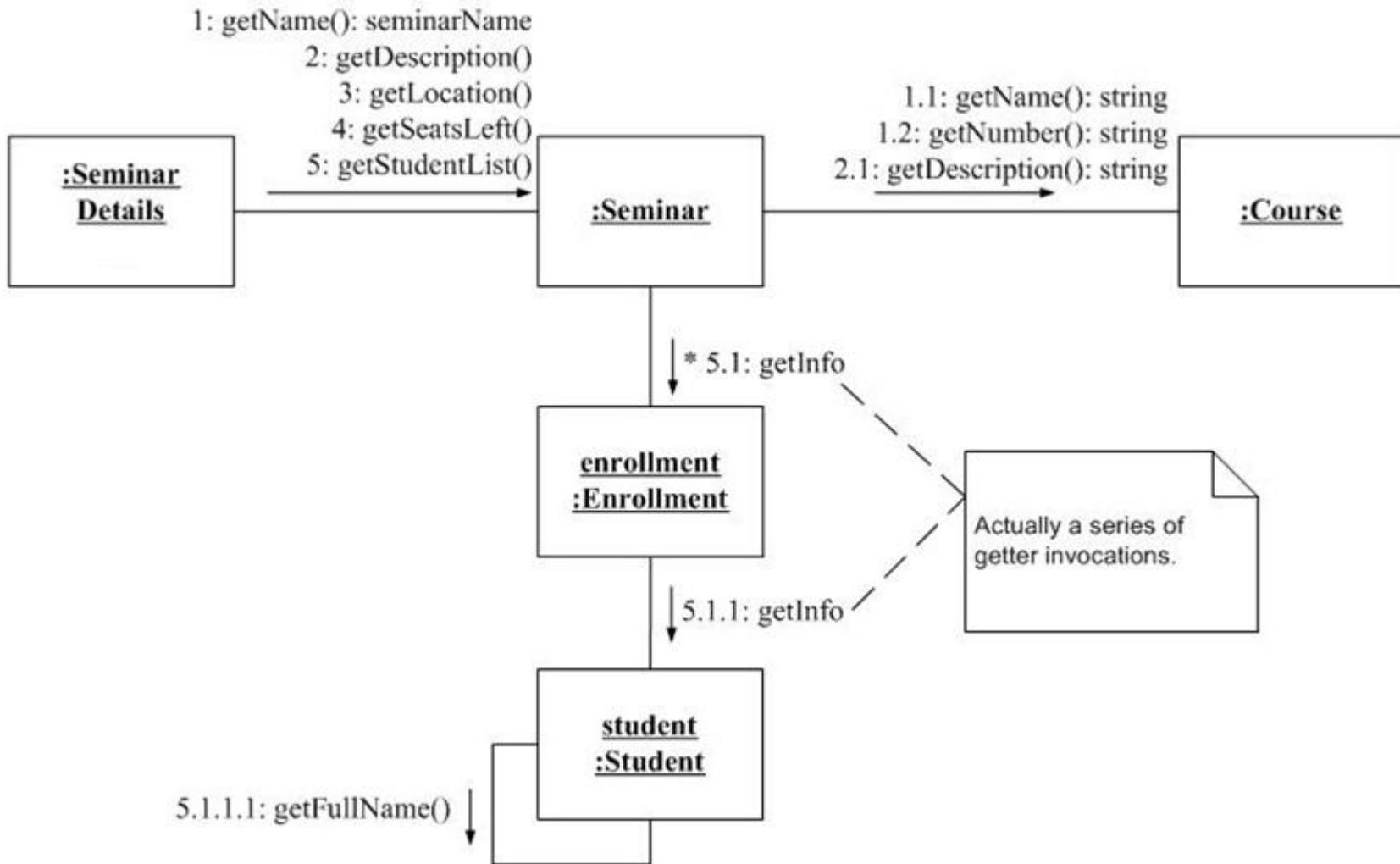
The details of the associations, such as their multiplicities, are not modeled because this information is contained on the class diagrams: remember, each UML diagram has its own specific purpose and no single diagram is sufficient on its own.

Messages are depicted as a labeled arrow that indicates the direction of the message, using a notation similar to that used on sequence diagrams.

Optionally, you may indicate the sequence number in which the message is sent, indicate an optional return value, and indicate the method name and the parameters (if any) passed to it. Sequence numbers should be in the format A.B.C.D to indicate the order in which the messages were sent. In the previous example message1 is sent to the Seminar object which in turn sends messages 1.1 and then 1.2 to the Course object. Message 5 is sent to the Seminar object, which sends message 5.1 to enrollment, which in turn sends message 5.1.1 to student, and it finally sends message 5.1.1.1 to itself.

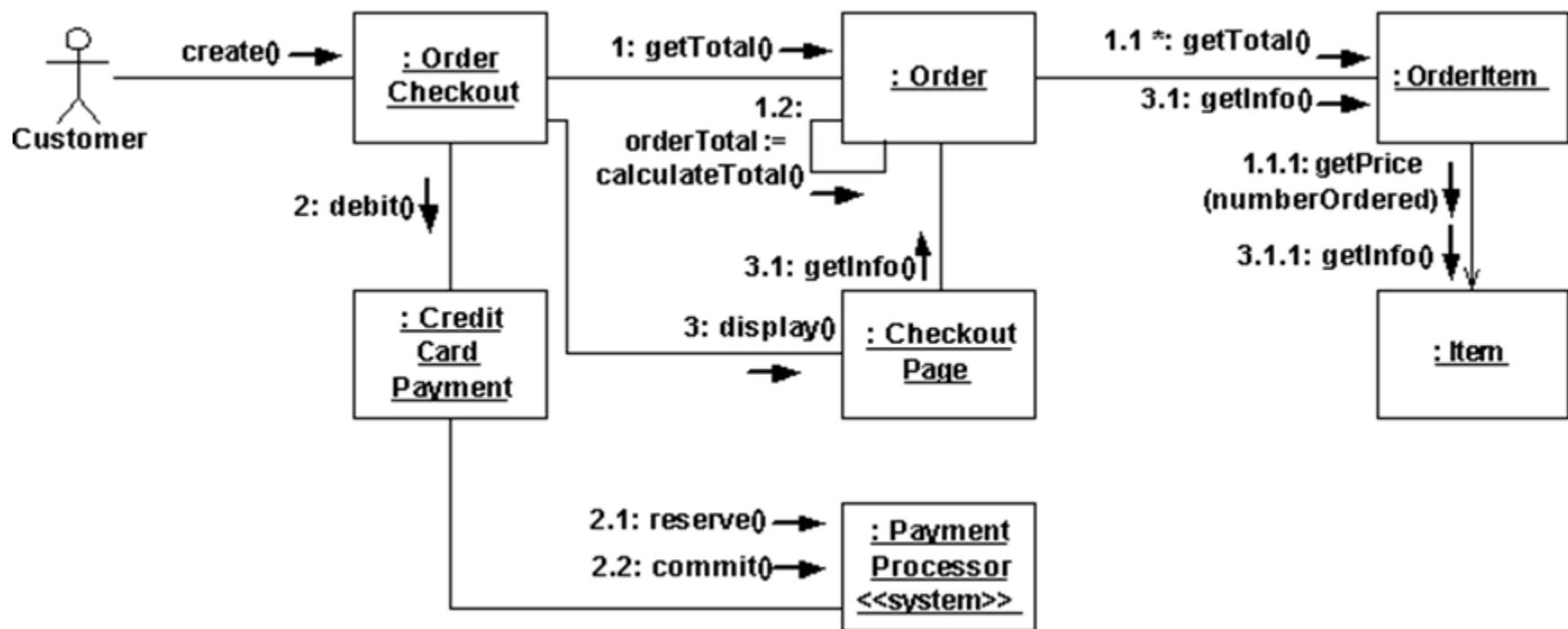
COMMUNICATION DIAGRAM

FURTHER EXAMPLES



COMMUNICATION DIAGRAM

FURTHER EXAMPLES



SYSTEM SEQUENCE DIAGRAMS (SSD)

SSDs are **visual summaries** of the **individual use cases**.

SSD **shows the events that external actors generate**, their order, and possible inter-system events.

All systems are treated as a black box; the diagram places emphasis on events that cross the system boundary from actors to systems.

A SSD should be done for the main success scenario of the use case and frequent or complex alternative scenarios.

A SSD should specify and show the following:

- External actors.
- Messages (methods) invoked by these actors.
- Return values (if any) associated with previous messages.
- Indication of any loops or iteration area.

USE CASE DIAGRAM OF A VENDING MACHINE

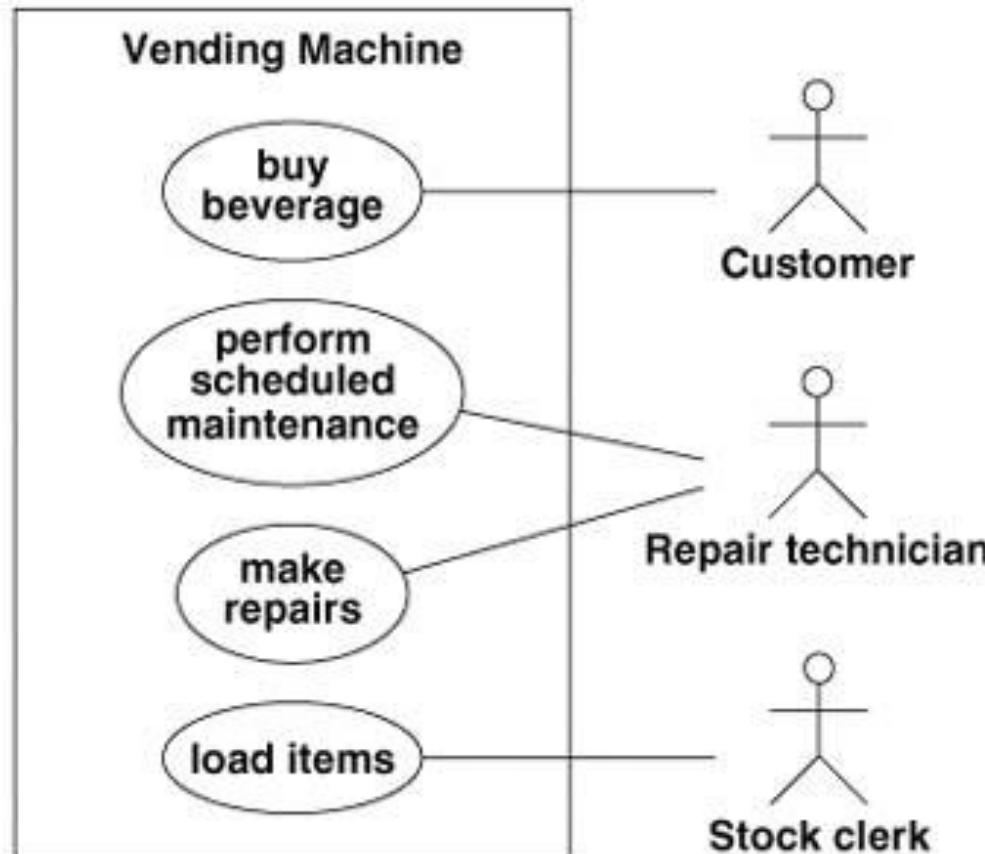


Figure 7.3 Use case diagram for a vending machine. A system involves a set of use cases and a set of actors.

VENDING MACHINE USE CASES SUMMARIES

- **Buy a beverage.** The vending machine delivers a beverage after a customer selects and pays for it.
- **Perform scheduled maintenance.** A repair technician performs the periodic service on the vending machine necessary to keep it in good working condition.
- **Make repairs.** A repair technician performs the unexpected service on the vending machine necessary to repair a problem in its operation.
- **Load items.** A stock clerk adds items into the vending machine to replenish its stock of beverages.

Figure 7.1 Use case summaries for a vending machine. A use case is a coherent piece of functionality that a system can provide by interacting with actors.

FROM USE CASE TO SYSTEM SEQUENCE DIAGRAM

Consider : “Buying a Beverage” Use Case.

Think of the vending machine (system) as a black box and draw system sequence diagram to model the interactions between the system and the actors (**system events**).

Tips:

- Prepare at least one system sequence diagram per use case.
- Prepare a system sequence diagram for each important Exception.

Use Case: Buy a beverage

Summary: The vending machine delivers a beverage after a customer selects and pays for it.

Actors: Customer

Preconditions: The machine is waiting for money to be inserted.

Description: The machine starts in the waiting state in which it displays the message "Enter coins." A customer inserts coins into the machine. The machine displays the total value of money entered and lights up the buttons for the items that can be purchased for the money inserted. The customer pushes a button. The machine dispenses the corresponding item and makes change, if the cost of the item is less than the money inserted.

Exceptions:

Canceled: If the customer presses the cancel button before an item has been selected, the customer's money is returned and the machine resets to the waiting state.

Out of stock: If the customer presses a button for an out-of-stock item, the message "That item is out of stock" is displayed. The machine continues to accept coins or a selection.

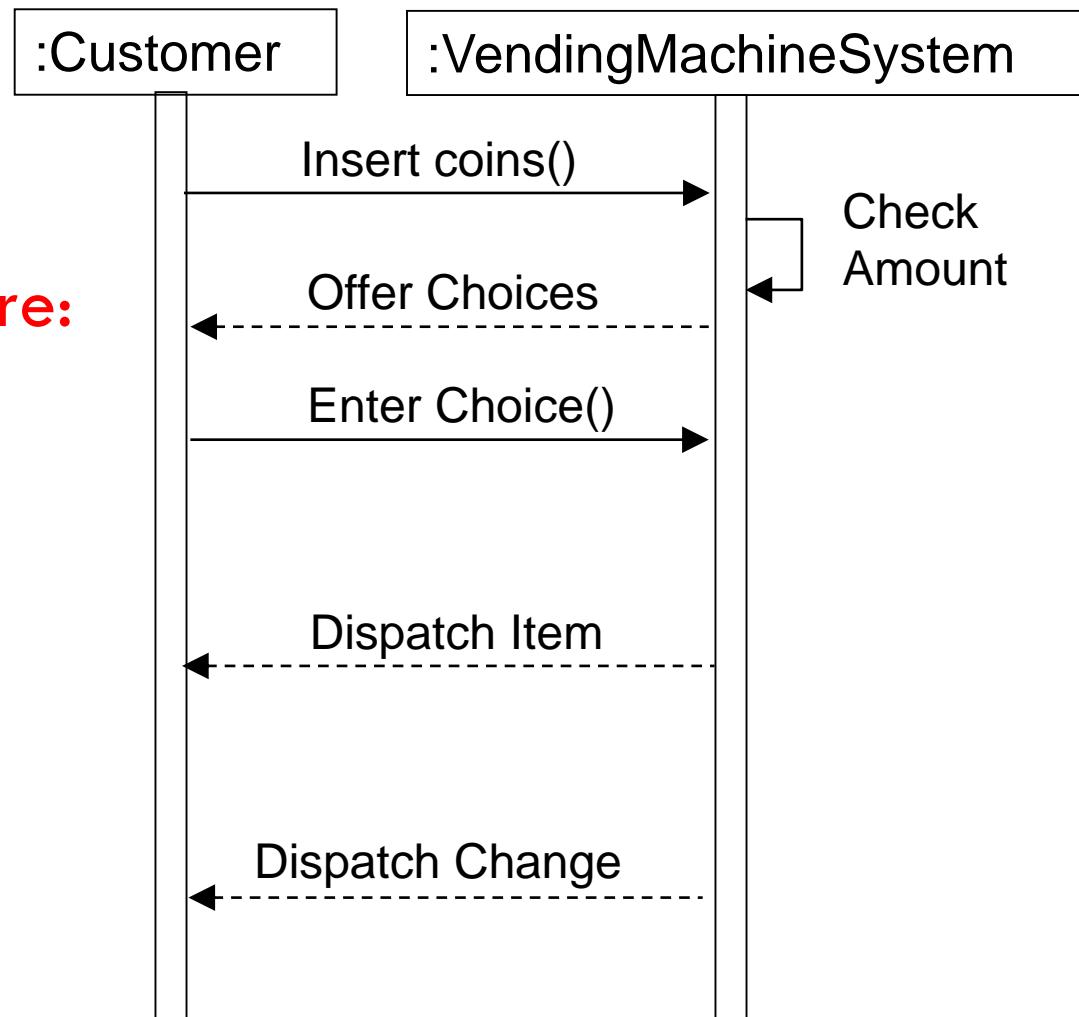
Insufficient money: If the customer presses a button for an item that costs more than the money inserted, the message "You must insert \$nn.nn more for that item" is displayed, where *nn.nn* is the amount of additional money needed. The machine continues to accept coins or a selection.

No change: If the customer has inserted enough money to buy the item but the machine cannot make the correct change, the message "Cannot make correct change" is displayed and the machine continues to accept coins or a selection.

Postconditions: The machine is waiting for money to be inserted.

Figure 7.2 Use case description. A use case brings together all of the behavior relevant to a slice of system functionality.

SYSTEM SEQUENCE DIAGRAM FOR “BUY A BEVERAGE” USE CASE



- The system events are:
 - Insert coins()
 - Enter choice()

DESIGN ISSUES: STRATEGIES FOR IMPLEMENTING USE-CASES

You have seen that when you construct interaction diagrams, the first two design decisions that have to be made are the following:

- Which message should be sent from the interface?
- To which object should the interface send the message?

Three strategies for choosing which objects are to receive messages from the user interface:

1. One Central Class
2. Actor Class
3. Use-Case class

DESIGN ISSUES: STRATEGIES FOR IMPLEMENTING USE-CASES

One Central Class: Making the interface sends all messages to a single object – usually some general object like a System or Hotel- who will in turn forward message to the concerned object. From the interface's point of view, it has to know the existence of only one object.

Advantages:

- Minimizing the knowledge the interface must have of the business model which minimizes the dependency of the user interface on the rules and concepts of the business domain.
- There is good traceability from use cases to code because every use case links to a method in the same central class (System or Hotel).

Disadvantage:

- One central class becomes overloaded with use cases.
- One possible solution to this problem is to divide the software system into several packages. Each package could still make use of one class to respond to messages from the interface.

DESIGN ISSUES: STRATEGIES FOR IMPLEMENTING USE-CASES

Actor classes: The message should be sent to the software object corresponding to the real-world actor object who initiated the operation (use actors as classes). Example: In case of check in a hotel, a person arrives to hotel initiates check in process by talking to receptionist who uses interface to do so. Therefore, the message ought to go to the Guest object.

Advantage:

- From some designers point of view this approach has clearer structure than the previous one.

Weakness:

- Traceability is more difficult than one central class.
- Limitation of reusability; For example: an Actor-Class as Guest includes methods that are related to the role of the guest in this particular application, thus reducing its usefulness in another application.
- There is a further complication when there are two actors that can initiate an interaction.

DESIGN ISSUES: STRATEGIES FOR IMPLEMENTING USE-CASES

Use cases as classes: A new class is identified for each use case, E.g.: CheckerIn, and CheckerOut. Each class have a method with a name like run, with suitable arguments. The user interface would then create a single instance of the appropriate class, initialize it suitably, & then send the run(...) message.

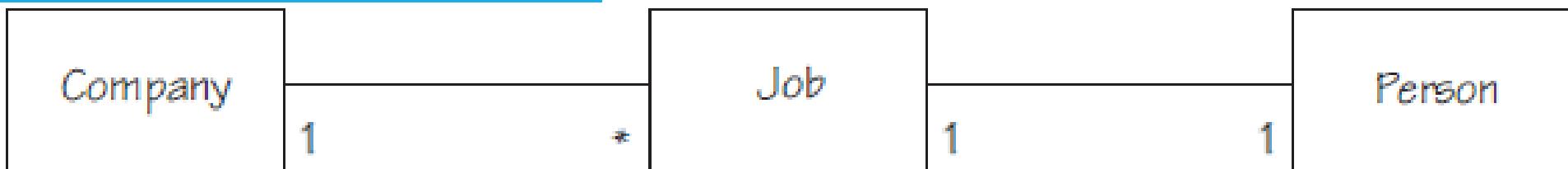
Advantages:

- Overcomes the reuse limitations posed by the strategy of actors as classes.
- Using **use case objects** lets you change or even replace the software to implement a given scenario so as to minimize the effects upon the core concepts.
- Use case objects provide traceability from each use case to a class; so each use case can be understood in isolation.

Disadvantages:

- Large number of extra classes must be defined – one for each use case.
- Many of the use case classes can be very similar, resulting in duplicated code and more difficult maintenance.

DESIGN ISSUES: FORKS & CASCADES



When designing an interaction diagram, sometimes an object needs to send a message to another object with which it has no direct associations.

- For example, in the given below figure, Company has an association with a number of instances of Job, and each Job has an association with a Person.
- A Company has no direct association with a Person.

We can implement this as either **forks** or **cascades**.

A **fork** centralizes control in the sender.

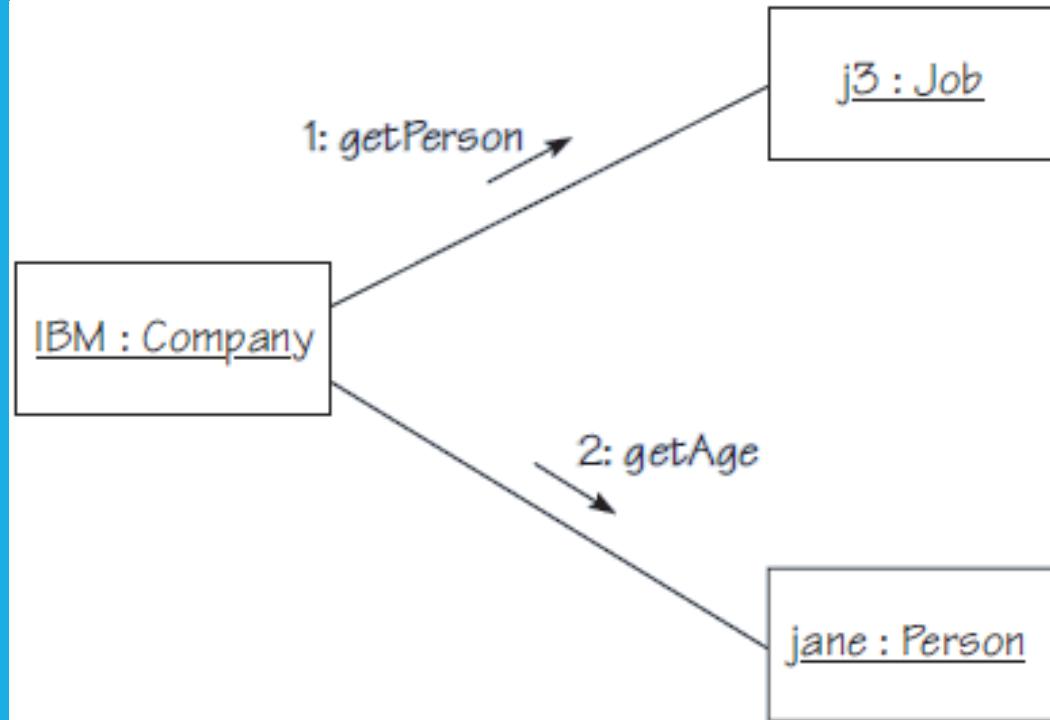
A **cascade** delegates responsibility to another object.

DESIGN ISSUES: FORKS & CASCADES

Suppose that the company needs to collect information about the ages of all its employees.

There are two ways to design this:

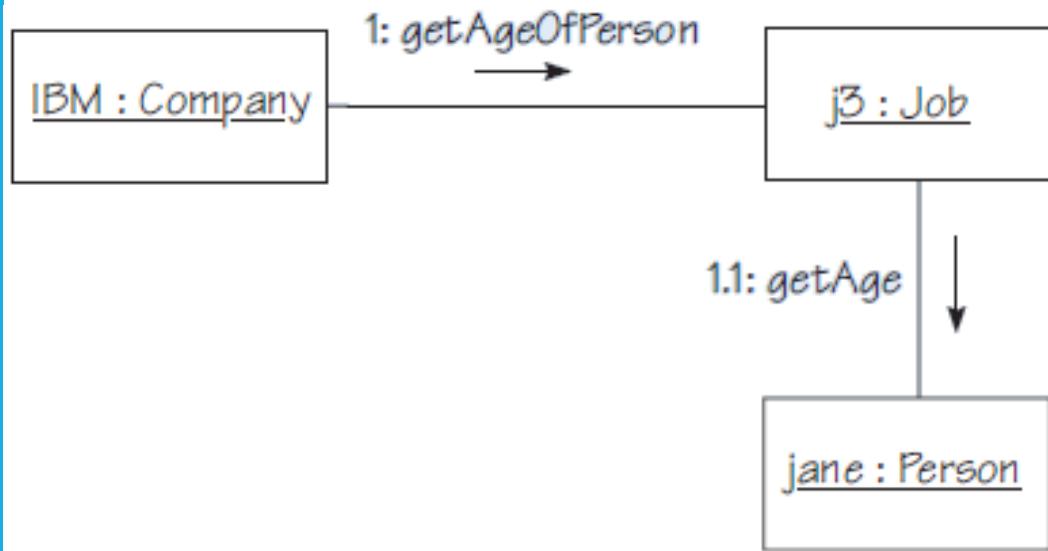
Using **fork** pattern: where Company can send a message to Job to get back person. Then send message to Person to get the age. In this pattern, the company contacts two classes directly, using the value returned from the first message as parameter for the second one.



DESIGN ISSUES: FORKS & CASCADES

Using **Cascade** pattern: Company send `getAgeofPerson()` to Job class, where Job class send another message `getAge()` to Person class. So there is no direct interaction between Company and Person.

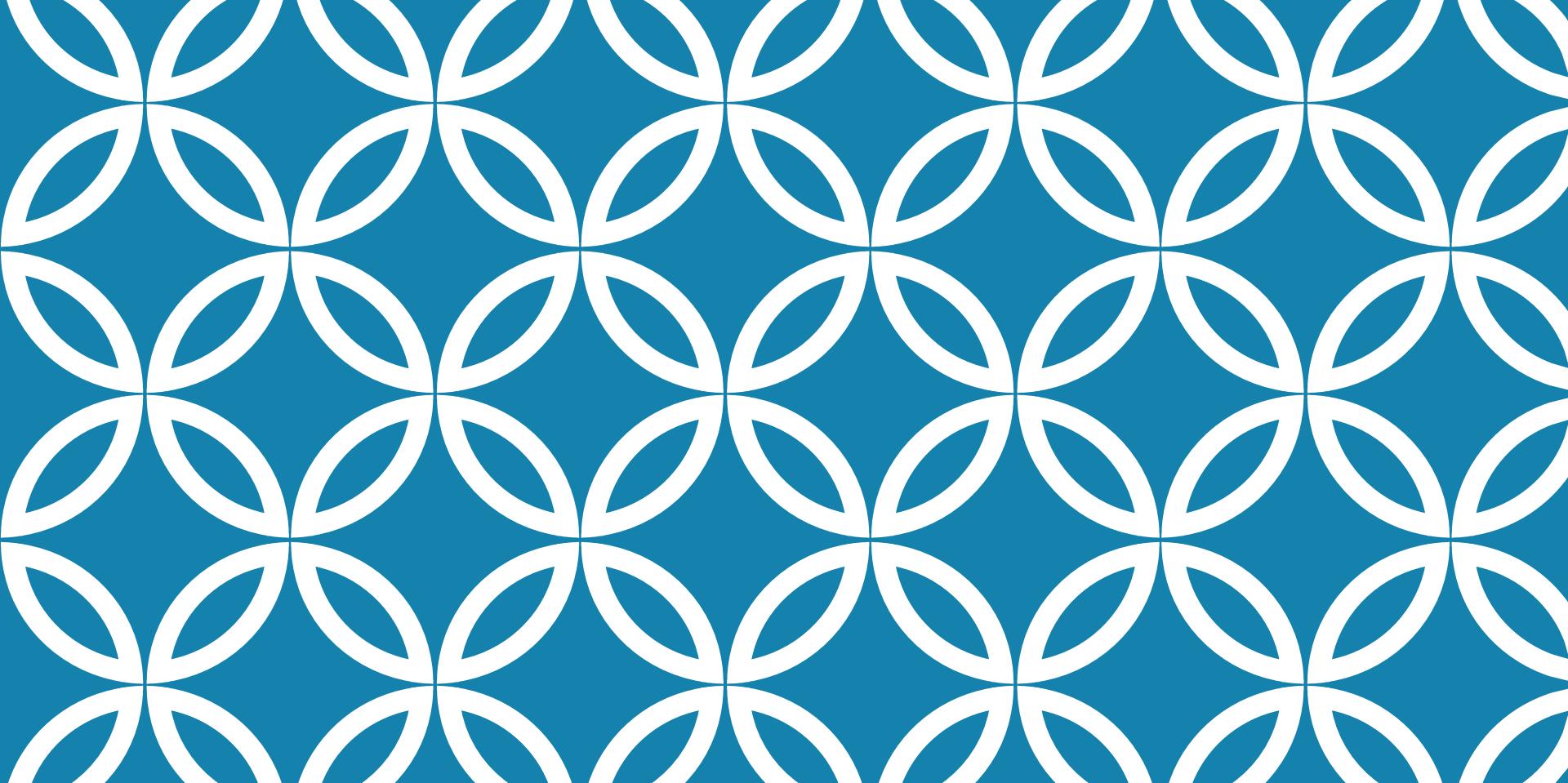
- A reasonable position is that Job should need to know only about the existence of Person, not particular attributes such as age.



THANK YOU!

Questions?





(CS251) SOFTWARE ENGINEERING I

Lecture 7
Reuse in SWE – An
Introduction to Design &
Architectural Patterns

LECTURE OBJECTIVES:

Introduce the concept of **Reuse in Software Engineering**.

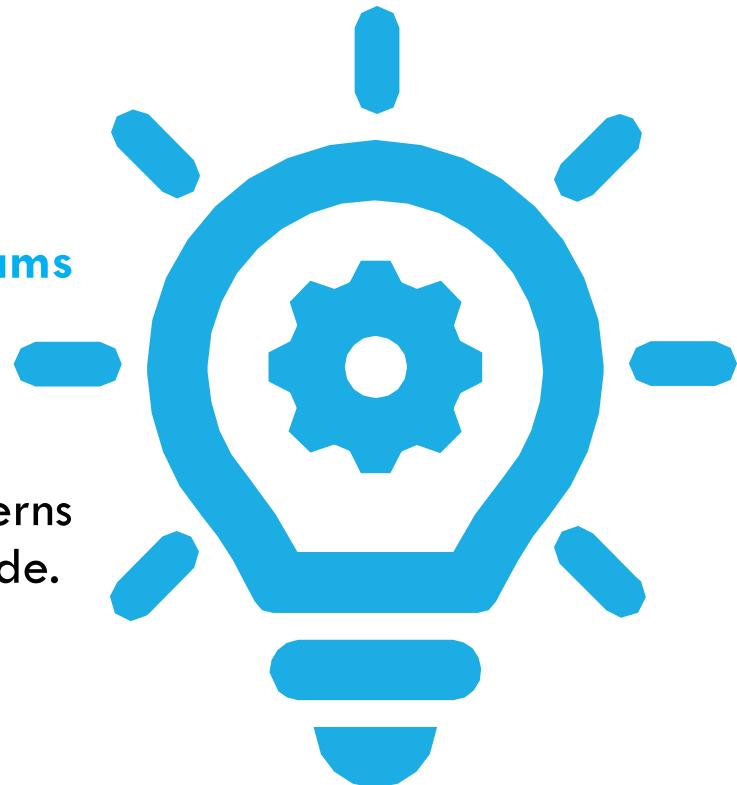
Introduce the concepts of **Architectural & Diagrams & Patterns**.

Introduce the concept of **Design Patterns**.

Discuss why architectural patterns & design patterns are important and what advantages they provide.

Discuss 10 selected architectural patterns.

Discuss 5 selected design patterns.

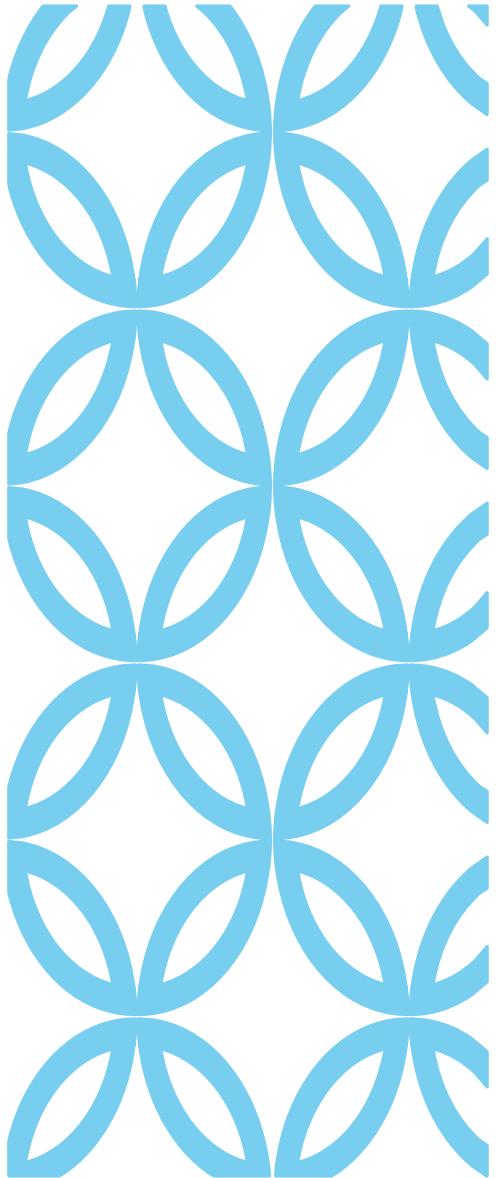


REUSE IN SOFTWARE ENGINEERING

A major aspiration of software engineering is **reuse**; *taking what you or others have done or created in the past and using it either unchanged or with relatively little adaptation.*

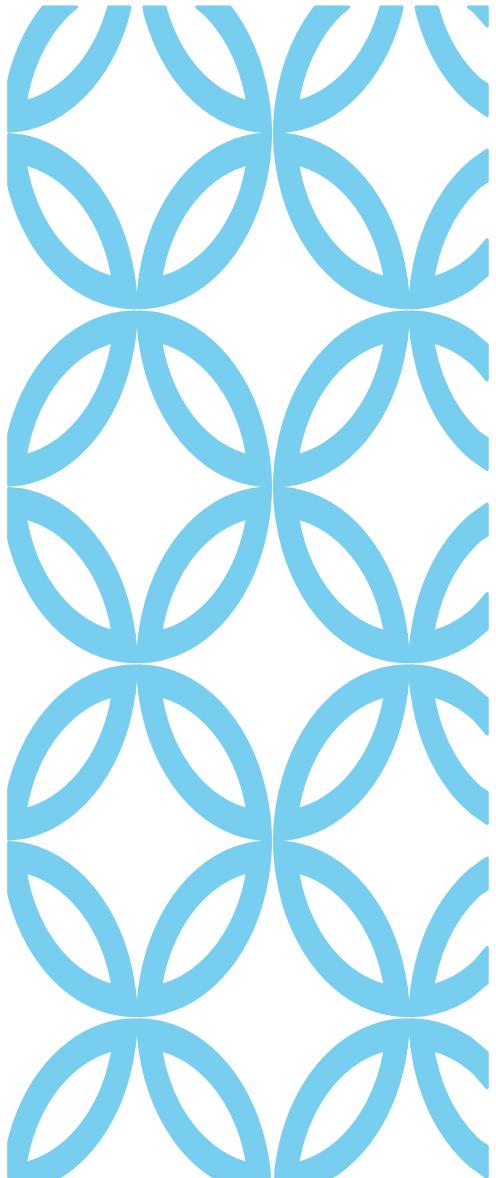
Following are some reasons why reuse is desirable:

- It **avoids duplication of effort** & reinventing already existing solutions, which saves resources.
- It **promotes reliability** because developers can use tried & trusted solutions.
- It **speeds up development** because developers can take existing solutions without starting from the beginning every time.
- It is a mechanism for **spreading good practice** among the software development community & familiarizing practitioners (developers) with tried & tested solutions.



- A ***pattern*** is the outline of a reusable solution to a recurring problem encountered in a particular context.
- Many of them have been systematically documented for all software developers to use.
- A good pattern should contain a solution that has been proven to effectively solve the problem in the indicated context.
- ***Studying patterns is an effective way to learn from the experience of others***

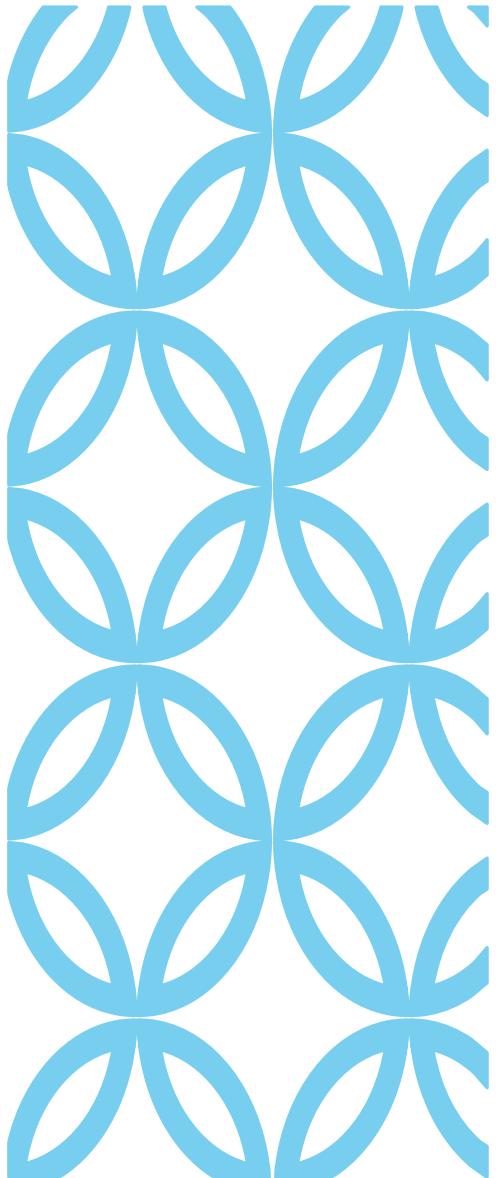
INTRODUCTION TO PATTERNS



Patterns provide **common language** between developers. For example if a developer tells another developer that he is using a **Singleton**, the other developer should know exactly what this means.

They help to **improve software quality & reduce development time**. As They provide proven solution and are based on the basic principles and heuristics of object oriented design.

MOTIVATION FOR PATTERNS

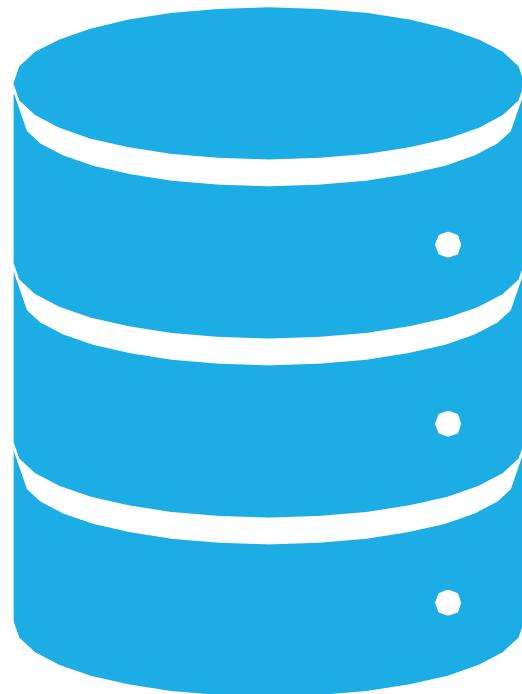


Reuse exists on several levels. Thus, Software Patterns include the following types:

- 1) Requirements Patterns
 - 2) Analysis Patterns
 - 3) **Architectural Patterns**
 - 4) Assigning responsibilities patterns
 - 5) **Design Patterns**
 - 6) Idioms
-

SOFTWARE PATTERNS

SOFTWARE ARCHITECTURAL DESIGN & — ARCHITECTURAL PATTERNS/STYLES



SOFTWARE ARCHITECTURAL DESIGN

DEFINITIONS

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.”

A **software architecture** separates the overall structure of the system, in terms of components and their interconnections, from the internal details of the individual components.

The emphasis on components and their interconnections is sometimes referred to as **programming-in-the-large**, and the detailed design of individual components is referred to as **programming-in-the-small**.

A software architecture can be described at **different levels of detail**. At a high level, it can describe the decomposition of the software system into subsystems. At a lower level, it can describe the decomposition of subsystems into modules or components. In each case, the emphasis is on the external view of the subsystem/component – that is, the interfaces it provides and requires – and its interconnections with other subsystems/components.

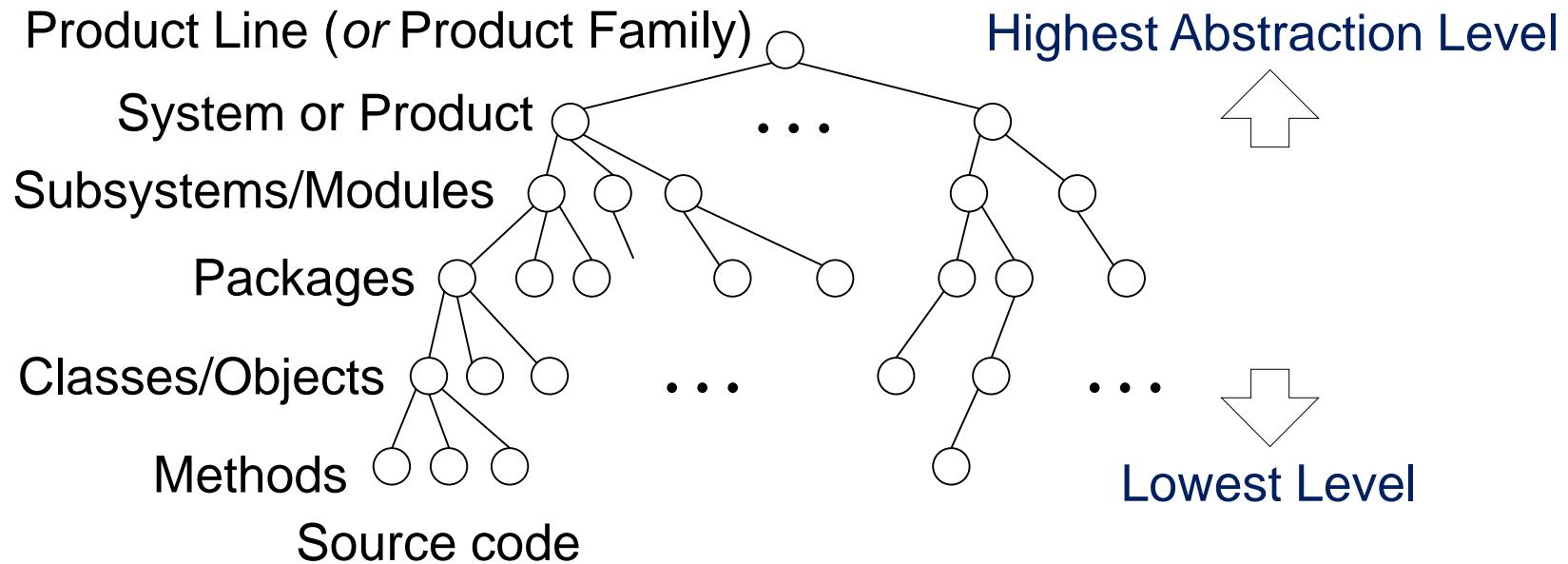
SOFTWARE ARCHITECTURAL DESIGN ..

WHY DECOMPOSE SYSTEMS?

- Tackle complexity by “divide-and-conquer”.
- See if some parts already exist & can be reused.
- Support flexibility and future evolution by decoupling unrelated parts, so each can evolve separately (“separation of concerns”).

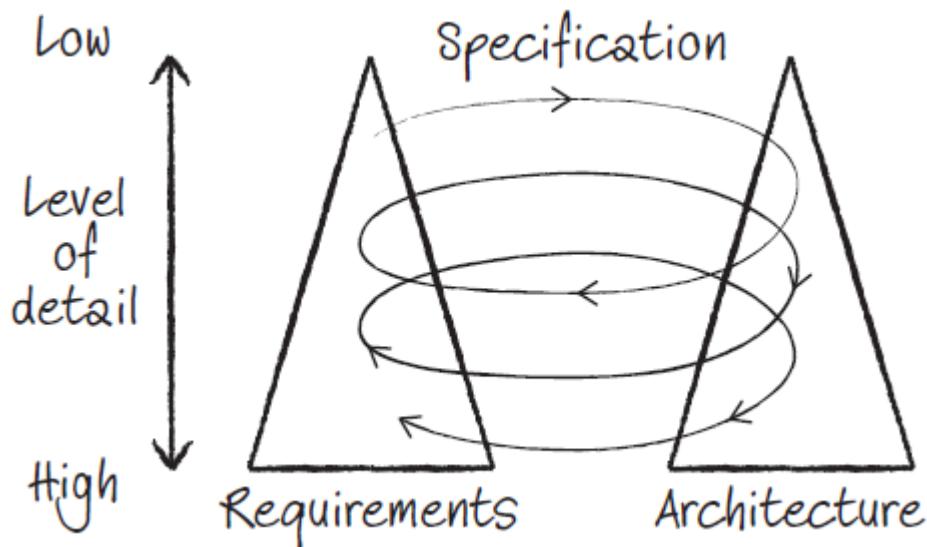
SOFTWARE ARCHITECTURAL DESIGN ..

DIFFERENT LEVELS OF DETAIL



SOFTWARE ARCHITECTURAL DESIGN .. & .. REQUIREMENTS

It is important to ensure that the architecture **fulfills** the software **requirements**, **both functional** (what the software has to do) and **nonfunctional** (how well it should do it).



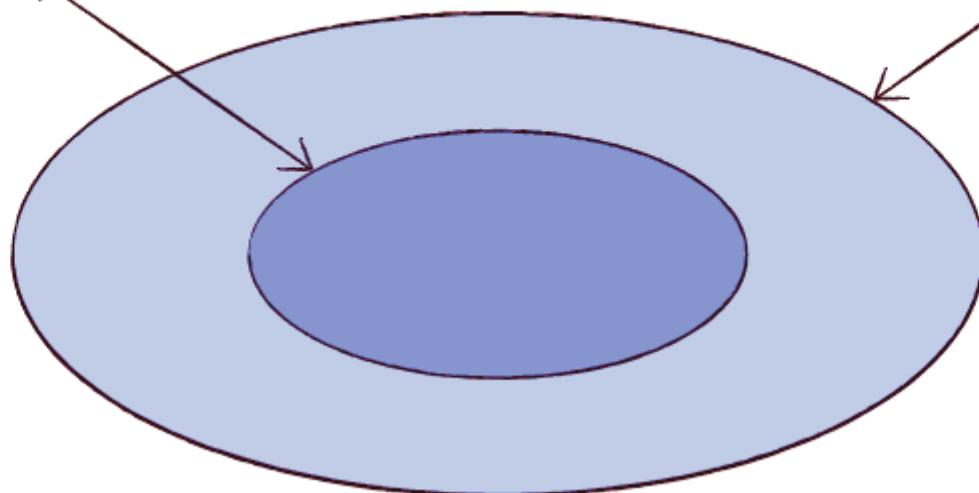
◀ Twin-peaks model (Nuseibeh, 2001), which develops requirements & architecture concurrently and iteratively.

SOFTWARE ARCHITECTURAL DESIGN .. & .. FUNCTIONAL VS. NON-FUNCTIONAL REQ.

There can be multiple architectures that meet functional requirements but they **will not all be equal when it comes to non-functional requirements**. Moreover, not all non-functional requirements will be of equal importance when making architectural decisions.

Architectures that also meet non-functional requirements

Architectures that meet functional requirements



SOFTWARE ARCHITECTURAL DESIGN .. & .. NON-FUNCTIONAL REQUIREMENTS

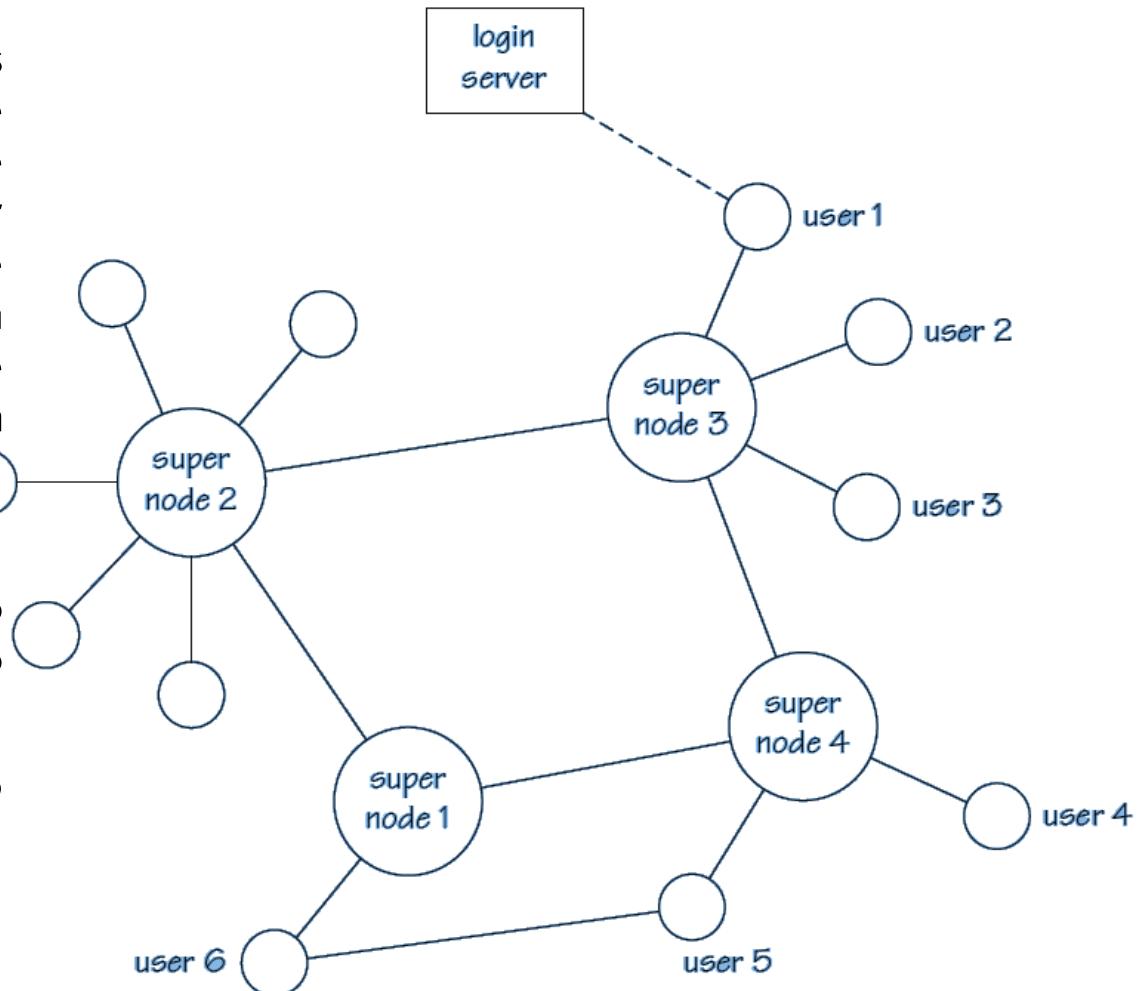
The software **quality attributes of a system should be considered** when developing the software architecture. These attributes relate to how the architecture **addresses important nonfunctional requirements**, such as performance, security, and maintainability.

Example: How **Skype's architecture** is affected by a non-functional requirement (i.e., Reliability)

- Skype is a well known internet telephony service.
- To go online, an individual user must first contact a login server, and it then connect to the nearest super node.
- The super-nodes then route telephone calls from one user to another, yet it's also possible for two users to be connected directly.

SOFTWARE ARCHITECTURAL DESIGN .. & .. NON-FUNCTIONAL REQUIREMENTS

- Originally, all the nodes except the login server were equivalent. Any suitable node could be promoted to a super node and become responsible for routing calls (a style known as **peer-to-peer "P2P"** where there is no clear distinction between clients and servers).



SOFTWARE ARCHITECTURAL DESIGN

ARCHITECTURAL PATTERNS/STYLES

Software architectural patterns provide the **skeleton or template for the overall software architecture** or high-level design of an application.

Architectural styles or patterns of software architecture: are recurring architectures used in a variety of software applications. These include such widely used architectures as client/server and layered architectures.

Software architectural patterns can be grouped into two main categories: (1) architectural structure patterns, which address the static structure of the architecture, and (2) architectural communication patterns, which address the dynamic communication among distributed components of the architecture.

SOFTWARE ARCHITECTURAL DESIGN

ARCHITECTURAL PATTERNS/STYLES

Commonly employed architectural styles include:

- Client–Server
- Call-Return
- Layered
- Peer-to-Peer
- Data-flow
- Data-Centred
- Independent Components
- Service-Oriented
- Notification
- Model-View-Controller

Important Note: there are relationships between these different styles and some overlap with others.

SOFTWARE ARCHITECTURAL STYLES

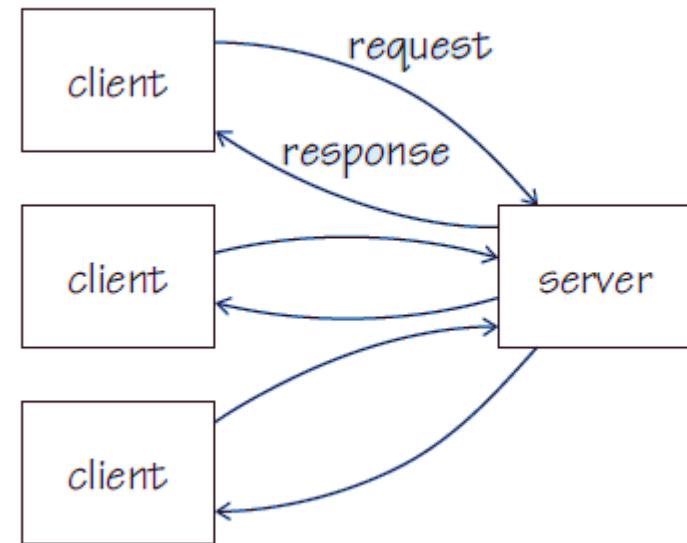
CLIENT-SERVER ARCHITECTURE

Probably the best known of all architectural styles.

One component (the Server) provides a service to the other components (the Clients).

A server waits for requests from clients, processes them when received, and returns a response to the client.

Examples: requests from web-browsers to web-servers.

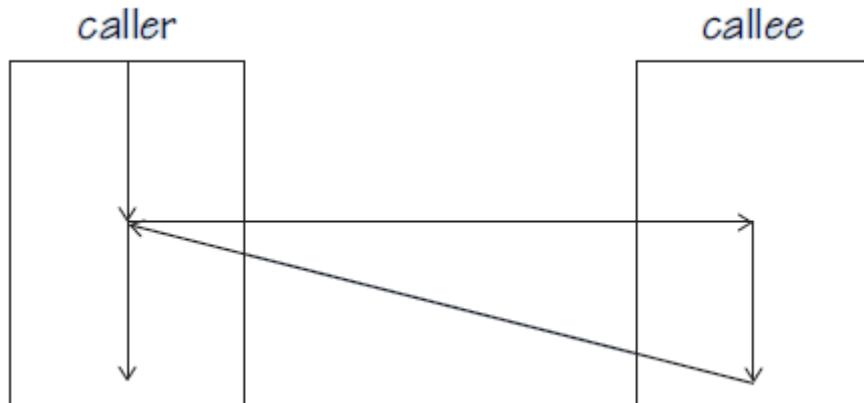


SOFTWARE ARCHITECTURAL STYLES

CALL-RETURN ARCHITECTURE

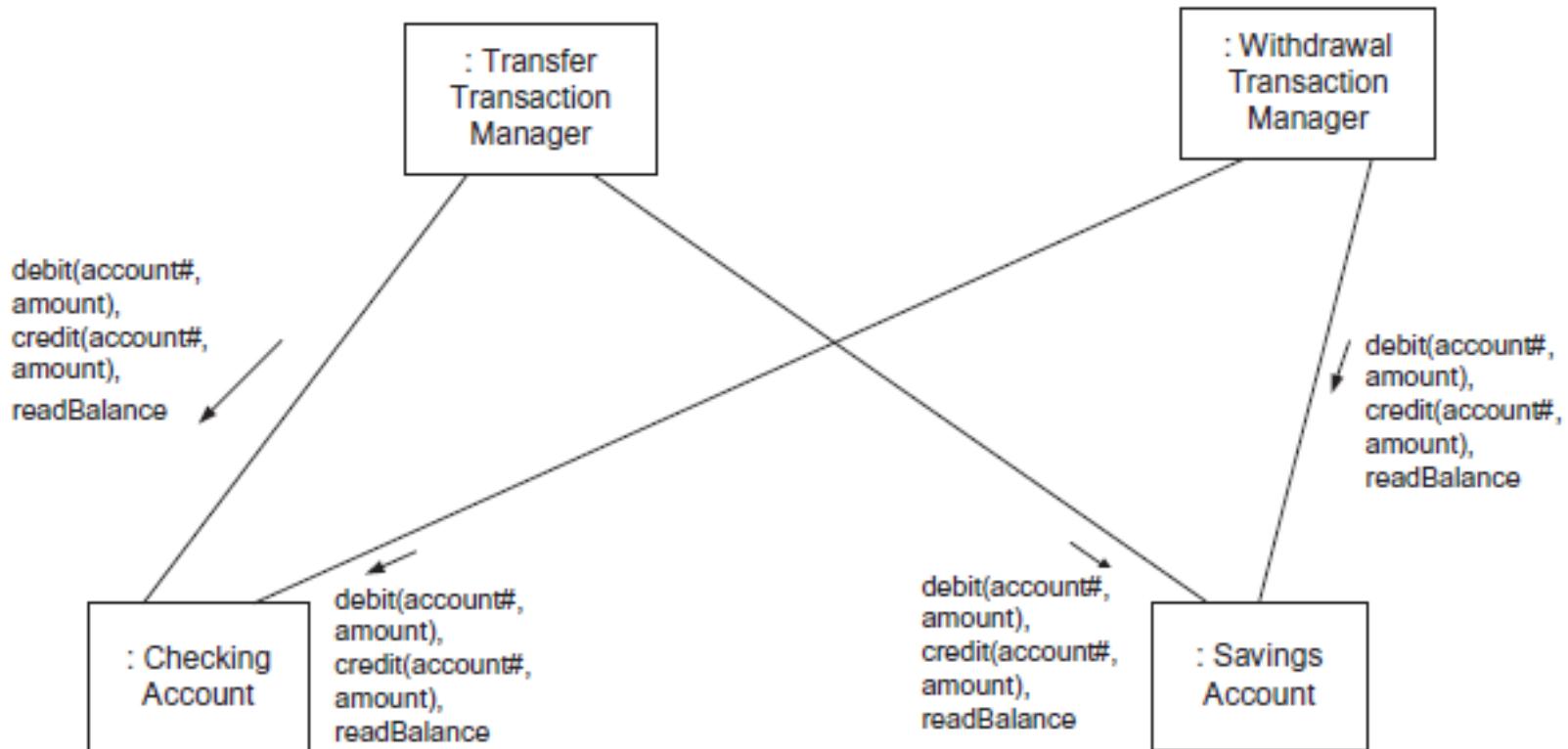
A component (the Caller) makes a procedure call to another component (the Callee) and waits for the call to return.

Examples: Main program calls to a subprogram (in a traditional/structural software) or methods invocation (in OOP).



SOFTWARE ARCHITECTURAL STYLES

CALL-RETURN ARCHITECTURE



SOFTWARE ARCHITECTURAL STYLES

LAYERED (TIERED) ARCHITECTURE

The system is structured as a series of layers that are visualized as stacked on top of another.

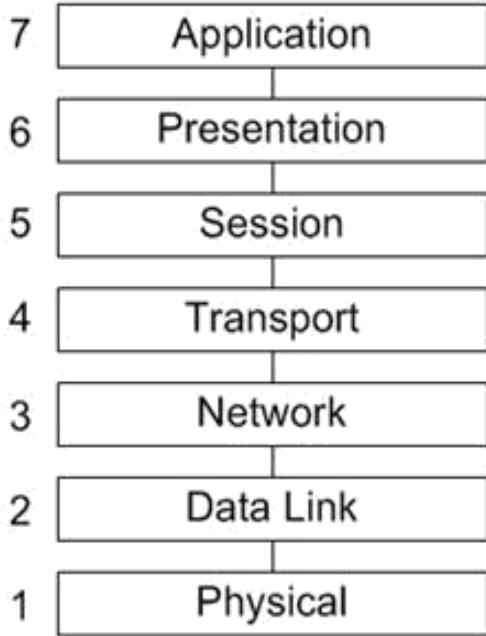
Each layer uses services provided by the layers below (usually just by the layer immediately below), and it supplies services to the layer above.

Examples:

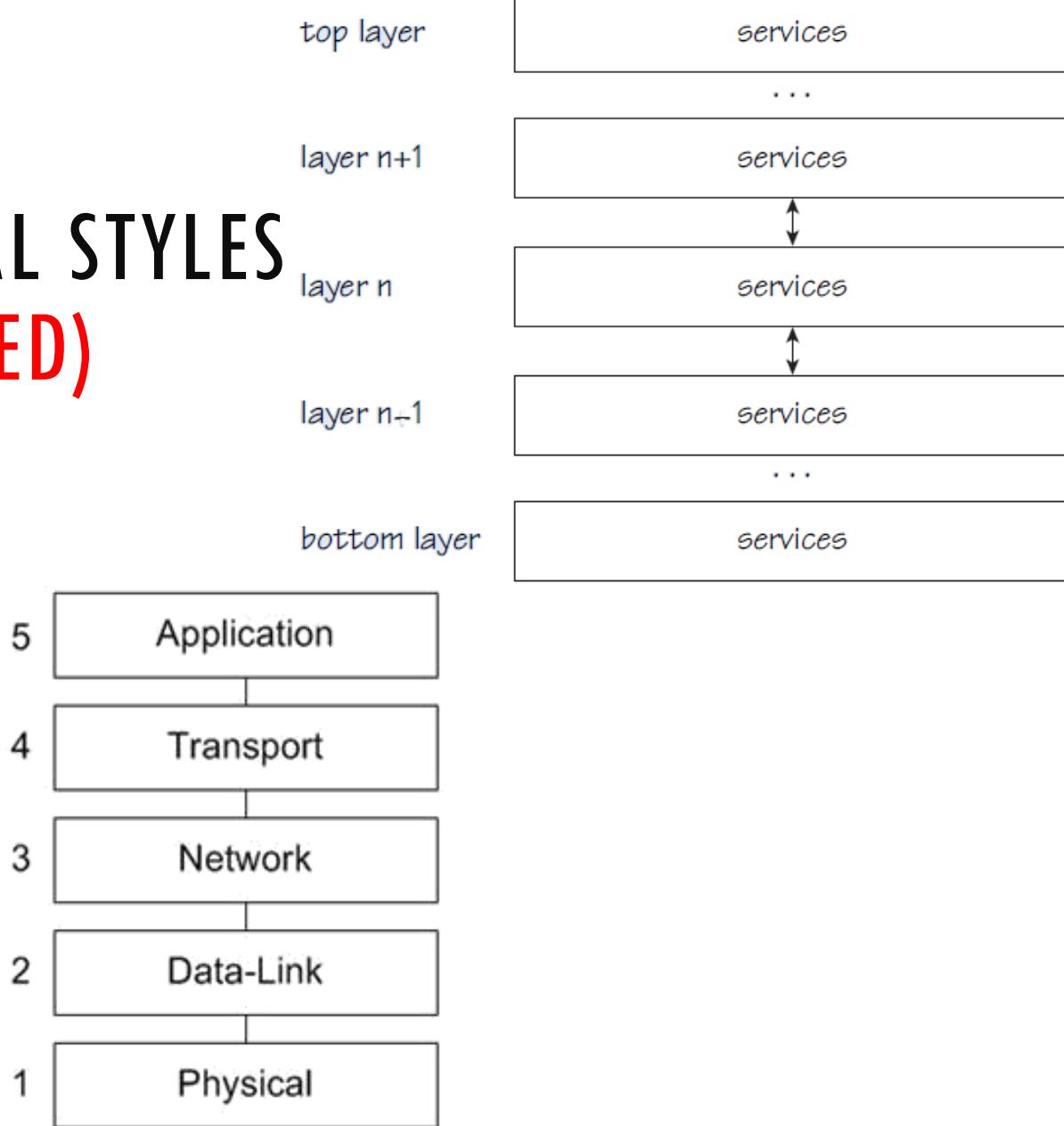
- A compiled Java program, which executes in a Java Virtual Machine that in turn makes calls to services supplied by the Operating system.
- Open Systems Interconnection (OSI) & Transmission Control Protocol/Internet Protocol (TCP/IP).
- Client-Server Architecture is a special case in which there are just 2 layers.

SOFTWARE ARCHITECTURAL STYLES

LAYERED (TIERED) ARCHITECTURE

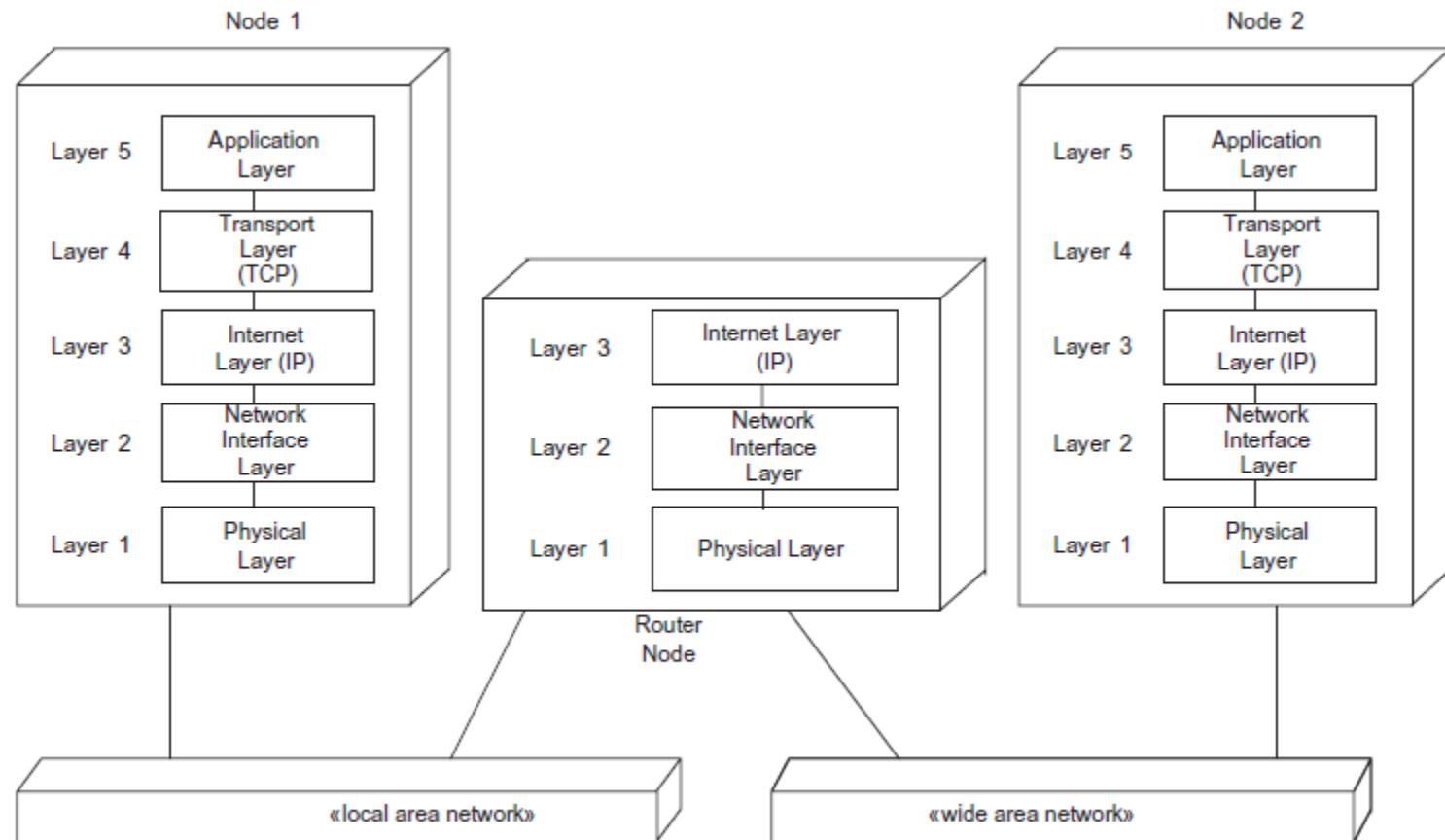


TCP/IP Model



SOFTWARE ARCHITECTURAL STYLES

LAYERED (TIERED) ARCHITECTURE



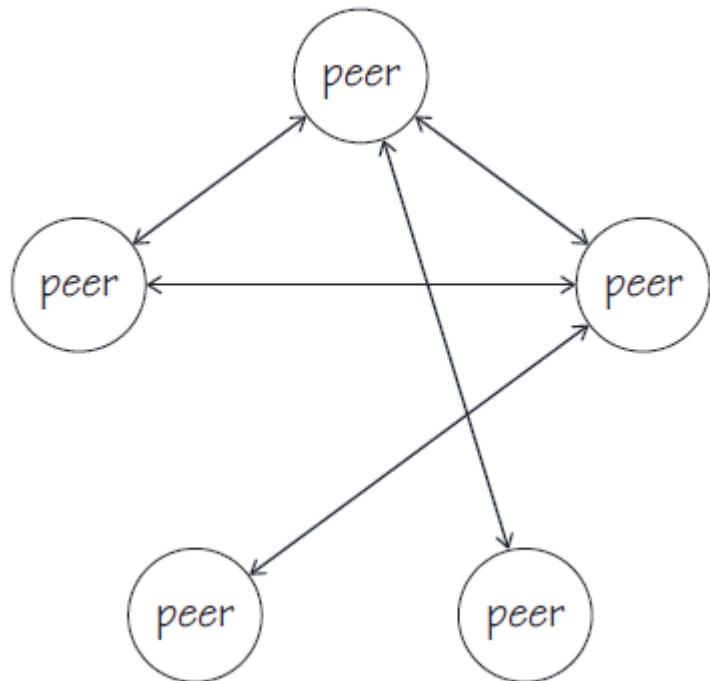
Layers of Abstraction architectural pattern: Internet communication with TCP/IP

SOFTWARE ARCHITECTURAL STYLES

PEER-TO-PEER ARCHITECTURE

Peer-to-Peer resembles the Client-Server style except that all the components are both clients and servers and any component can request services from any other component.

Example: a streaming music service, where listeners generally get streamed tracks from the nearest peer than can be located by sending a request that hops from one peer to another until the desired track is located.



SOFTWARE ARCHITECTURAL STYLES

DATA-FLOW (PIPES & FILTERS) ARCHITECTURE

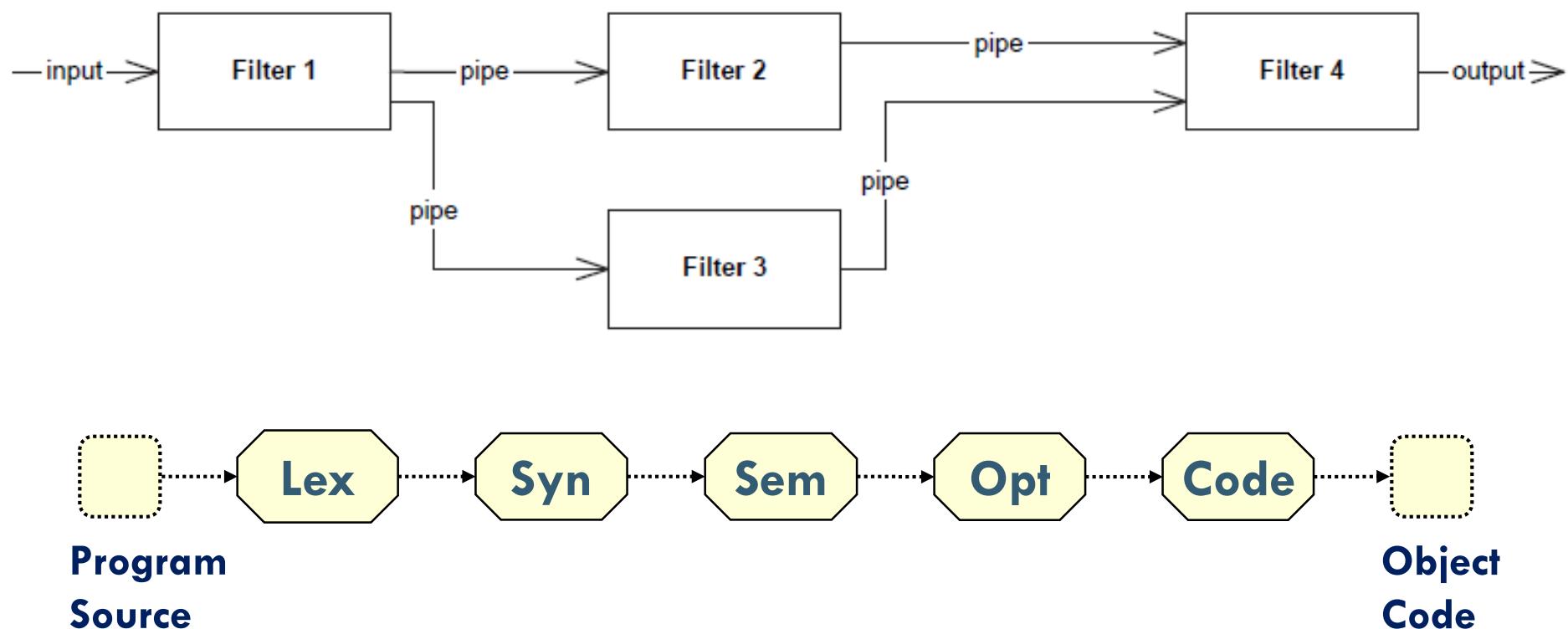
The data-flow style components are objects or small independent subprograms (filters) that process a stream of data and pass the results on to other components for further processing.

Communication is unidirectional and uses fixed channels, and each filter has no knowledge of other filters upstream or downstream (simply accepts data, processes it, and then passes it on).

Example: A Pipelined Compiler Architecture.

SOFTWARE ARCHITECTURAL STYLES

DATA-FLOW (PIPES & FILTERS) ARCHITECTURE



SOFTWARE ARCHITECTURAL STYLES

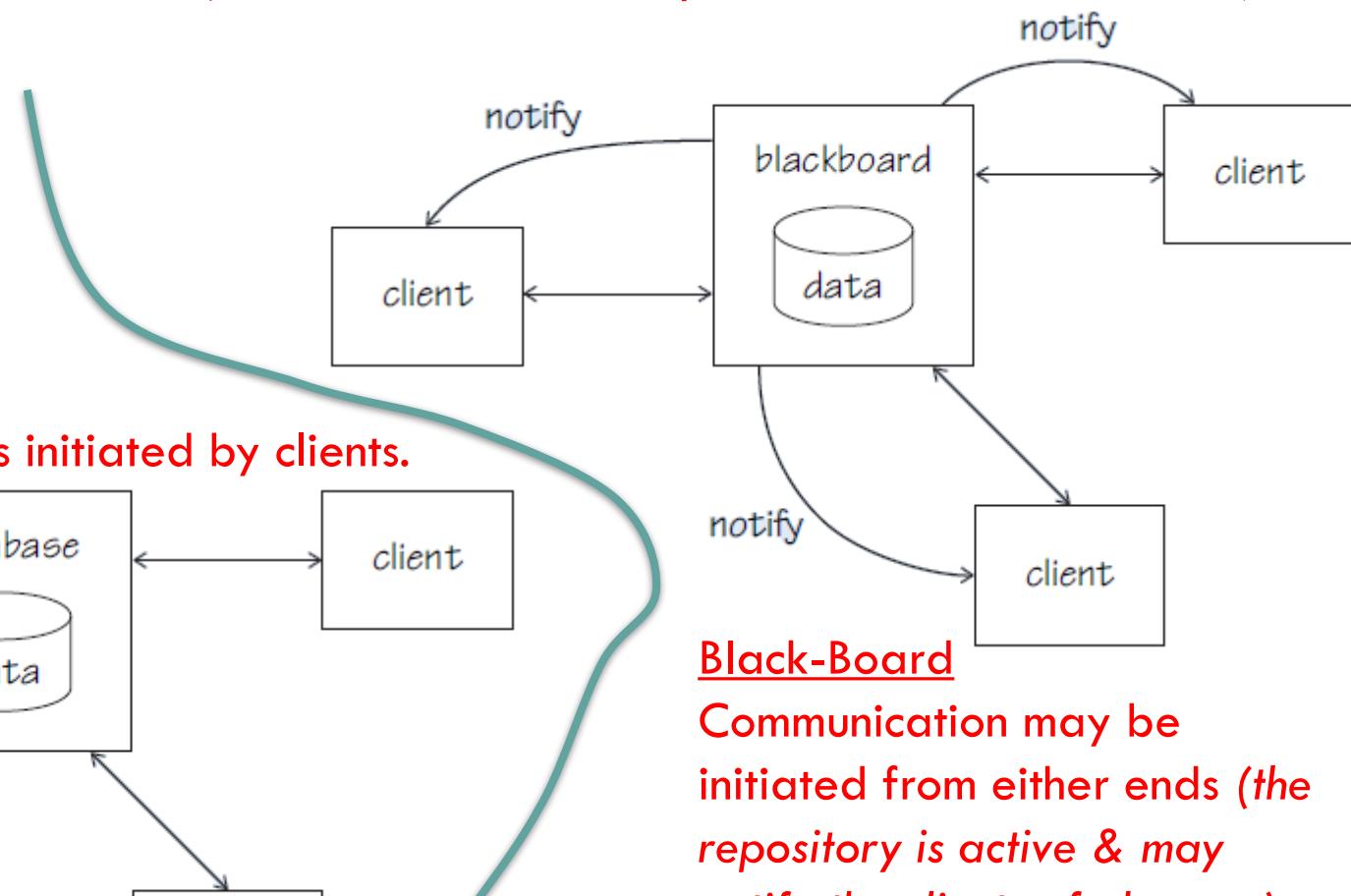
DATA-CENTRED (REPOSITORY / BLACK-BOARD)

- There is a data provider that is a centralized store of persistent data.
- The structure of the data, the types of items & their relationships are stable (change rarely or not at all).
- Many Clients (who are data consumers) can have items created, queried, updated, & destroyed upon request.
- The central store may be duplicated to provide backup (in case of failure) or deal with a greater volume of requests.

Example: A database holding personnel records with authorized users being able to log on and submit queries.

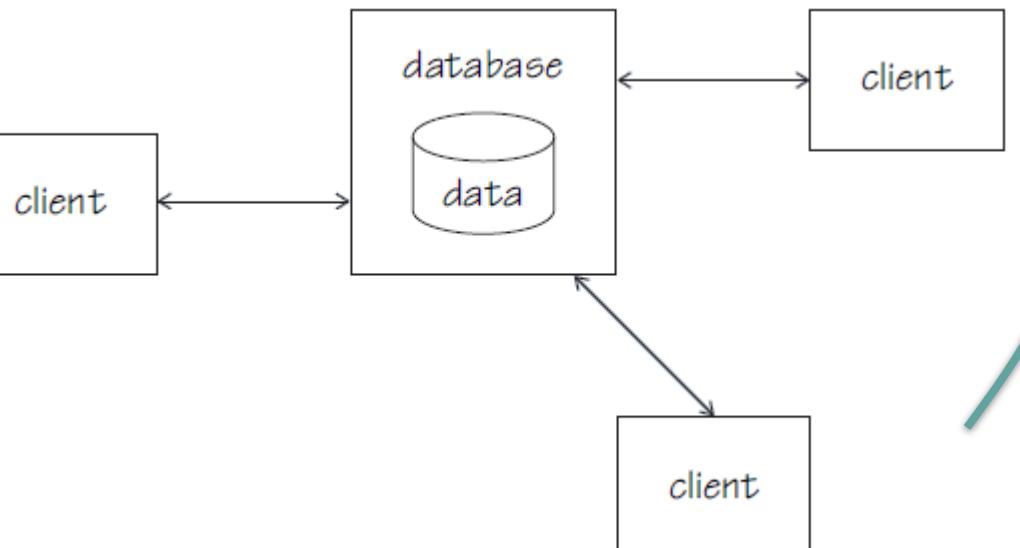
SOFTWARE ARCHITECTURAL STYLES

DATA-CENTRED (REPOSITORY / BLACK-BOARD)



Database (Repository)

Communication is always initiated by clients.



Black-Board

Communication may be initiated from either ends (*the repository is active & may notify the clients of changes*).

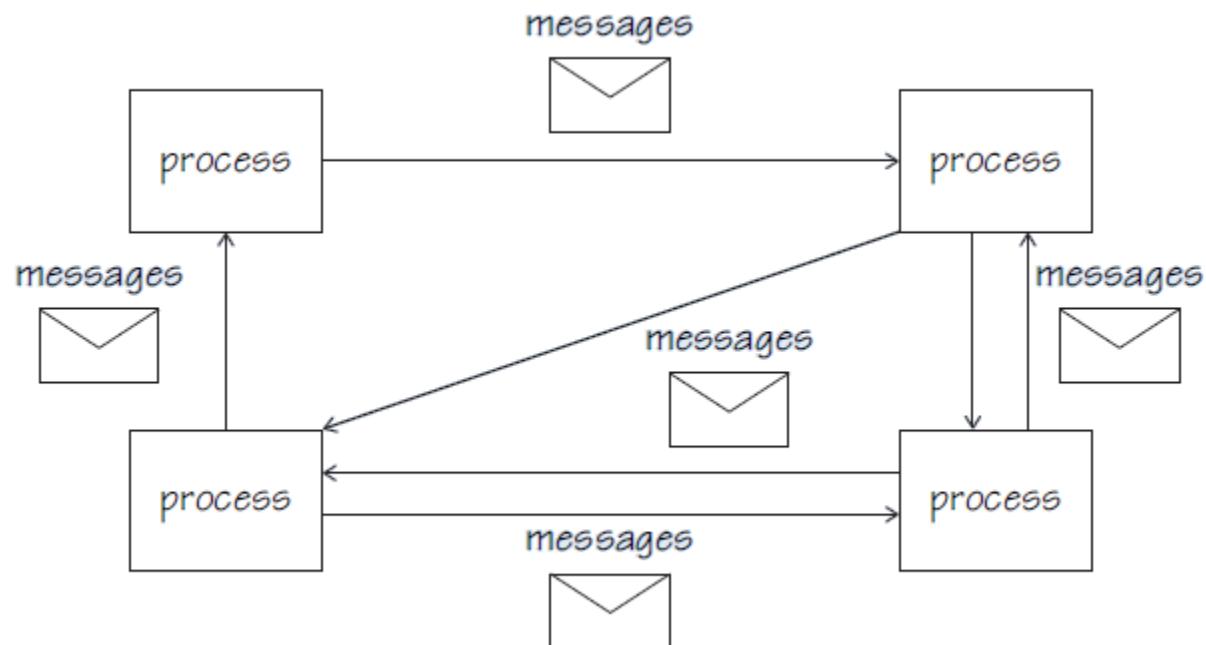
SOFTWARE ARCHITECTURAL STYLES

INDEPENDENT COMPONENTS ARCHITECTURE

Components execute concurrently and are decoupled as far as possible, but can communicate by messages that allow them to exchange data or coordinate their operations.

Example:

Concurrent/Real-time Systems .. A set of components controlling different parts of a chemical processing plant, independently regulating the part each is responsible for, but sharing data and coordinating with one another by exchanging messages.

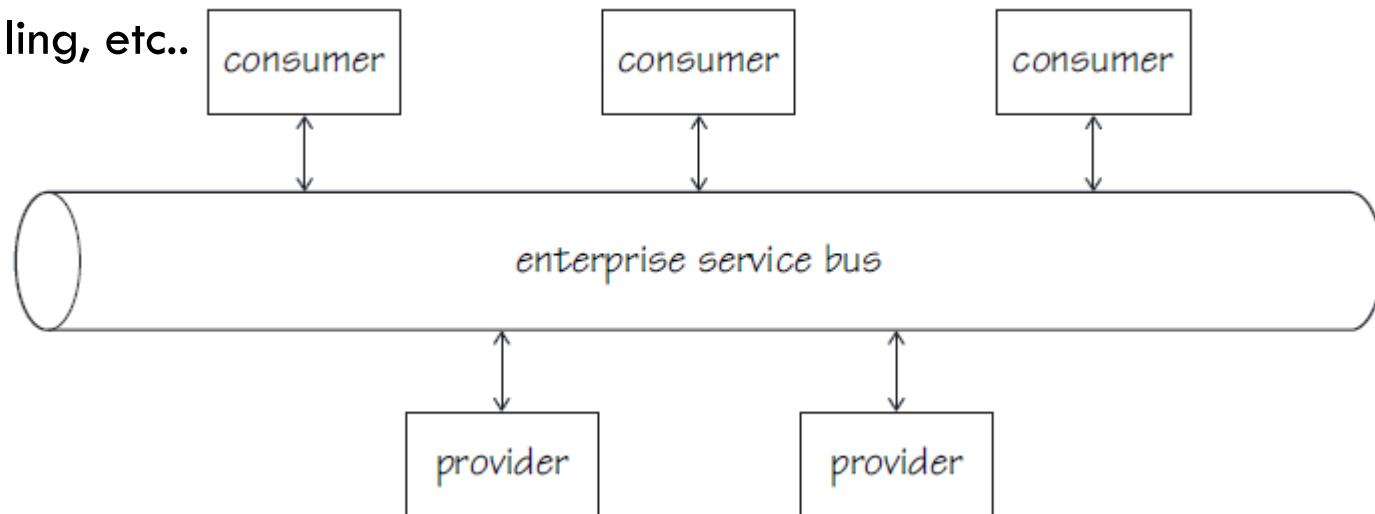


SOFTWARE ARCHITECTURAL STYLES

SERVICE-ORIENTED ARCHITECTURE

Two kinds of components; **consumers** & **providers**. A set of service providers makes services available to a set of service consumers. Consumers can combine services to carry out the business processes they require.

Example: Different divisions of an organization whose systems all use a common set of services such as payroll, personnel, customer records, management billing, etc..

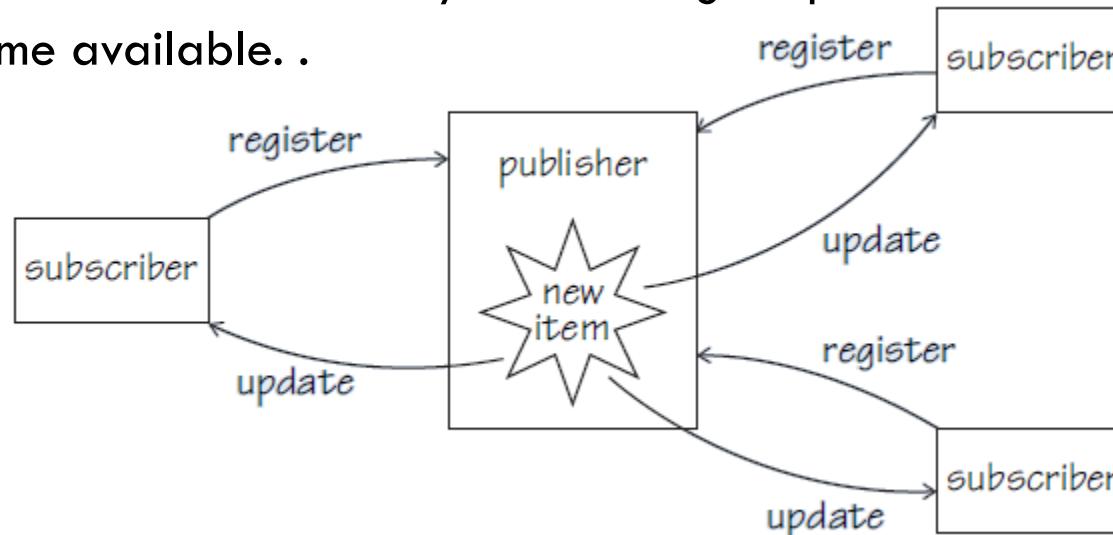


SOFTWARE ARCHITECTURAL STYLES

NOTIFICATION / IMPLICIT-INVOCATION / PUBLISH-SUBSCRIBE ARCHITECTURE

Two kinds of components; **observers** & **subjects**. Observers/subscribers can register themselves with a subject/publisher to be kept notified whenever some particular event happens at the subject's end.

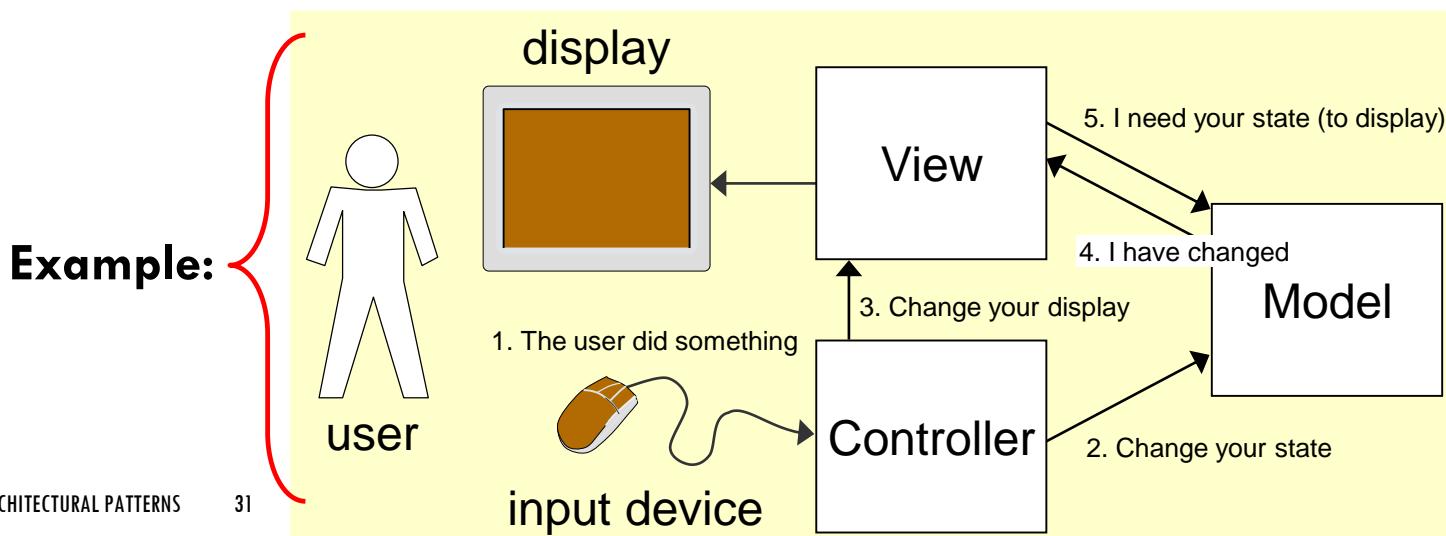
Example: RSS feed where anyone who signs up receives news updates as they become available. .



SOFTWARE ARCHITECTURAL STYLES

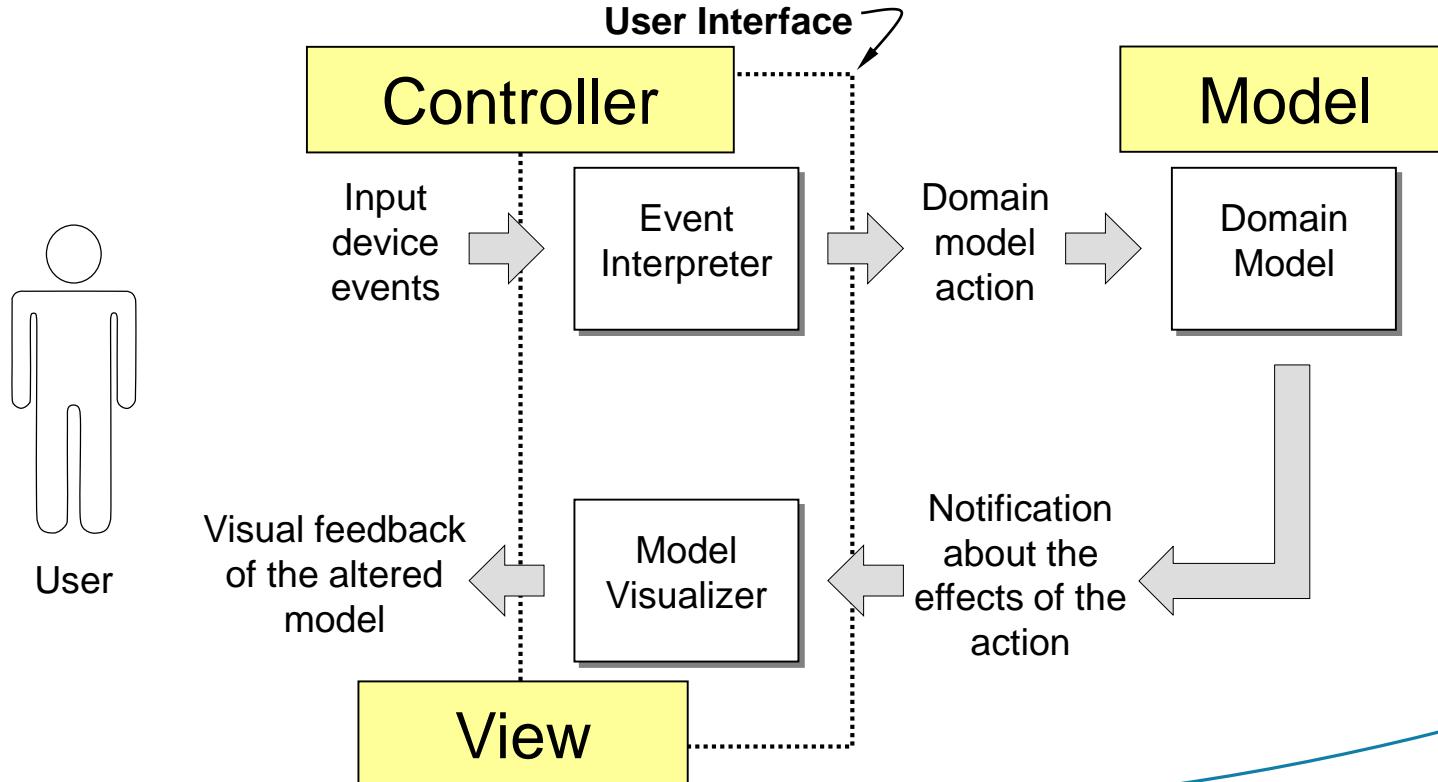
MODEL-VIEW-CONTROLLER ARCHITECTURE

- **Model:** Holds all the data, state and application logic. Oblivious to the View and Controller. Provides API to retrieve state and send notifications of state changes to “observer”.
- **View:** Gives user a presentation of the Model. Gets data directly from the Model.
- **Controller:** Takes user input and figures out what it means to the Model.



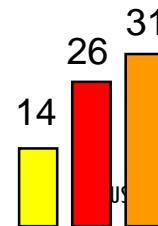
SOFTWARE ARCHITECTURAL STYLES

MODEL-VIEW-CONTROLLER ARCHITECTURE

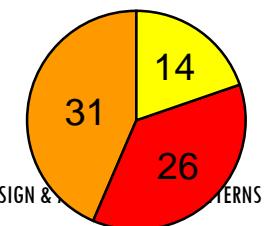


Model: array of numbers [14, 26, 31]

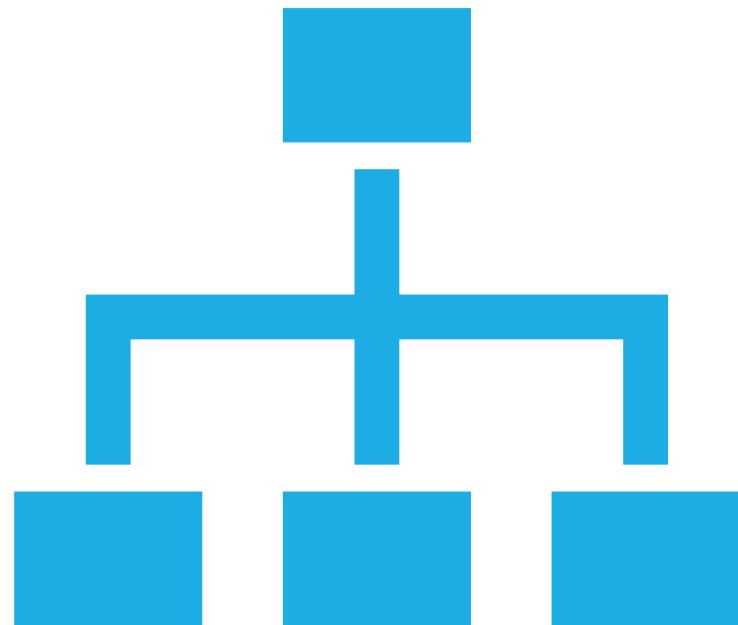
→ Different Views for the same Model:



versus



SOFTWARE DESIGN PATTERNS



DESIGN PATTERNS .. WHAT ARE THEY?

A Design pattern is a **solution** to a **recurring design problem** within a particular **context**.

Design Pattern = Design Problem & Solution pair in a context

Design Patterns descriptions are often independent of programming languages and implementation details.

DESIGN PATTERNS × CLASS LIBRARIES

Class Libraries :

- Reusable **Code**; ex. String class, Math class,...
- Language **dependent**.

Design Patterns :

- Reusable **Design**; Design problem & solution
- Language **independent**.

DESIGN PATTERNS' DESCRIPTION

Four main elements to describe any pattern:

- The name of the pattern
- The purpose of the pattern: what problem it solves
- How to solve the problem
- The constraints we have to consider in our solution

DESIGN PATTERN DESCRIPTION (TEMPLATE 1)

Name:

Context:

- The general situation in which the pattern applies

Problem:

- A short sentence or two raising the main difficulty.

Forces:

- The issues or concerns to consider when solving the problem

Solution:

- The recommended way to solve the problem in the given context.

Antipatterns: (Optional)

- Solutions that are inferior or do not work in this context.

Related patterns: (Optional)

- Patterns that are similar to this pattern.

References:

- Who developed or inspired the pattern.

DESIGN PATTERN DESCRIPTION (GOF TEMPLATE)

Name:

Intent:

- The general situation in which the pattern applies

Problem:

- A short sentence or two raising the main difficulty.

Solution:

- The approach to solve the problem .

Structure:

- Class diagram

Participants:

- Entities involved in the pattern.

Consequences:

- Effect the pattern has on your system

Implementation:

- Example ways to implement the pattern.

DESIGN PATTERNS' CATEGORIZATION

Patterns are classified into three categories

- **Creational Design Patterns**
 - Concerned with the creation of objects.
- **Structural Design Patterns**
 - Concerned with the composition of classes or objects into larger structures.
- **Behavioral Design Patterns**
 - Determines the ways in which the classes interact and distribute responsibilities (determines the flow of control in a complex program).

EXAMPLES - GOF PATTERNS

Creational

- Abstract Factory
- Factory Method
- Builder
- Prototype
- Singleton

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioural

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

SELECTED DESIGN PATTERNS

CREATIONAL ..

- ***Singleton Pattern***
 - *Ensures that one instance of a class will be created.*
- ***Immutable Pattern***
 - *Ensures that the state of the object never changes after creation.*

SELECTED DESIGN PATTERNS

CREATIONAL: SINGLETON PATTERN

- **Context:**

- It is very common to find classes for which only one instance should exist (*singleton*) .
- Examples: Company or university class, Main Window class in GUI.

- **Problem:**

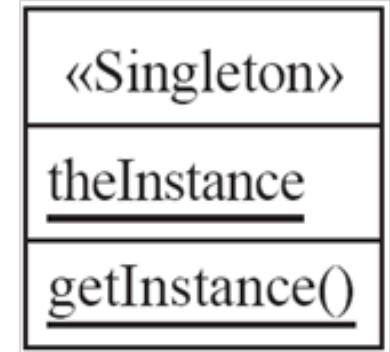
- How do you ensure that it is never possible to create more than one instance of a singleton class. And provide a global point of access to it.

- **Forces:**

- The use of a public constructor cannot guarantee that no more than one instance will be created.
- The singleton instance must also be accessible to all classes that require it, therefore it must often be public.

SELECTED DESIGN PATTERNS

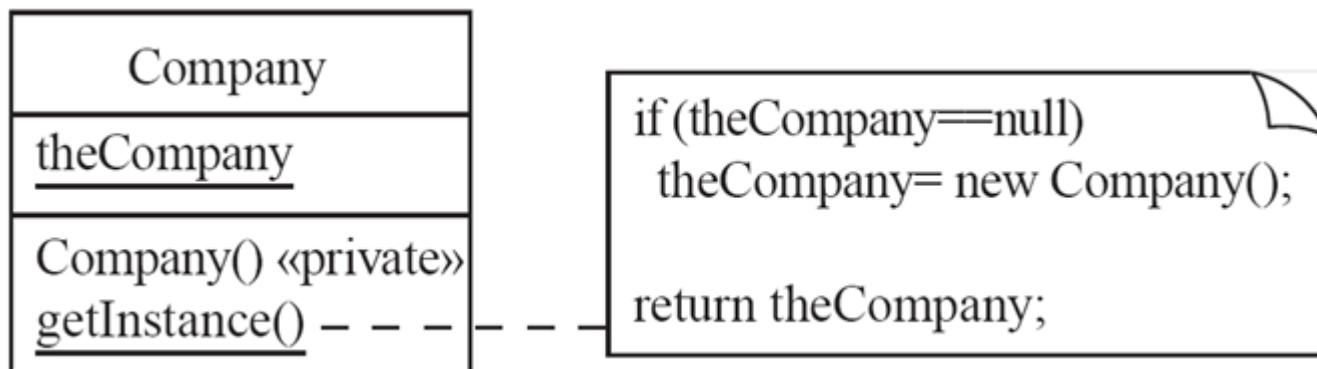
CREATIONAL: SINGLETON PATTERN



▪ **Solution:**

- Have the constructor private to ensure that no other class will be able to create an instance of the class singleton.
- Define a public static method, The first time this method is called , it creates the single instance of the class “singleton” and stores a reference to that object in a static private variable.

▪ **Example:**



SELECTED DESIGN PATTERNS

CREATIONAL: IMMUTABLE PATTERN

- **Context:**
 - An immutable object is an object that has a state that never changes after creation.
- **Problem:**
 - How do you create a class whose instances are immutable?
- **Forces:**
 - There must be no loopholes that would allow ‘illegal’ modification of an immutable object.
- **Solution:**
 - Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified.
 - Instance methods which access properties must not change instance variables.

SELECTED DESIGN PATTERNS

STRUCTURAL ..

- *Delegation Pattern*
 - For reusing methods.
- *Abstraction-Occurrence Pattern*
 - Employed in domains model where you find a set of related objects (occurrences), the members of such a set share common information but also differ from each other in important ways.

SELECTED DESIGN PATTERNS

STRUCTURAL: DELEGATION PATTERN

- **Context:**

- You are designing a method in a class. You realize that another class has a method which provides the required service.
- Inheritance is not appropriate (e.g. because the is-a rule does not apply).

- **Problem:**

- How can you most effectively make use of a method that already exists in the other class?

- **Forces:**

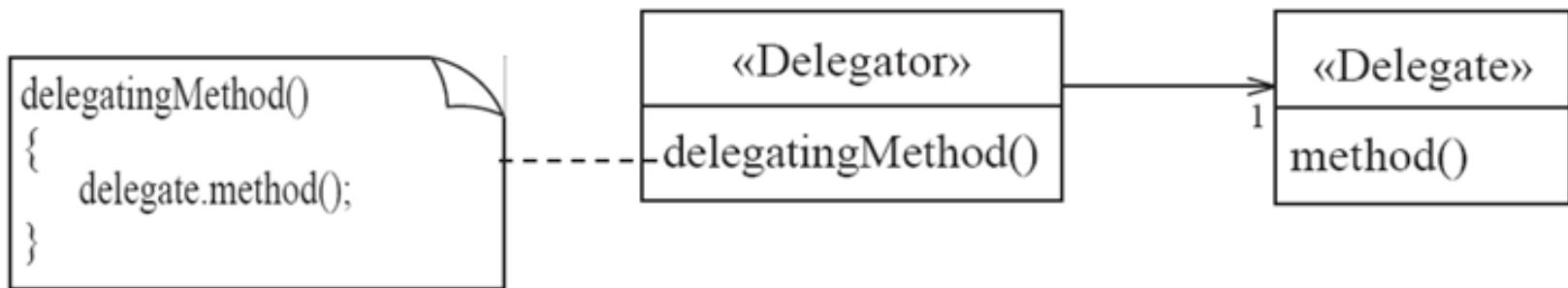
- You want to minimize development cost by reusing methods.

- **Solution:**

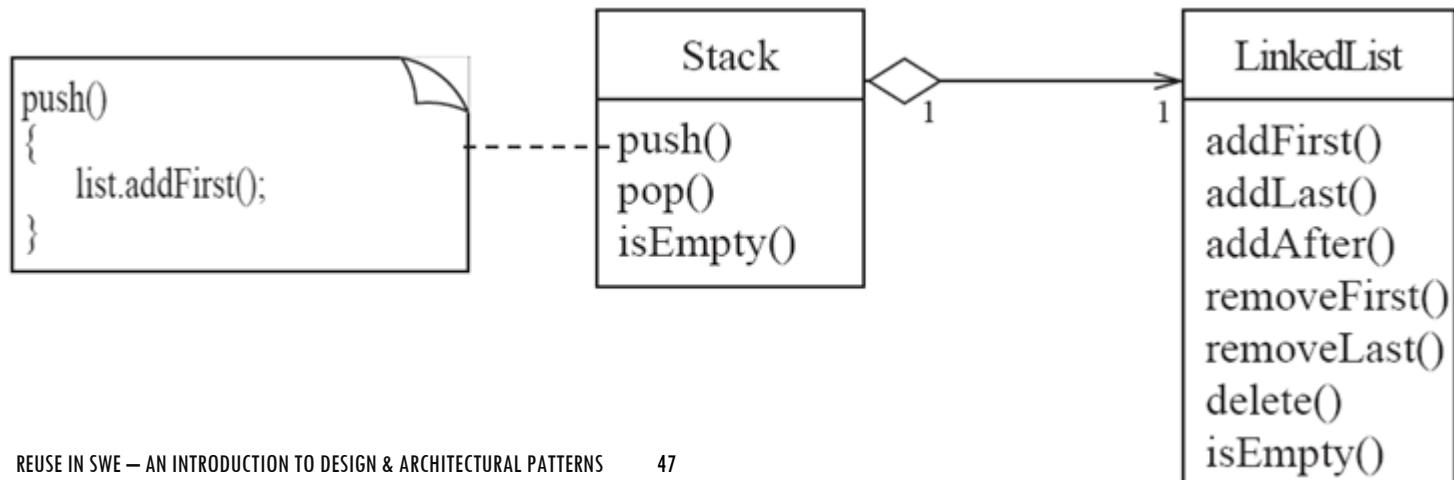
- The delegating method in the delegator class calls a method in the delegate class to perform the required task. An association must exist between the delegator and delegate classes.

SELECTED DESIGN PATTERNS

STRUCTURAL: DELEGATION PATTERN



- Example: The stack class could be created using an existing class in the java collection framework called *linkedList* using delegation pattern. The push method of stack calls the *addfirst* method of *linkedList*, etc.



SELECTED DESIGN PATTERNS

STRUCTURAL: DELEGATION ANTI-PATTERNS

1. It's common for people to overuse generalization and inherit the method that's to be reused.
 - Example: What is the problem with making the stack a subclass of the linked list?
 2. Duplication of chunks of code.
-

Note: The delegation pattern brings together **two** design principles that encourage flexible design:

1. *Favoring association over inheritance when the full power of inheritance is not needed.*
2. *Avoiding duplication of chunks of code.*

SELECTED DESIGN PATTERNS

STRUCTURAL: ABSTRACTION-OCCURRENCE

Context:

- In a domain model you find a set of related objects “occurrences”; the members of such a set share common information but also differ from each other in important ways.
- Example: All copies of the same book in a library, Episodes of a television series have different story lines, Flights that leave at the same time every day for the same destination.

Problem:

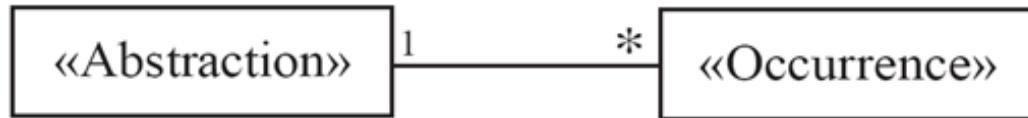
- What is the best way to represent such sets of occurrences?

Forces:

- You want to represent the members of each set of occurrences without duplicating the common information.

SELECTED DESIGN PATTERNS

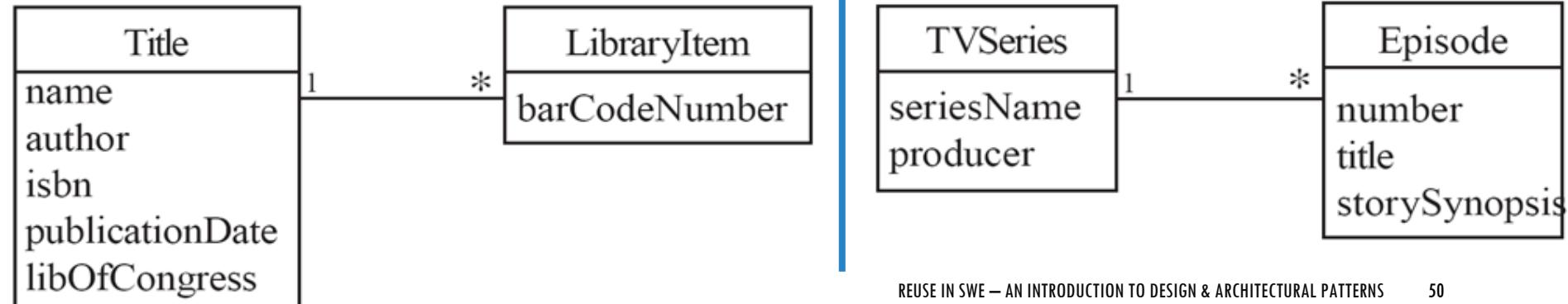
STRUCTURAL: ABSTRACTION-OCCURRENCE



Solution:

1. Create an “abstraction” class that contains the common data.
2. Then create an “occurrence” class representing the occurrences of this abstraction.
3. Connect these classes with a one-to-many association.

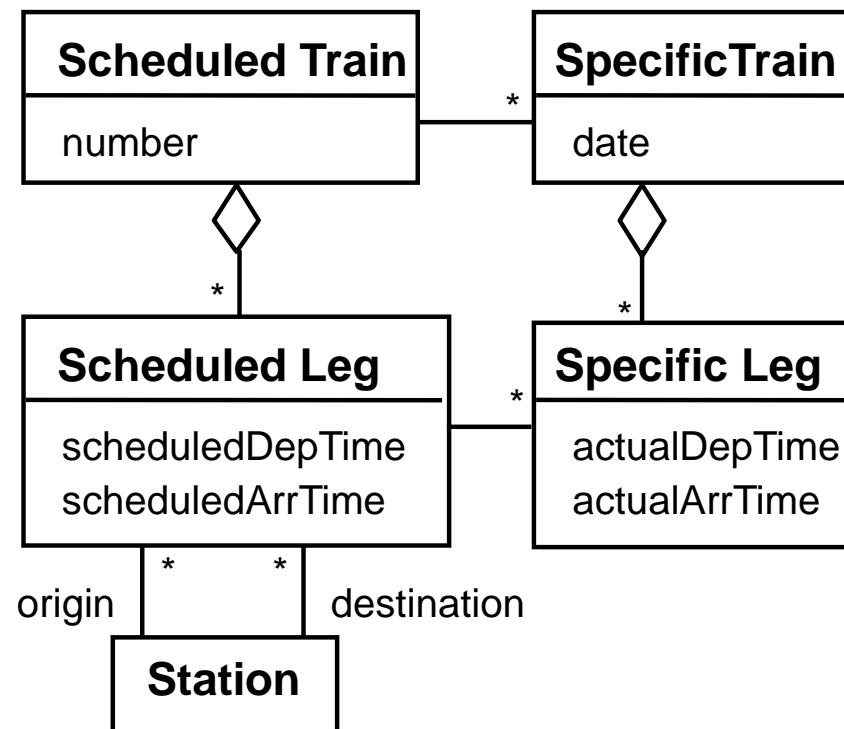
Examples:



SELECTED DESIGN PATTERNS

STRUCTURAL: ABSTRACTION-OCCURRENCE

More Examples:



SELECTED DESIGN PATTERNS

STRUCTURAL: ABSTRACTION-OCCURRENCE ANTI-PATTERNS

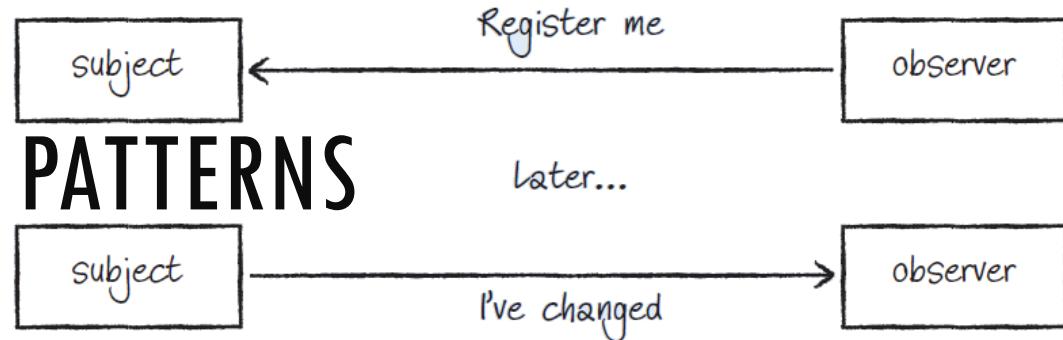
1. Using single class, there will be an object for each book; This solution leads to duplication of information in the multiple copies of the book.
2. Inheritance also will lead to the same result.



LibraryItem
name
author
isbn
publicationDate
libOfCongress
barCodeNumber

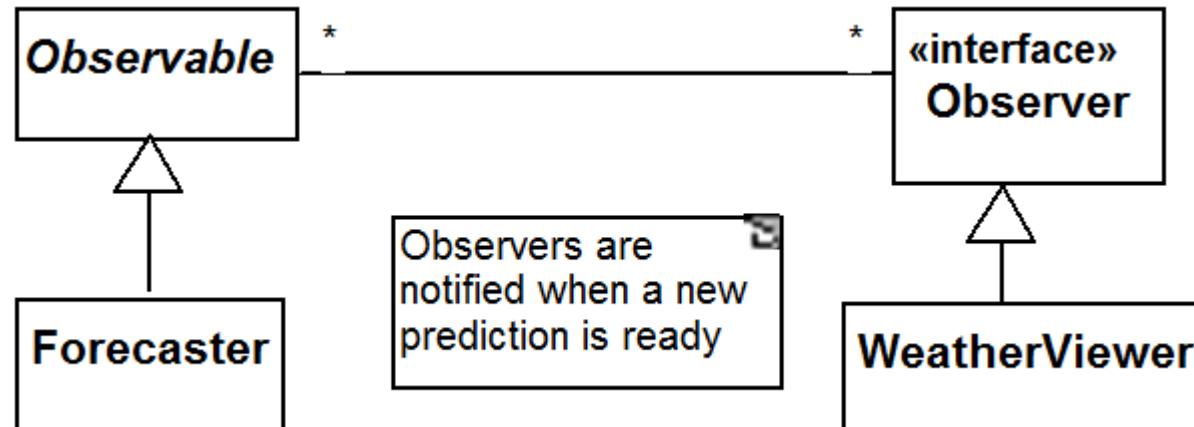
SELECTED DESIGN PATTERNS

BEHAVIOURAL ..



▪ **Observer (Publish-Subscribe) Pattern**

- An **observer** registers to receive notifications whenever the state of an **observable** (a.k.a. **subject**) changes. You should notice that this design pattern is closely related to the Notification architectural style. **Examples:** Many online forums allow you to subscribe to them and receive an email notification whenever a new forum post is made.



SELECTED DESIGN PATTERNS

BEHAVIOURAL: OBSERVER / PUBLISH-SUBSCRIBE

Context:

- When partitioning a system into individual classes you want the coupling between them to be loose so you have the flexibility to vary them independently.

Problem:

- A mechanism is needed to ensure that when the state of an object changes related objects are updated to keep them in step.

Forces:

- The different parts of a system have to kept in step with one another without being too tightly coupled.

SELECTED DESIGN PATTERNS

BEHAVIOURAL: OBSERVER / PUBLISH-SUBSCRIBE

Solution:

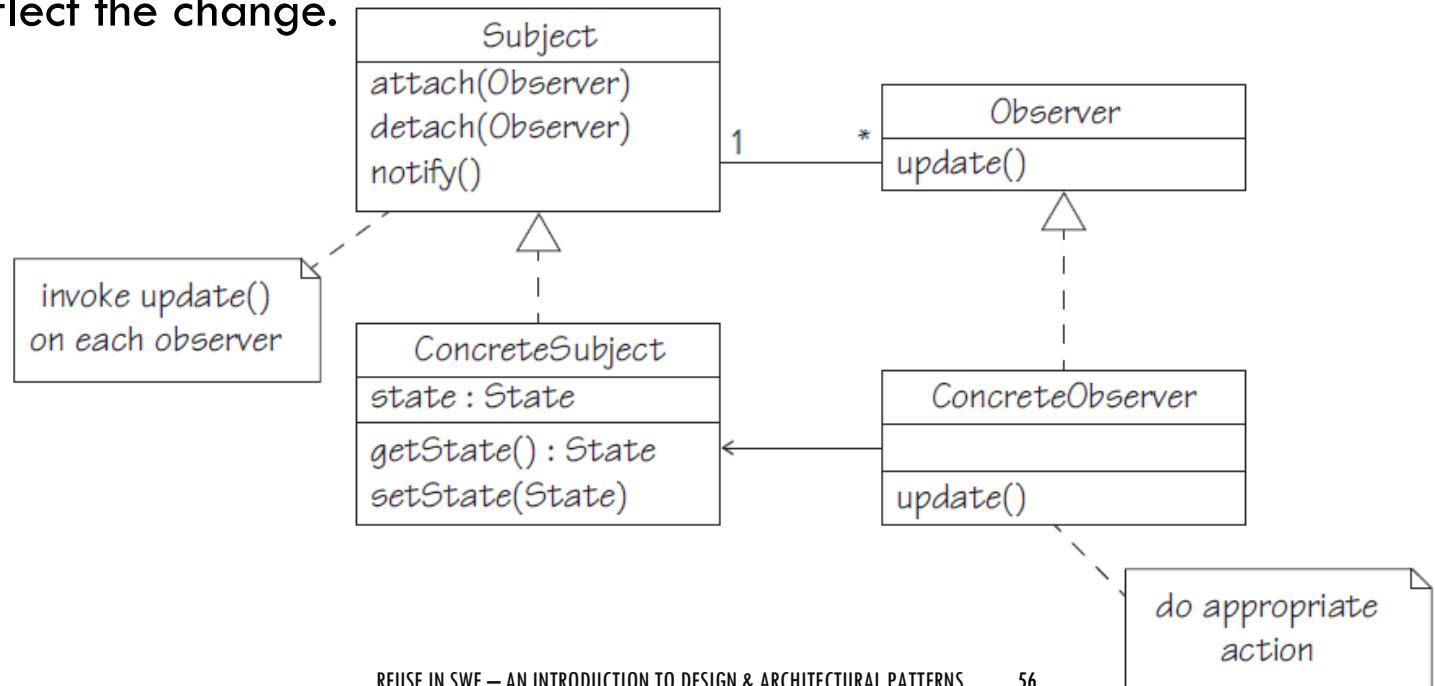
- One object has the role of the subject/publisher and one or more other objects the role of observers/subscribers. The observers register themselves with the subject, & if the state of the subject changes the observers are notified & can update themselves.
- There are two variants:
 - the **Push Model** where the subject send the observers detailed information about the change that has occurred, and ..
 - the **Pull Model** where the subject simple notifies the observers that there have been changes, and it's the responsibility of the observers to find out the details they need to update themselves.

SELECTED DESIGN PATTERNS

BEHAVIOURAL: OBSERVER / PUBLISH-SUBSCRIBE

Example:

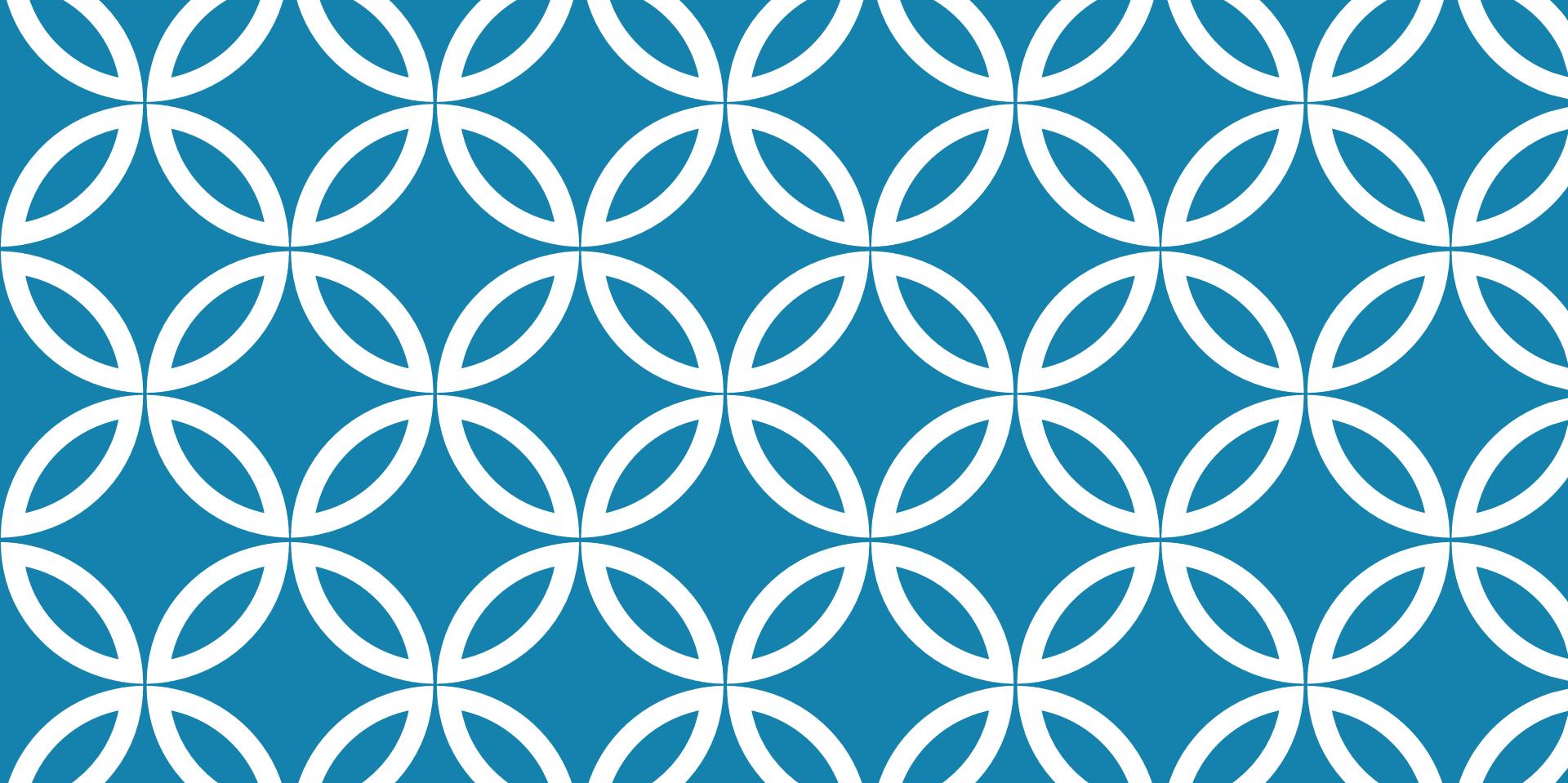
- The relationship between the view and the model in an MVC design can be realized by applying the observer pattern. The view registers with the model and is notified every time the model's state changes, allowing it to update itself to reflect the change.



THANK YOU!

Questions?





(CS251) SOFTWARE ENGINEERING I

Lecture 8
A Brief Introduction to
Testing & to Designing
Concurrent, Real-Time, &
Embedded Systems

Dr. Amr S. Ghoneim, Spring 2019

A BRIEF INTRODUCTION TO TESTING & TO DESIGNING CONCURRENT, REAL-TIME, & EMBEDDED SYSTEMS

LECTURE OBJECTIVES:

Briefly introduce the concepts of Software Testing & Quality Assurance.

Define Test-Driven Development (TDD), the Software Quality Factors, and the General Categories of Testing.

Present various techniques for testing: black box versus white box Testing, Partitioning / Boundary, Path Testing.

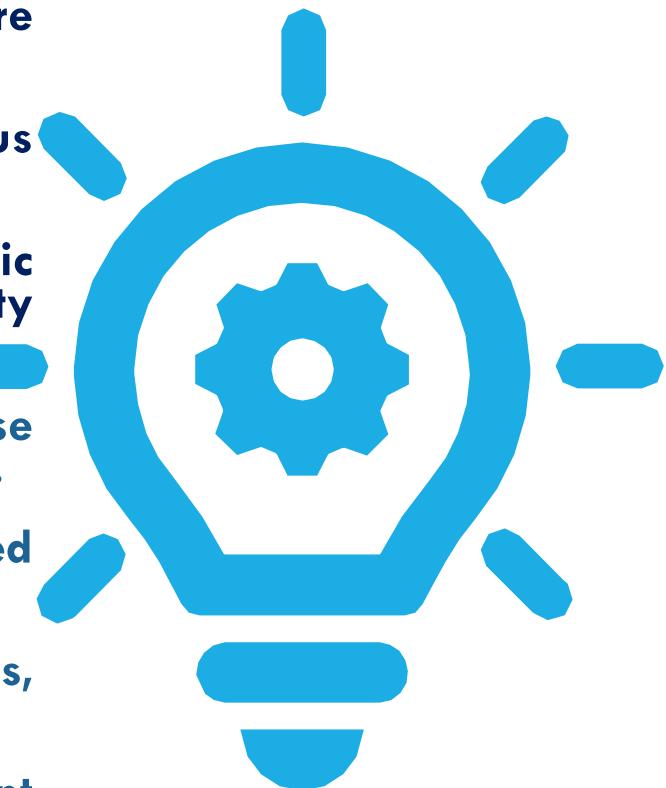
Presenting various Complexity Metrics: Cyclomatic Complexity Metric and other Object-Oriented Complexity Metrics

Outline the differences between general-purpose computer applications and concurrent/real-time systems.

Give an overview of practical real-time/embedded systems, and the degrees of real-time.

Describe concurrent problems, concurrent applications, and concurrent tasks & their various considerations.

Introduce UML for concurrent systems; concurrent collaboration diagrams, & finite state machines.



SOFTWARE TESTING & QUALITY ASSURANCE

A BRIEF INTRODUCTION

TESTING IN AGILE VERSUS PLAN-DRIVEN DEVELOPMENTS METHODS

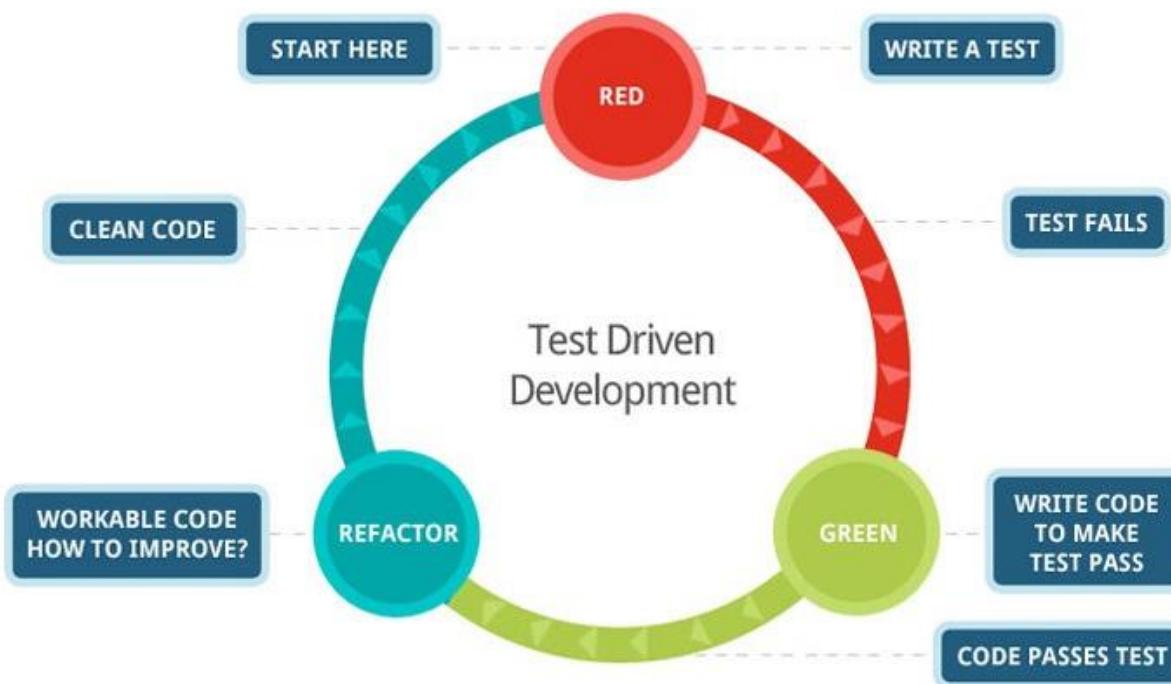
Some of the most vital differences between agile development methods and plan-driven methods stem from competing views about the proper place of testing.

One of the principal motivations for agile methods was to counter the escalating cost of changes that occur late in the development process.

Many approaches can be used to promote testing throughout development:

- ❖ **Prototyping:** The rapid creation of exploratory software artefacts that are discarded after evaluation.
- ❖ **Iterative Approaches;** Where early software artefacts are built upon rather than discarded.
- ❖ **Test-Driven Development (TDD);**

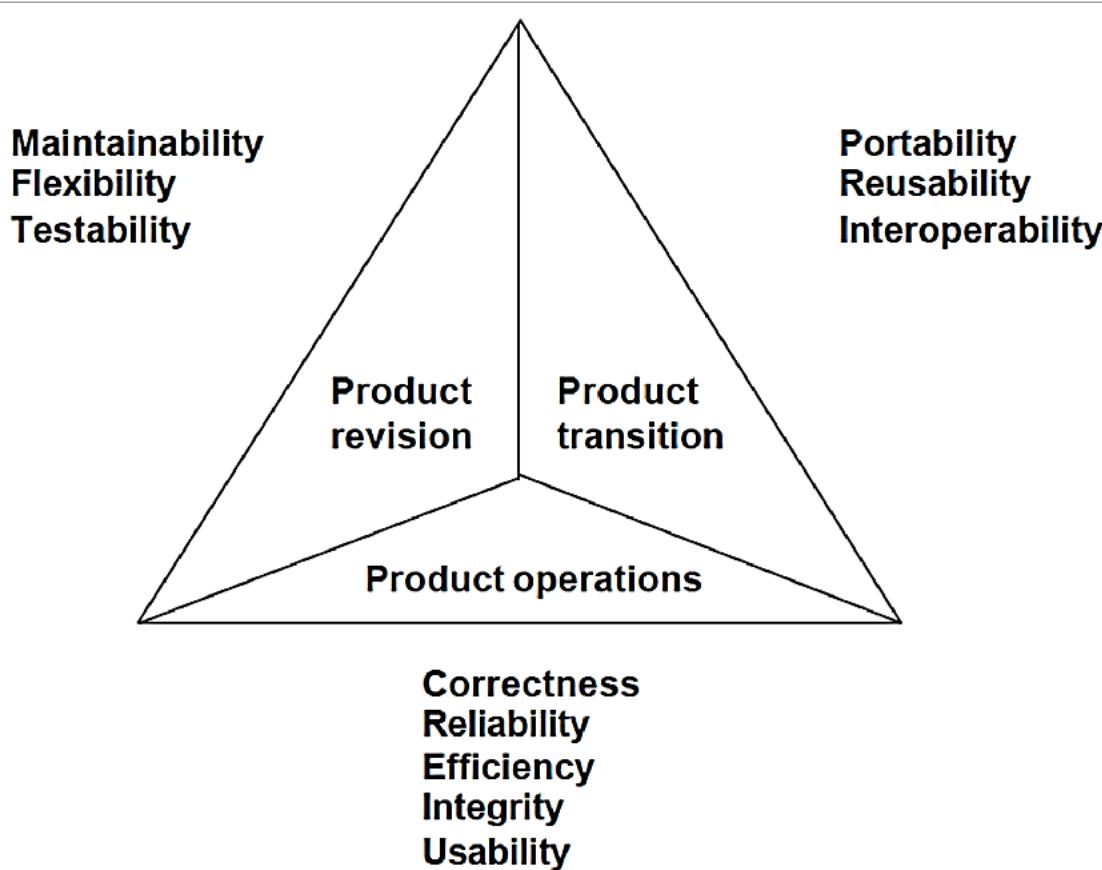
TEST-DRIVEN DEVELOPMENT (TDD)



The essence of the TDD cycle is as follows:

- Decide on code increment.
- Decide on a test.
- Write the test.
- Run all tests, expecting the new test to fail (so that you know that the test has 'teeth').
- Write the code.
- Run all the tests and succeed.
- Refactoring (if noticing a code smell or a design smell).

SOFTWARE QUALITY FACTORS



Software quality is the characteristics of a software that make it fit for its purpose.

Software Quality Factors (SQFs) are the attributes of a software product.

Product Operation Requirements; How the product will be used.

Product Revision Requirements; How the product will be changed.

Product Transition Requirements; How the product will be modified for different operating environments.

SOFTWARE QUALITY FACTORS

Quality Factor	Definition
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives
Reliability	Extent to which a program can be expected to perform its intended function with required precision
Efficiency	The amount of computing resources required by a program to perform a function
Integrity	Extent to which access to software or data by unauthorized persons can be controlled
Usability	Effort required to learn operate, prepare input, and interpret output of a program
Maintainability	Effort required to locate and fix an error in an operational program
Testability	Effort required to test a program to ensure it performs its intended function
Flexibility	Effort required to modify an operational program
Portability	Effort required to transfer a program from one hardware configuration and/or software system environment to another
Reusability	Extent to which a program can be used in other applications - related to the packaging and scope of the functions that programs perform
Interoperability	Effort required to couple one system with another

SOFTWARE QUALITY FACTORS

SOME PAIRS ARE NOT INDEPENDENT

Integrity & Efficiency; Increasing integrity within a system could affect efficiency. Increasing integrity within a system means making the system more secure. This might involve the use of passwords to access certain data & an authentication server to check a user's identity, or it might mean that network traffic needs to be encrypted and decrypted. Each of these factors adds overhead to processing, so efficiency is likely to be reduced.

Usability & Portability; Many of the features of the Apple Macintosh that contribute to its reputation of usability are built into its operating system. Applications that take advantage of these features are less portable to other systems, such as Windows or Linux.

SOFTWARE QUALITY FACTORS

PRIMARY SQFS FOR EVERYDAY SOFTWARE PRODUCTS

Not all SQFs are of primary importance in all situations. For everyday software products Tom Gilb (1988) has identified **4 primary SQFs: Correctness, Integrity, Maintainability, and Usability.**

Uncomplicated measures for assessing the quality of these four factors:

Correctness; is measured by **defects per a thousand lines of code (Defects per KLOC).** A defect (bug) is a verified lack of conformance to requirements. Defects are reported by a user of a software product after the product has been released for general use, and are counted over a standard period of time, e.g., one month.

Maintainability; is measured indirectly using the **Mean Time To Change (MTTC),** which is the average of the times it takes to analyze a bug report, design an appropriate modification, implement the change, test it, and distribute the change to all users.

SOFTWARE QUALITY FACTORS

PRIMARY SQFS FOR EVERYDAY SOFTWARE PRODUCTS

Integrity; is measured by considering the **proportion of 'attacks' on a product as opposed to bona fide (authentic) uses.** In cases where integrity is of great importance, & different kinds of attacks can be identified, it can be useful to measure the likelihood that an attack of a given type will occur within a given time, and the likelihood that it will be repelled (these probabilities can be more accurately calculated from historical data).

Usability; is measured by **users systematically trying out the user interface** of the system behind it (or a simulation of the system). A **heuristic review/evaluation** may also be used. It is a usability inspection method for computer software that helps to identify usability problems in the user interface (UI) design. It specifically involves evaluators examining the interface and judging its compliance with recognized usability principles (the "heuristics").

GENERAL CATEGORIES OF TESTING

FOUR DISTINCT CATEGORIES OF TESTING

Requirements-based testing: Draws on previously gathered testable requirements to check that a system meets the customer's requirements. The final stage in this form of testing is the acceptance testing.

Usability testing: Refers to testing the user interface (e.g., **Heuristic Reviews**).

Developmental testing: Refers to all of the testing carried out by the team developing the software. It has three levels/scopes (**Unit testing** 'e.g. **TDD**', **Integration/Component testing**, and **System testing**).

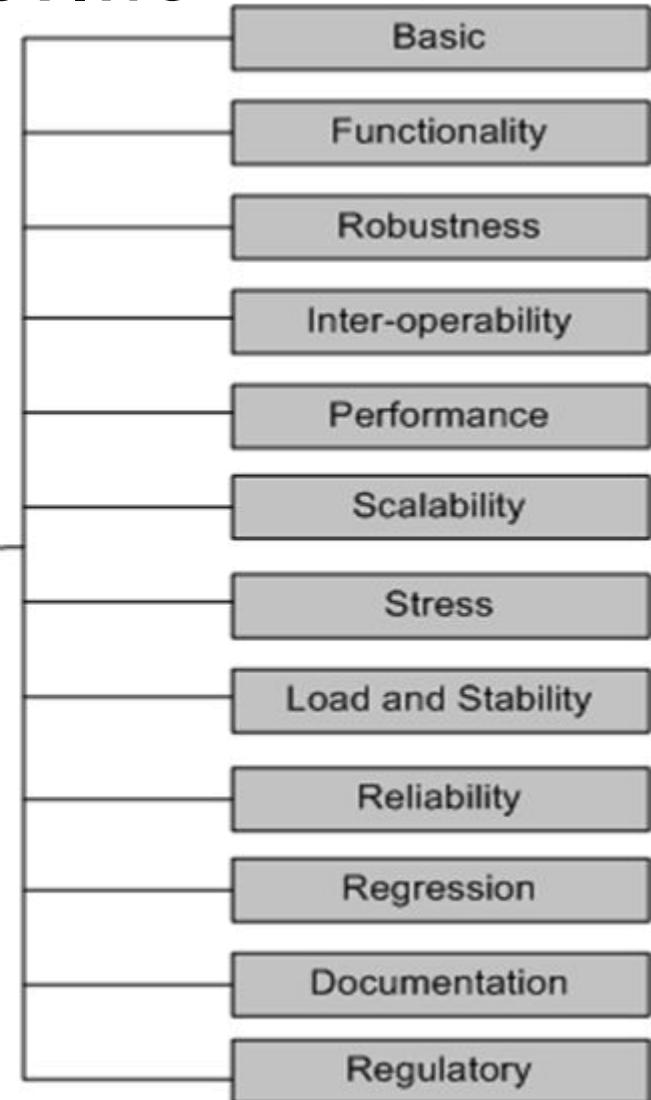
Regression testing: A form of testing during development or a system maintenance that checks that fixing one bug has not introduced others.

GENERAL CATEGORIES OF TESTING

SYSTEM TESTING

Consists of checking that a completed software system performs in accordance with its requirements specification.

Types of System Tests



GENERAL CATEGORIES OF TESTING

SYSTEM TESTING .. EXAMPLES

Functionality Testing: To make sure that functionality of product is working as per the requirements defined, within the capabilities of the system.

Interoperability Testing: To make sure whether the system can operate well with third-party products or not.

Performance Testing: To make sure system's performance under the various condition, in terms of performance characteristics.

Scalability Testing: To make sure system's scaling abilities in various terms like user scaling, geographic scaling and resource scaling.

Reliability Testing: To make sure system can be operated for longer duration without developing failures.

Regression Testing: To make sure system's stability as it passes through an integration of different subsystems and maintenance tasks.

Documentation Testing: To make sure that system's user guide and other help topics documents are correct and usable.

STRATEGIES FOR CREATING TEST CASES

BLACK BOX VERSUS WHITE BOX TESTING



Testing as a User



SOME KNOWLEDGE



Testing as a Developer

Black-box testing: Used to test that each aspect of the customer's requirements is handled correctly by an implementation.

White-box testing: Used to check that the details of an implementation are correct.

Both tests have complementary roles in the testing process.

STRATEGIES FOR CREATING TEST CASES

BLACK BOX TESTING

Equivalence Partitioning & Boundary Testing; an ideal black-box technique that focuses on producing test data at the boundaries between partitions of the input space.

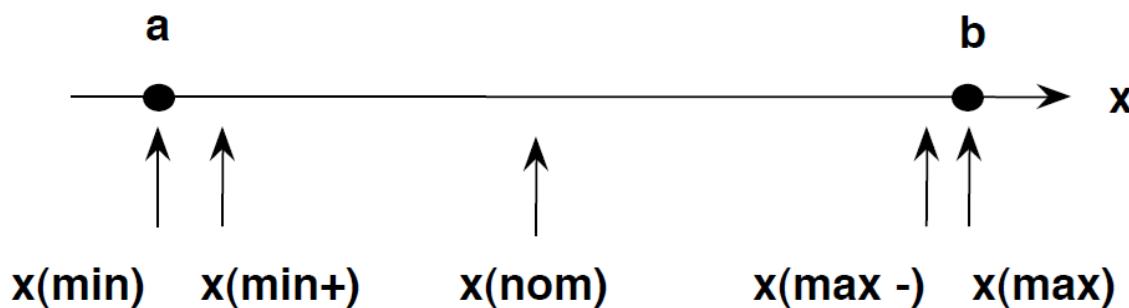
Random Testing; the test data is randomly generated within the input data.

Error Guessing; the most unexpected test data that can be thought of are presented to the program unit (an informal technique).

STRATEGIES FOR CREATING TEST CASES BLACK BOX TESTING .. PARTITIONING / BOUNDARY

Equivalent Class Partitioning is a black box technique (code is not visible to tester) which can be applied to all levels of testing like unit, integration, system, etc. In this technique, you divide the set of test condition into a partition that can be considered the same.

Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values. So these extreme ends like Start-End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing". The basic idea in boundary value testing is to select input variable values at their:



STRATEGIES FOR CREATING TEST CASES

WHITE BOX TESTING .. E.G. PATH TESTING

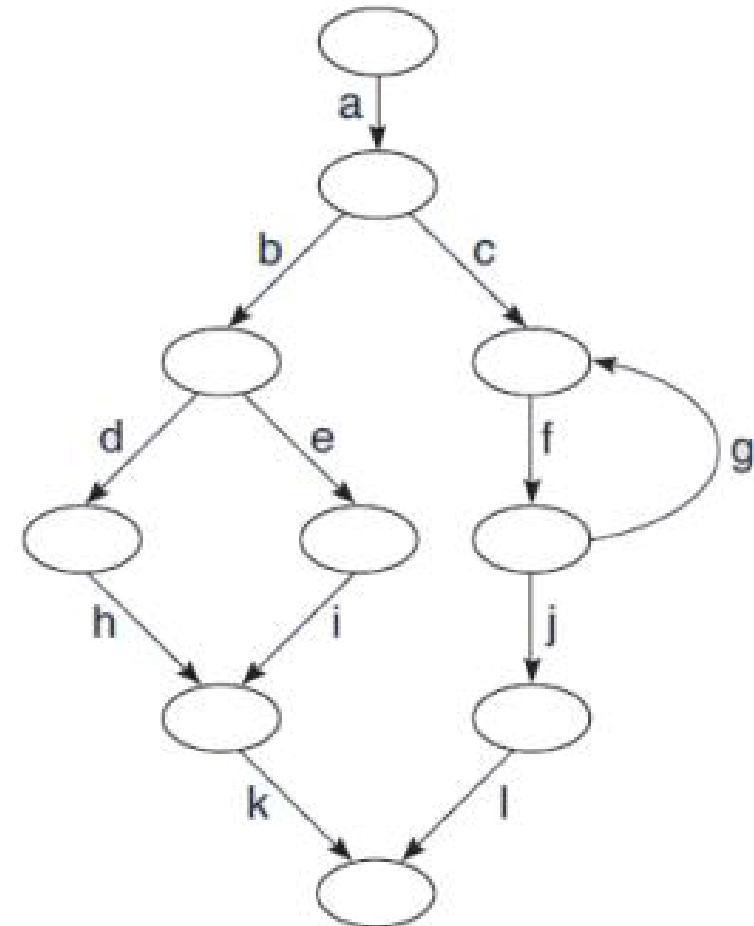
The objective of **path testing** is to ensure that the set of test cases is such that each path through the program is executed at least once.

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.

Number of test cases = number of independent paths.

Based on the Cyclomatic Complexity.



MEASURING COMPLEXITY

CYCLOMATIC COMPLEXITY METRIC

Code from `java.util.Arrays`

Line number	Code
1	public static boolean equals
	(long[] a, long[] a2) {
2	if (a==a2)
	return true;
4	if (a==null a2==null)
	return false;
6	int length = a.length;
7	if (a2.length != length)
	return false;
9	for (int i=0; i<length; i++)
	if (a[i] != a2[i])
11	return false;
12	return true;
13	}

Complexity could be measured by two approaches:

Lines-of-Code metric (LOC): given by counting the number of lines in a piece of code.

Cyclomatic Complexity metric (CCM): given by counting the number of independent paths through a method body. Cyclomatic Complexity is always counted from 1, then 1 is added for every if statement, loop statement, switch statement, try for each catch, && and || ..

◀ CCM = 7

MEASURING COMPLEXITY

CCM VERSUS LOC

Code A

```
int i = 1;  
  
while (i <= 5) {  
    playACard(i);  
    if (playerHasWon(i))  
        break;  
    i++;  
}
```

Code B

```
int j = 0;  
  
int i = 2;  
j = i;  
j = j + i;  
j++;  
  
System.out.println(j);  
System.out.println(i);
```

MEASURING COMPLEXITY

OBJECT-ORIENTED COMPLEXITY METRICS

WMC (Weighted Methods per Class) ▼ WMC is the sum of the complexity of the methods of a class. WMC = Number of Methods (NOM), when all method's complexity are considered UNITY .. OR .. the sum of the CCM of each methods.

DIT (Depth of Inheritance Tree) [Trade-off] The maximum length from the node to the root of the tree.

NOC (Number of Children) [Trade-off] Number of immediate subclasses subordinated to a class in the class hierarchy.

CBO (Coupling Between Objects) ▼ It is a count of the number of other classes to which it is coupled.

RFC (Response for Class) ▼ It is the number of methods of the class plus the number of methods called by any of those methods.

LCOM (Lack of Cohesion of Methods) ▼ Measures the dissimilarity of methods in a class via instanced variables.

A Brief Introduction

DESIGNING CONCURRENT, REAL-TIME, & EMBEDDED SYSTEMS



CONCURRENT APPLICATIONS

- Early days of computing, most applications were batch programs, each program was sequential and ran offline ..
- Nowadays, with many interactive applications, & tendency toward distributed systems, many applications are concurrent in nature ..
- Characteristics of concurrent applications:
 - ✓ Typically have many activities occurring in parallel ..
 - ✓ The order of incoming events in those applications is often not predictable and might be overlapping ..

CONCURRENT APPLICATIONS

- Concurrent versus Sequential Problems.
- Concurrent versus Sequential Applications.
- Concurrent Tasks.
- + Timing Constraints =

Real-time Systems and Applications

- + Execution at geographically different locations =

Distributed Systems and Applications

SEQUENTIAL VERSUS CONCURRENT PROBLEMS

SOME APPLICATIONS ARE CONCURRENT IN NATURE

- Activities take place in strict sequence Vs. many activities are happening in parallel ..
- Example, the execution of several compilation phases within a conventional compiler, OR processing in sequence the payroll records of each employee in a batch payroll application Vs. many users interacting with the system in parallel in a multiuser interactive system (*no prediction of which user will provide next input*) OR many activities occurring in parallel in an air traffic control system that monitor several aircrafts (*weather changes leads to unpredictable behaviors*).

SEQUENTIAL VERSUS CONCURRENT APPLICATIONS

- A sequential program that consists of **passive objects** (*has only one thread of control*) Vs. typically several **active objects** (*each with its own thread of control*) ..
- Only **synchronous message communications** (*procedure call or method invocation*) is supported Vs. **Asynchronous message communication** (*an active object can send an asynchronous message to an active destination object and then continue executing, regardless of when the destination object receives the message “buffered if the receiving object is busy when it arrives”*) ..

CONCURRENT TASKS (CONCURRENT PROCESSES)

- A fundamental concept in the design of concurrent applications.
- A concurrent application consists of many tasks that execute in parallel (*thus applicable to both Real-time and Distributed systems*). In addition .. applied extensively in:
 - Operating Systems.
 - Database Systems.
 - Interactive Systems.
- Key Issues: .. *Providing capabilities for ..*
 - Structuring the application into concurrent tasks.
 - Tasks to communicate and synchronize their operations.

ADDITIONAL CONSIDERATIONS FOR CONCURRENT APPS: TIME .. REAL-TIME SYSTEMS & APPLICATIONS

- .. Concurrent Systems with timing constraints.
- Real-time systems usually refers to the whole system, including the real-time application, real-time operating system, and the real-time I/O subsystem.

ADDITIONAL CONSIDERATIONS FOR CONCURRENT APPS: TIME .. REAL-TIME SYSTEMS & APPLICATIONS

Characteristics:

- **Embedded Systems;** a component of a larger hardware/software system.
- **Interaction with External Environment;** typically interacts with an external environment that is to a large extent nonhuman.
- **Timing Constraints;** must process events within a given timeframe.
- **Real-time Control;** makes control decisions based on input data without any human intervention.
- **Reactive Systems;** are event driven and must respond to external stimuli.

DEGREES OF “REAL-TIME”

- All practical systems are ultimately real-time systems
- Even a batch-oriented system — for example, grade processing at the end of a semester — is real-time.
- Although the system may have response times of days, it must respond within a certain time.
- Even a word-processing program should respond to commands within a reasonable amount of time.
- Most of the literature refers to such systems as soft real-time systems.

SOFT, HARD, AND FIRM “REAL-TIME”

Definition: Soft Real-Time System

A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response-time constraints.

Definition: Hard Real-Time System

A hard real-time system is one in which failure to meet even a single deadline may lead to complete or catastrophic system failure.

Definition: Firm Real-Time System

A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete or catastrophic system failure.

PRACTICAL EMBEDDED SYSTEMS

Aerospace

- Flight control
- Navigation
- Pilot interface

Automotive

- Airbag deployment
- Antilock braking
- Fuel injection

Household

- Microwave oven
- Rice cooker
- Washing machine

Industrial

- Crane
- Paper machine
- Welding robot

Multimedia

- Console game
- Home theater
- Simulator

Medical

- Intensive care monitor
- Magnetic resonance imaging
- Remote surgery

ADDITIONAL CONSIDERATIONS FOR CONCURRENT APPS: VARYING LOCATIONS .. DISTRIBUTED SYSTEMS & APPLICATIONS

- .. Concurrent applications that execute in an environment consisting of multiple nodes that are in geographically different locations (each node is a separate computer system, connected to each other by a LAN/WAN) ..
- Distributed Systems usually refer distributed operating systems, distributed file systems, and distributed databases.

ADDITIONAL CONSIDERATIONS FOR CONCURRENT APPS: VARYING LOCATIONS .. DISTRIBUTED SYSTEMS & APPLICATIONS

Advantages:

- **Improved Availability;** no single point of failure (when some nodes are unavailable).
- **More Flexible Configuration;** appropriate number of nodes can be selected for any given application.
- **More Localized Control & Management;** distributed subsystems are designed to be autonomous (execute independently).
- **Incremental System Expansion;** new nodes added when system is overloaded.
- **Reduced Cost;** cheaper than an equivalent centralized solution.
- **Load Balancing;** overall system load can be shared among nodes.
- **Improved response time;** requests of local users on local systems processed in a more timely fashion.

CONCURRENT PROCESSING

- Overall system concurrency is obtained by having multiple tasks executing in parallel.
 - A task represents the execution of a sequential program/component within the concurrent system.
 - Thus, no concurrency is allowed within a task.
- Tasks usually execute asynchronously (i.e., at different speeds), and are relatively independent of each other for significant periods of time.
 - Thus, from time to time they need to communicate and synchronize their operations with each other.

CONCURRENT PROCESSING

COOPERATION BETWEEN CONCURRENT TASKS

- The mutual exclusion problem.
- Task synchronization problem.
- The producer/consumer problem.

COOPERATION BETWEEN CONCURRENT TASKS

THE MUTUAL EXCLUSION PROBLEM

- It arises when it's necessary for a shared resource to be accessed by only one task at a time.
- For example, If two or more tasks are :
 - .. allowed to write to a printer simultaneously; output will be randomly interleaved.
 - .. allowed to write to a data repository simultaneously; inconsistent/incorrect data will be written.

COOPERATION BETWEEN CONCURRENT TASKS

THE MUTUAL EXCLUSION PROBLEM

- A **synchronization mechanism** ensures that access to a critical resource by concurrent tasks is mutually exclusive.
- A task .. Acquires the resource .. Uses the resource .. Releases the resource.
- **Binary Semaphore; Classical Solution (proposed by Dijkstra)**
 - A Boolean variable, only accessed by 2 indivisible operations: '**P**' **to acquire** (semaphore) and '**V**' **to release** (semaphore).
 - Semaphore is initially set to 1 (i.e., free resource),
 - The acquire operation decrements the semaphore to 0 (thus other tasks are suspended).
- Code executed by a task while it has access to the mutually exclusive resource is referred to as the **critical section/region**.

COOPERATION BETWEEN CONCURRENT TASKS

TASK SYNCHRONIZATION PROBLEM

- Two tasks need to synchronize their operations with each other.
- **Event Synchronization** is used when two tasks need to synchronize their operations without communicating data between tasks.
 - The source task executes a signal (event) operation; signals that an event has taken place.
 - The destination task executes a wait (event) operation; suspends the task until the source task has signaled the event (*if the event has already been signaled, the destination task is not suspended*).
 - Event Synchronization is asynchronous.

COOPERATION BETWEEN CONCURRENT TASKS

TASK SYNCHRONIZATION PROBLEM

- For example; each robot system is designed as a concurrent task, & controls a moving robot arm.
- A **pick-and-place robot** brings a part to the work location so that a **drilling robot** can drill four holes in the part. On the completion of the drilling operation, the pick-and-place robot moves the part away.

Synchronization Problems:

1. A collision zone where the pick-and-place robot & drilling robot arms could potentially collide.
2. The pick-and-place robot must deposit the part before the drilling robot can start drilling the holes.
3. The drilling robot must finish drilling before the pick-and-place robot can remove the part.

COOPERATION BETWEEN CONCURRENT TASKS

TASK SYNCHRONIZATION PROBLEM

- For example; each robot system is designed as a concurrent task, & controls a moving robot arm.
- A **pick-and-place robot** brings a part to the work location so that a **drilling robot** can drill four holes in the part. On the completion of the drilling operation, the pick-and-place robot moves the part away.

Synchronization Problems:

1. A collision zone where the pick-and-place robot & drilling robot arms could potentially collide.
2. The pick-and-place robot must deposit the part before the drilling robot can start drilling the holes.
3. The drilling robot must finish drilling before the pick-and-place robot can remove the part.

COOPERATION BETWEEN CONCURRENT TASKS

TASK SYNCHRONIZATION PROBLEM

Pick & Place Robot:

```
while workAvailable do
    Pick up part
    Move part to work location
    Release part
    Move to safe position
    signal (partReady)
    wait (partCompleted)
    Pick up part
    Remove from work location
    Place part
end while;
```

Drilling Robot:

```
while workAvailable do
    wait (partReady)
    Move to work location
    Drill four holes
    Move to safe position
    signal (partCompleted)
end while;
```

COOPERATION BETWEEN CONCURRENT TASKS

PRODUCER / CONSUMER PROBLEM

- When tasks need to communicate with each other (*inter-process communication* “IPC”) in order to pass data from one task to another.
 - **Producer task** produces information,
 - It’s then consumed by the **consumer task**.
- In **sequential programs**, a calling operation (procedure) passes data (and control) to a called operation.
- In **concurrent systems**, each task has its own thread of control and executes asynchronously (*thus it’s necessary to synchronize their operations when exchanging data*).
 - Why? .. The producer must produce the data before the consumer can consume

COOPERATION BETWEEN CONCURRENT TASKS

PRODUCER / CONSUMER PROBLEM

Example:

- A vision system has to inform a robot system of the type of part coming down a conveyor (e.g., *the car body frame is sedan or station wagon*).
- The robot has a different welding program for each body type.
- The vision system has to send the robot information about the location and orientation of a part on a conveyor (*usually as an offset/relative-position from a point known to both systems*).

COOPERATION BETWEEN CONCURRENT TASKS

PRODUCER / CONSUMER PROBLEM

Vision System:

```
while workAvailable do
    wait (carArrived)
    Take image of car body
    Identify the model of the car
    Determine the location and
        .. orientation of car body
    signal carIDMessage (carModelID,
        .. carBodyOffset) to Robot System
    wait for reply
    signal (moveCar)
end while;
```

Robot System:

```
while workAvailable do
    wait for message from Vision System
    recieve carIDMessage
        .. (carModelID, carBodyOffset)
    Select welding program for
        .. carModelID
    Execute welding program using
        .. carBodyOffset for car position
    send (doneReply) to Vision System
end while;
```

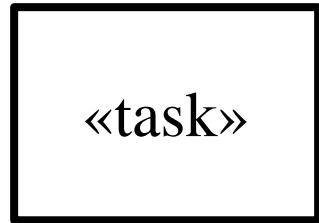
ADDITIONAL UML NOTATION

CONCURRENT COLLABORATION DIAGRAMS

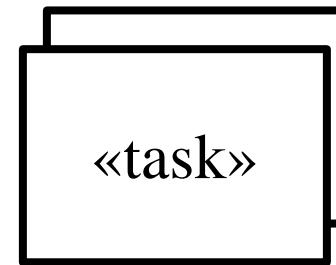
- An **active object** has its own thread of control and executes concurrently with other objects.
- A **passive object** does not have a thread of control. It executes only when another object (passive or active) invokes one of its operations.
- **Tasks** (active objects) are depicted on concurrent collaboration diagrams (depicting the concurrency aspects of the system).

ADDITIONAL UML NOTATION

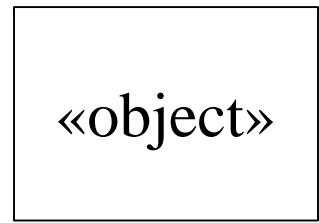
CONCURRENT COLLABORATION DIAGRAMS



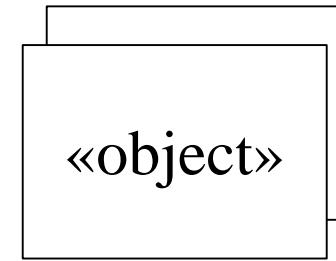
Active object



Active multiobject



Passive object



Passive multiobject

An active object or task is depicted by using a **thick outline** for the object box, while **thin black lines** are used for passive objects.

ADDITIONAL UML NOTATION

INTRODUCTION TO STATE MACHINES

In this section we will describe a technique for modelling the life history of an object of a class as part of the dynamic behavior of a system.

That is, you will be shown how to model the changes within an object as a result of receiving a message.

- This technique can also be used to model the changes occur as a result of events in that domain.
- A **State machine**: is a model that shows how an object changes from state to state in response to events.

INTRODUCTION TO STATE MACHINES

IMPORTANT TERMS IN STATE MACHINES

A **state**: represents some condition or situation in the life of an object.

- It is a description of something about the object that will remain true until something happens.
- For example a car's motor might be running, or a window might be shut or open; these are **states**.

An **event**: is something done to an object such as sending it a message. The receipt of a message is always an event.

- An event may result in a change of state or it may not.
- A car's motor might start (an **event**) to put it into the running state.

INTRODUCTION TO STATE MACHINES

IMPORTANT TERMS IN STATE MACHINES

An **action**: is something done by an object in reaction to an event.

- ▶ It is something an object does such as sending message to itself or to other objects.

Transition: occurs as a response to an event, and causes a change of state.

- For example, in an Hotel System, when an interface sends the message `checkIn(_)` to the Hotel object, in this case an **event** happens to the Hotel object.
- How the Hotel object will react and what actions he will do to respond to this event (message) are the **actions**.

INTRODUCTION TO STATE MACHINES

IMPORTANT TERMS IN STATE MACHINES

The term **state machine** refers to the technique used to model behavior of an object in terms of states and transitions.

State-chart diagram is a graphical representation of a state machine showing states, transitions, and events that captures the life history of an object.

- It models changes within an object as a result of receiving a message.
- It belongs to the dynamic modeling techniques.

INTRODUCTION TO STATE MACHINES

ELEMENTS OF STATE CHART DIAGRAMS

1. **States**: Boxes with rounded corners:

- **Initial state** (when an object is first created): Black solid circle
 - A *transition from initial state should not have an event label.*
- **Final state** (in which the state machine is completed according to some events): Black circle surrounding a solid black circle.

2. **Transitions**: Arrows between states:

- [..] “**guard**”, which represents a condition.
- In a state-chart, a guard is used to choose which transition to take if the same event appears on more than one transition.
- A guard is evaluated when an event occurs. It is used also when there are loops.

INTRODUCTION TO STATE MACHINES

ELEMENTS OF STATE CHART DIAGRAMS

3. **Events**: event / action: use backslash / to label a transition; to show an event and the associated action(s).

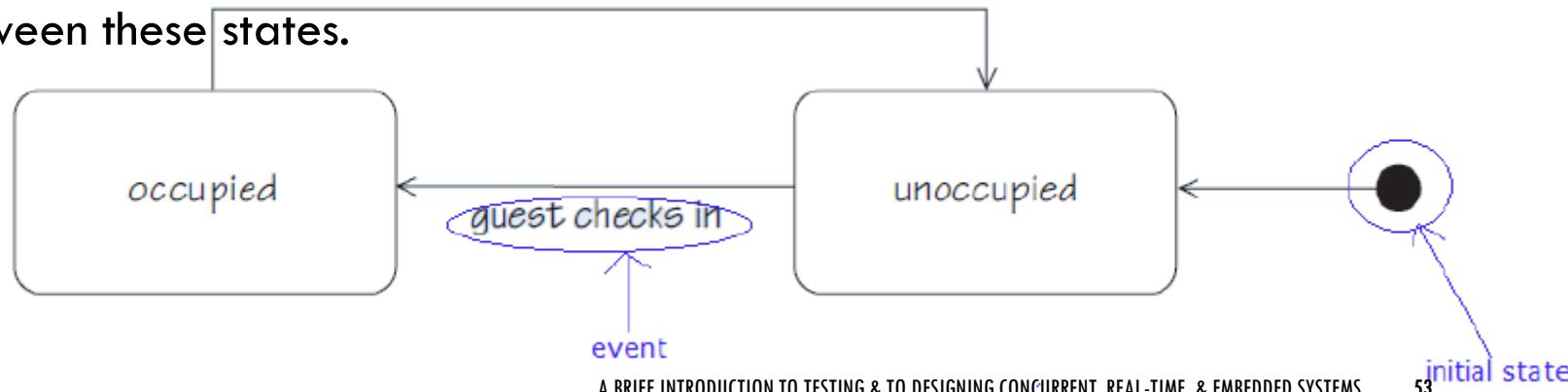
INTRODUCTION TO STATE MACHINES

ELEMENTS OF STATE CHART DIAGRAMS

The Figure shows how a state machine for instances of the class Room can be represented in a state-chart diagram.

A room becomes occupied when a particular guest checks into the hotel. It becomes unoccupied when that guest checks out.

- States:** Occupied and unoccupied
- Events:** Check-In and Checkout are the events that fire the transitions between these states.
guest checks out

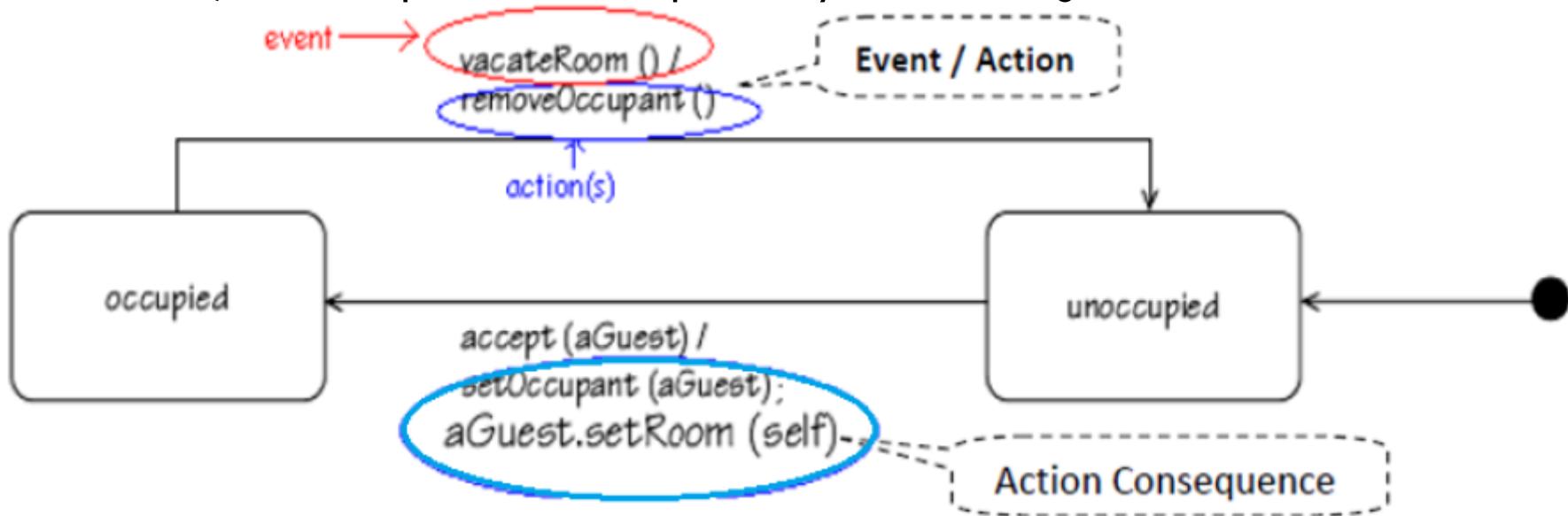


INTRODUCTION TO STATE MACHINES

ELEMENTS OF STATE CHART DIAGRAMS

This Figure shows a revised version of the state chart diagram of figure 1.

- **An action consequence:** is an ordered series of individual actions that are associated with a particular event. They are written as a list separated by semicolons, and are performed sequentially in left-to-right order.



INTRODUCTION TO STATE MACHINES

ELEMENTS OF STATE CHART DIAGRAMS

Note that: If two objects (of the same class) have different values for the same attributes, they are in different states, and therefore they *might* respond differently to the same message.

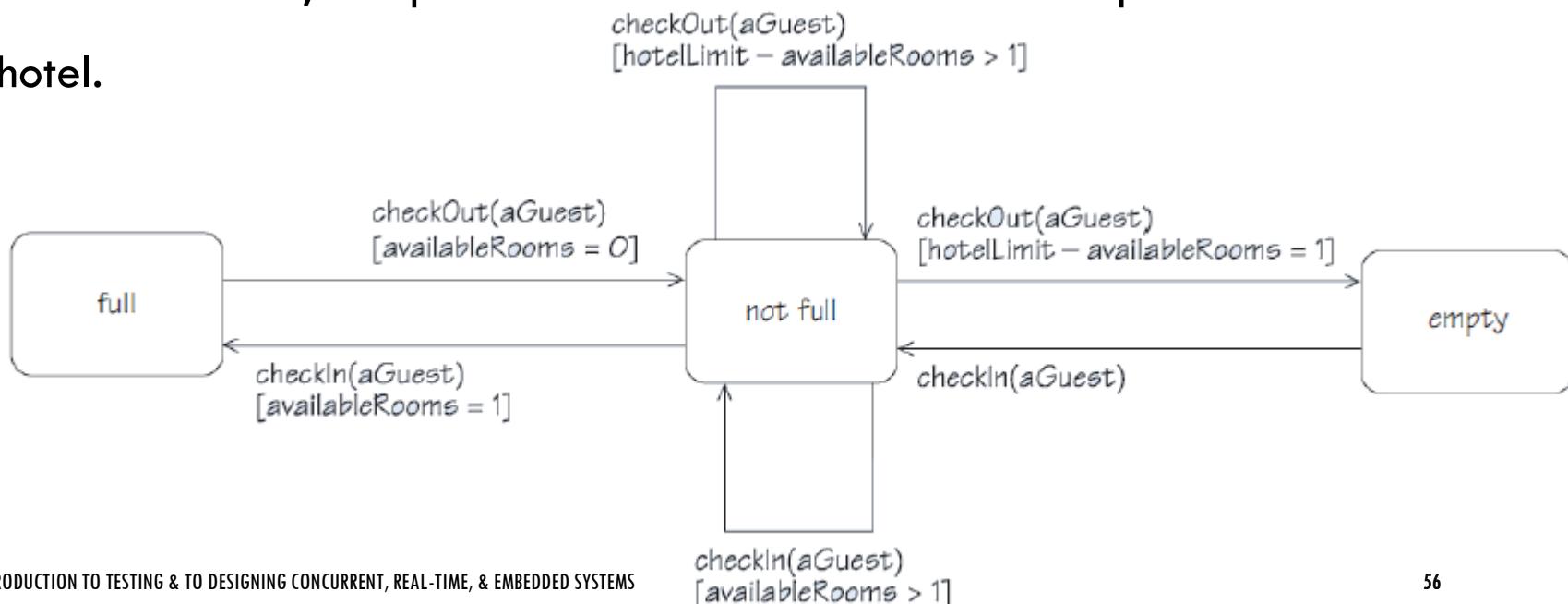
A transition in a state-chart diagram have a(n):

1. Source state,
2. Target state,
3. Event trigger,
4. Action (or a sequence of actions);,
5. Guard.

INTRODUCTION TO STATE MACHINES

ELEMENTS OF STATE CHART DIAGRAMS

This Figure shows one way to follow the checking in and out of guests into a hotel that might be full, empty or somewhere in between. The maximum number of rooms in a hotel is denoted by `hotelLimit`. A simple counter, called `availableRooms`, keeps track of the number of unoccupied rooms in the hotel.





IT'S OVER

IT'S FINALLY OVER