

Design of Information Systems

Lecture 1

Prof. Ramona BOLOGA
Ramona.bologa@ie.ase.ro

General objectives (Syllabus)



Acquiring the *competences* necessary to use the concepts, theories, principles, methods, processes and technological tools in order to *develop information systems* by following all the stages specific to the development cycle:

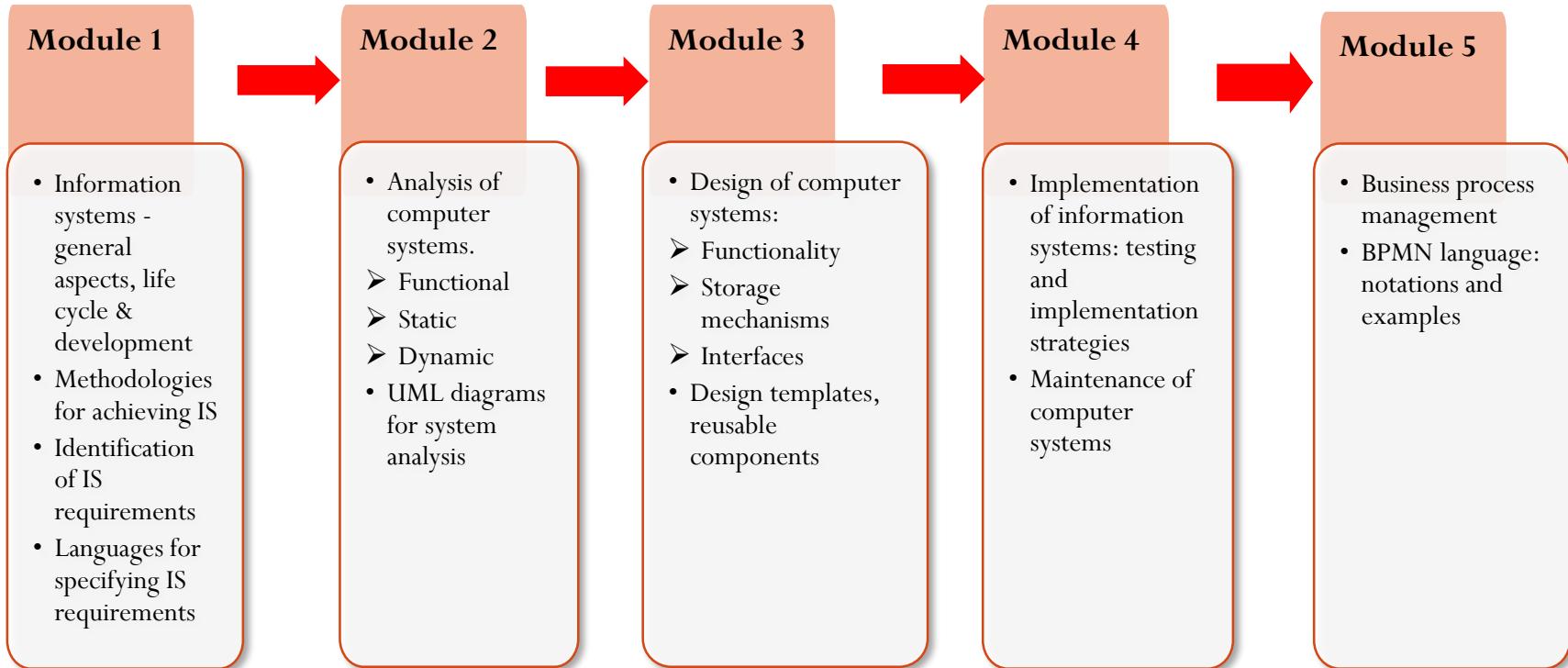
- Requirement specification
- Analysis
- Design
- Implementation
- Testing

Specific objectives(Syllabus)



- Knowledge of the **stages of developing** a computer system;
- Identifying, interpreting and modeling the **requirements** for designing and developing new information systems;
- Use of economic **notions** in solving problems by developing new IT subsystems or IT systems in the organization;
- Use of **analysis and design methods** specific to the development of information systems;
- Defining the requirements and characteristics of updating the IT systems in the organization;
- Elaboration of studies of **specifications** for the design and realization of components of computer systems

Course structure





Assessment method

- <http://online.ase.ro>
- **Final grade:**
 - 50% seminar grade (minimum 5 – individual project)
 - 50% course grade
 - 1st written test (in the 7th week): 8.XI, **2p**
 - Multiple choice quiz (includes practical questions), **8p**
 - **2 bonus points** for an ESSAY (see requirements and deadlines on online.ase.ro)
- **Guest presentations:**
 - 9th week (20.11)– Codeless (Ana Orsivschi)
 - 10th week (27.11)- Agile methodologies and BPMN (Elena Puica)
 - 12th week (11.12) – DevOPS (Pavel Craciun)

Resources

- <https://online.ase.ro/>
- Ion Lungu, Gheorghe Sabau, Manole Velicanu, Mihaela Muntean, Simona Ionescu, Elena Posdarie, Daniela Sandu - *Sisteme informaticе. Analiza, proiectare, implementare*, Ed. Economică, Bucuresti, 2003
- Anca Andreeșcu - *Dezvoltarea sistemelor software pentru managementul afacerilor*, Ed. ASE, 2010
- A. Dennis, B. H. Wixom and D. Tegarden - Systems analysis and design: An object-oriented approach with UML, John Wiley & Sons, 2015. Available at:
<http://www.arxen.com/descargas/PulzarCloud/Books/systems-analysis-and-design-with-uml-5th-edition.pdf>
- OMG specifications for UML. Available at:
<https://www.omg.org/spec/UML/2.5.1/PDF/>
- OMG specifications for BPMN. Available at:
<https://www.omg.org/spec/BPMN/2.0/PDF>

Agenda

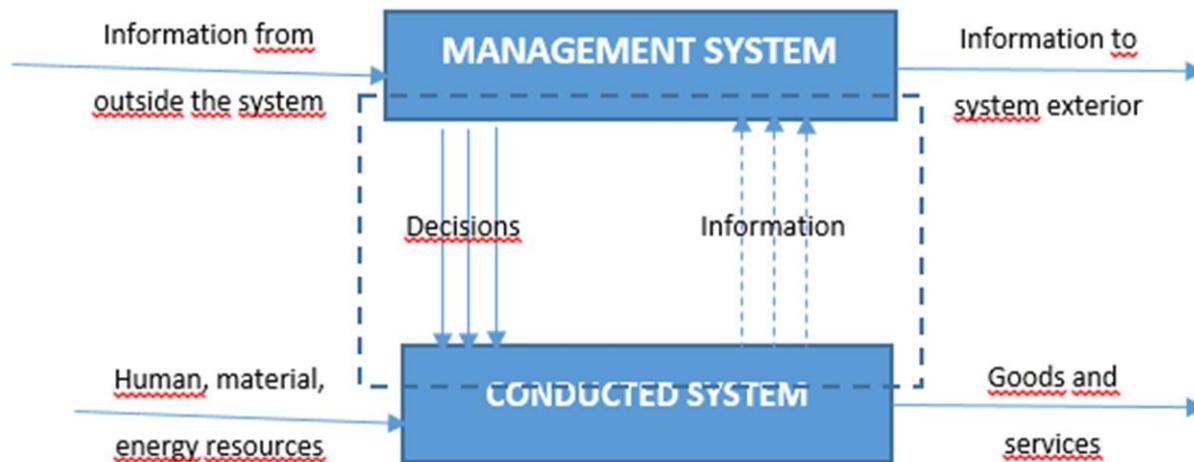
- ✓ Information system– definition and components
- ✓ Types of computer systems
- ✓ Computer system lifecycle and development cycle
- ✓ Strategies for computer-based automation of organization's activities

Information system concept

- **Information system** is used for collecting, storing, processing and generating the necessary information for activity management and decision making
- The use of information systems **adds value** into organizations, enhancing emergence of techniques and support technologies for **computer system** development.

Information system concept

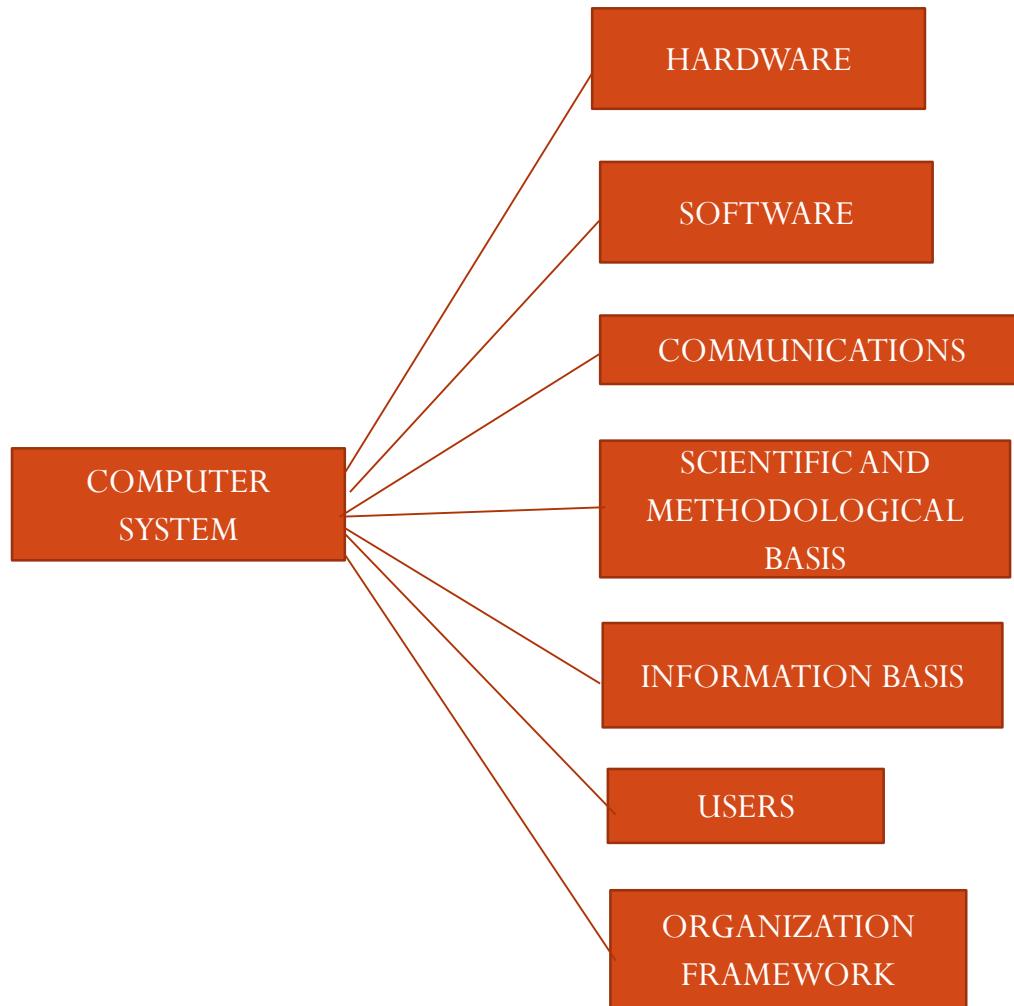
- Communication between various systems, subsystems and within them is performed via **information system**, that is situated between the conducted system and the management system
- Information system can be defined as a **technical and organizational assembly of procedures** for identifying, recording, collecting, verifying, transmitting, storing and processing data in order to meet the **information requirements** necessary for management in the process of decision making



Information system – definition and components

- When activities within the information system are accomplished using **electronic devices** for **automated** data collecting, transmitting, storing and processing, it is called information system automation; this causes the use of **computer system concept**
- Computer system** represents an assembly of functionally interrelated elements that **automate** the obtaining of necessary information used in decision making process
- The main components of a computer system are: hardware, software, communications, scientific and methodologic basis, information basis, users and organization framework. They are functionally interrelated within the system.

Computer system – definition and components

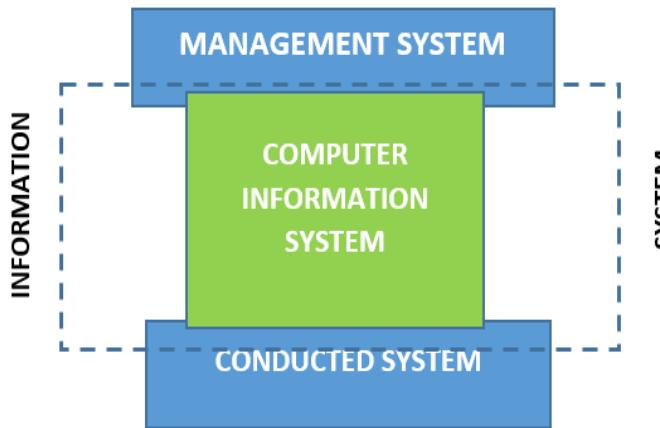


Computer information system – definition and components

- **HARDWARE** – all the technical devices involved in automated data collection, transmission, storage and processing
- **SOFTWARE** – all the software programs needed to make the information system run, according to its established functions and objectives. This applies to both basic programs (basic software) and application programs (application software).
- **COMUNICATIONS** – all equipment and technologies involved in data communication among systems.
- **SCIENTIFIC AND METHODOLOGIC BASIS** – all the models of economic processes and phenomena, methodologies, methods and techniques for information system development
- **INFORMATION BASES** – all data that are processed, all the information flows, coding systems and catalogs (nomenclatures)
- **USERS** – all the specialists necessary for running the information system. The specialty staff includes: computer specialists with higher or secondary education, analysts, programmers, system engineers, operators, etc.
- **ORGANISATION FRAMEWORK** – it is the one specified by the internal rules and regulations on the organization and functioning of the company which hosts the information system.

Role and place of the computer information system reported to the information system

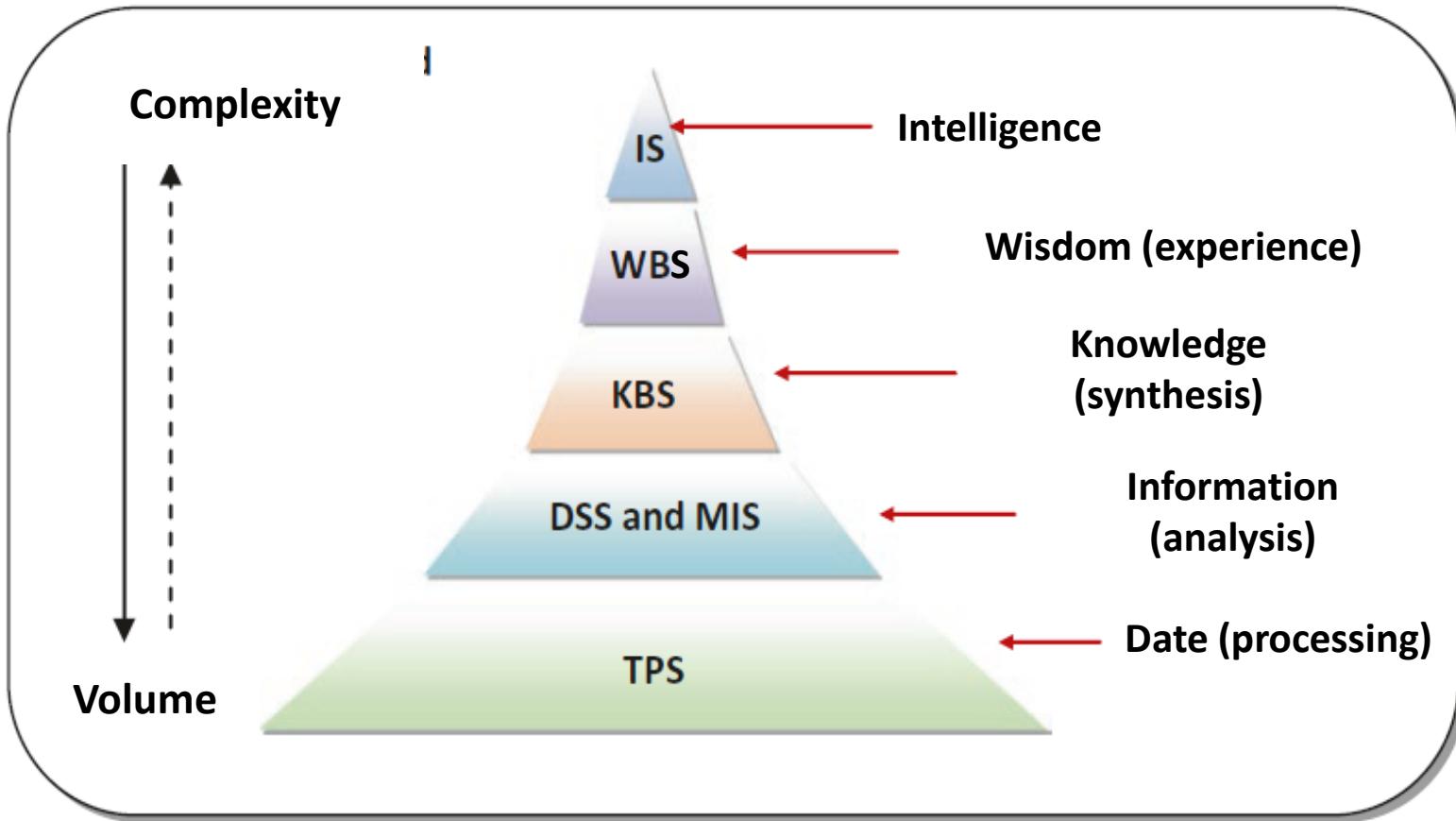
- The computer system is included within the information system and it has as main object of activity the process of **automated data collection, validation, transformation, storage and processing**
- As it implements mathematical models and it uses information technology, computer information system adds new aspects in terms of quantity and quality to the generic information system



- Computer information system scope tends to equal information system scope, but this will never be entirely possible because of the technology limitations. Usually, there will always be some activities that cannot be 100% automated within the information system scope.



TYPES OF INFORMATION SYSTEMS



Pyramid DIKW – Data Information Knowledge Wisdom

TPS - Transaction Processing Systems, DSS - Decision Support Systems,

MIS- Management Information Systems, KBS – Knowledge Based Systems,

WBS – Wisdom Based Systems, IS – Intelligent Systems



DATA PYRAMID

- **Data pyramid** is a popular representation model of the relationships between data, information, knowledge and wisdom in the *Data, Information, Knowledge, Wisdom*
- **Data** is represented by the elemental entities that form the basis of the pyramid. The data can be defined as raw observations.
- Once the data is collected and processed, the **information** will be generated. Data processing is the key operation that transforms data into information. Information has a factor of **usability** and **meaning** associated with it (Who, What, When and Where?)
- **Knowledge** means the appropriate collection of information that can make it be useful (How?).
- Knowledge can be assessed using time, experience, ethic values to generate **wisdom**. Using wisdom you can take a decision between the right and wrong, good and bad, or any improvement decisions. (Why)
- **Intelligence** is the ability to use the knowledge and wisdom gained.



DATA PYRAMID

Data: 7

Information: 7 degree C

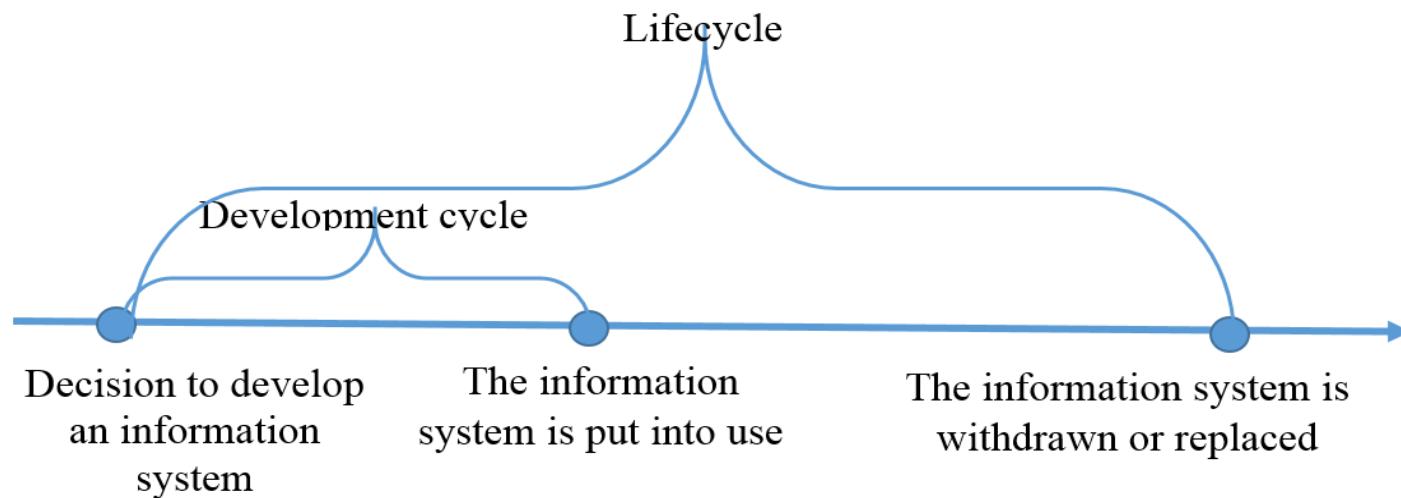
Knowledge: 7 degree C, today at 10.00 in Bucharest

Wisdom: I need to put on a thick coat when I leave the house.

Intelligence: Rules implemented by a mobile app that advises me what to wear depending on the weather conditions forecast (temperature, humidity, precipitation, wind)

Computer system lifecycle and development cycle

- The computer system **lifecycle** is a template used for ordering information system realization activities, covering the interval of time that begins with the decision to develop the information system and ends with the decision of withdrawal and replacement with a new system.
- The computer system **development cycle** is contained within the system lifecycle. It includes the time length that starts with the decision to develop the information system and it ends when the system is put into use and the maintenance process begins



Computer system lifecycle and development cycle

The activities belonging to the lifecycle of software products are grouped in many ways, in **stages** or **phases**. Here is such a grouping of activities:

- **User requirements specification**— it includes the identification and formulation of overall requirements regarding information system realization, as well as justification for its necessity and opportunity
- **Analysis** – it is the stage of analyzing the system functional and quality requirements, identifying the following elements: what are the functions to be met by the system, what data should be processed, what results must be obtained, what type of interface is to be used. Essentially, the analysis stage offers an answer to the question “**What should the system do?**” and it shouldn’t be concerned about the technology to be chosen for implementation. The *quality of results* is very important for this stage, as it represents a *bridge* between client requirements, architecture designs and implementation models that will be developed in the next stages
- **Design** – it essentially answers the question “**How will be implemented the requirements identified during analysis?**”, given the particular technology chosen for implementation. The design aims to: divide into modules and establish the system architecture, organize and structure data, design the algorithms needed for processing, design the user interface etc.

Information system lifecycle and development cycle

- **System implementation**
 - **System coding** - it is the effective writing of software programming code as specified in the design phase. Each of the application **modules** will be **individually implemented and tested**. At this level, the **overall integration and testing** assume that modules that have been implemented and tested during previous phase to be integrated, following the overall system testing to check the correctness of inter-modules relationships and the functionality of the system as a whole
 - **System deployment** and **user training**. Functioning in real conditions is particularly important as the system is validated using real data sets in real operating conditions.
- **System operation and support**

Within various lifecycle models, the stages presented above are included (totally or partially) in various combinations. There is a big variety of such models.

Strategies for automating activities with information systems (1)



- The strategies, approaches and techniques for computer system development have been continuously renewed and improved.
- At first, computer system development focused only on the use of databases and programming languages.
- Then, **commercial software components** and **packages** and **ERP systems** developed by software companies gradually entered the market and offered companies an alternative to full development of solutions, from scratch.
- Lately, organizations can use software without needing to install their own applications, by using Internet based **SaaS – Software as a Service solutions** .

Strategies for automating activities with information systems (2)



- There are two main strategies:
 1. System acquisition
 2. System development
 - Within organization
 - Externalization– the system is going to be developed by an external software company

1. System acquisition

- Is the first strategy that must be considered
- It involves that organization will use some **existing products**, with the possibility of configuration and customization.
- The main categories of existing software products that are purchased: commercial **software packages**, **ERP-type** integrated systems, **SaaS**.

Strategies for automating activities with information systems (3)



Commercial software packages

- They are available for sale or rent to the general public
- They usually address small and medium-size organizations
- They often have **limited customization features** to respond to specific requirements

ERP type integrated systems

- They facilitate **the integration of all business processes** of the organization functional units and manages connections with external organizations.
- They operate real-time;
- They have a unique database for all applications;
- They consist of a set of modules that can also run individually;
- They address all organization types.

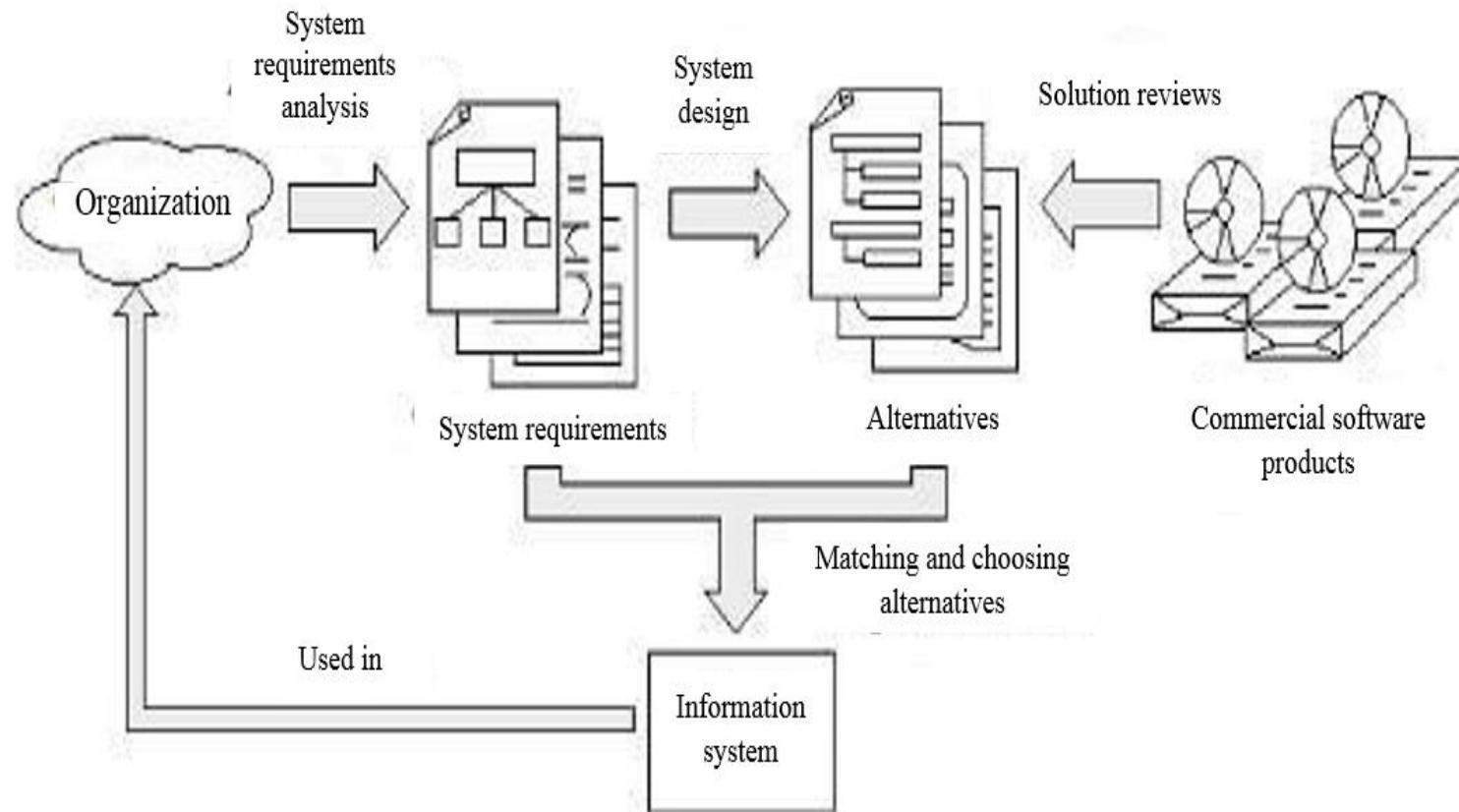
Strategies for automating activities with information systems (4)



Software as a Service (SaaS)

- It is a way to provide software which involves that **applications** and their related **data** are centrally **stored by the service provider** and are usually accessed by clients via Internet using a web browser.
- They can support **configuration**, less **customization**;
- They can be **quickly updated**;
- Many applications offer users functions for collaboration and information sharing;
- They are hosted **in cloud**, so the response time and security issues are critical factors.

Automation by system acquisition

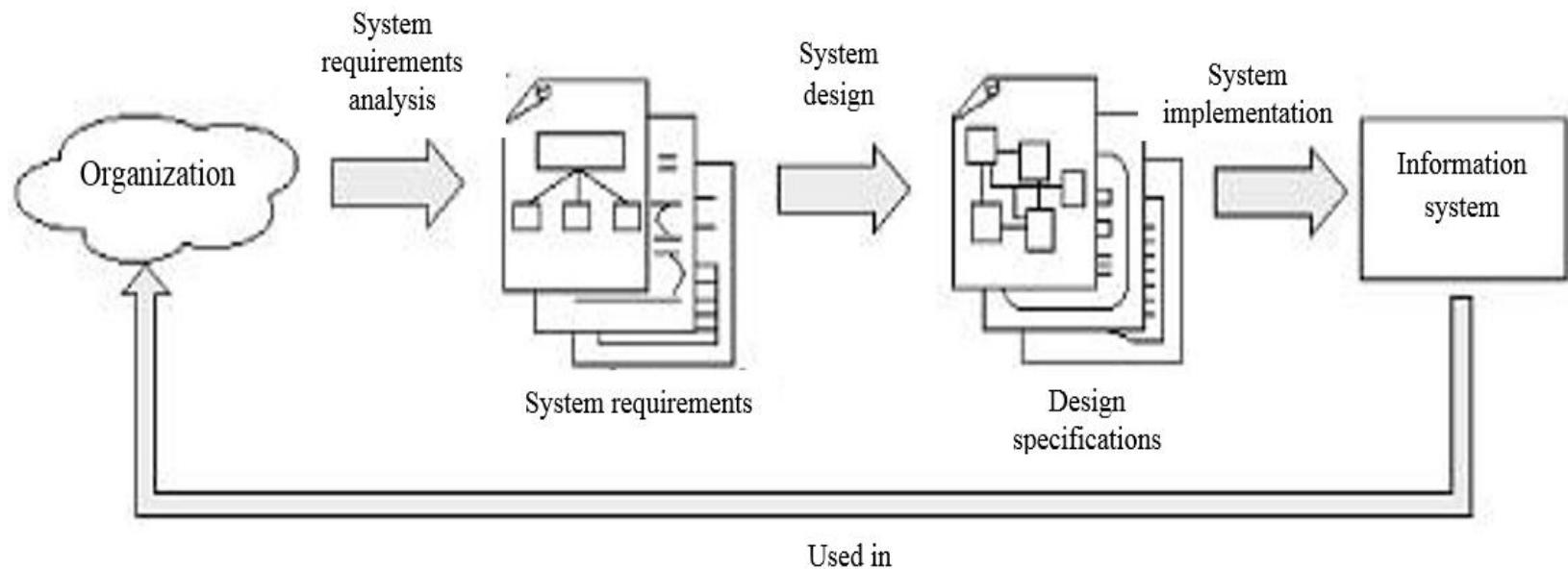


Strategies for automating activities with information systems (5)

2. System development

- Can be developed **within organization** or can be **externalized**.
- Generally, it is used for **specific, unique requirements** of the organization
- This is the method adopted by the software and information technologies developers
- It is a **time and resource consuming** solution.
- It involves **performing all the steps** of a system lifecycle.

Automation by system development



2nd Lecture – Identification of computer system requirements

Agenda

- ✓ Requirements engineering
 - ✓ Classes of computer system requirements
 - ✓ Non-functional requirements
 - ✓ Techniques for requirement identification
 - ✓ Quality features of the requirements
-
- ✓ UML use case diagram
 - ✓ Use case description using a template



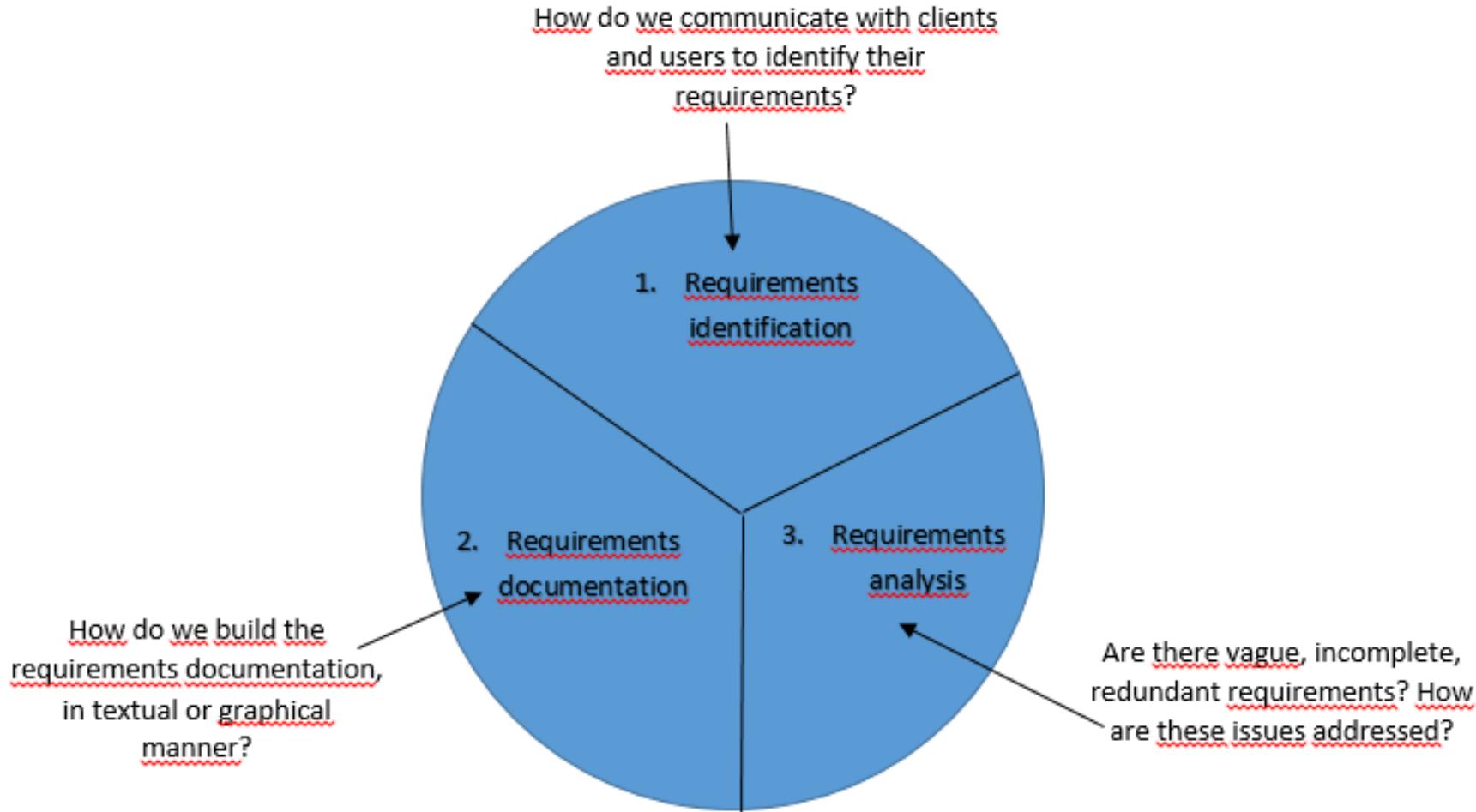
The background

- Poor quality requirements is consistently **ranked first** in the hierarchy of causes leading to the failure of software projects.
- One explanation: development teams **allocate too little time** to understanding:
 - *real problems of business,*
 - *user needs or*
 - *nature of the environment* in which the system will run.
- A second explanation: **changing requirements**

Requirements engineering (1)

- This requires the discovery, understanding, specifying and analyze of the following components:
 - **WHAT** problem should be solved;
 - **WHY** the problem needs to be solved;
 - **WHO** is involved and is going to be responsible with this problem solving.
- These three components constitute the foundation of ***Computer System Requirement Engineering***

Requirements engineering (2)



Core activities for building a requirement model

Requirements engineering (3)

- *Business logic* is the defining element for the modeling and automation process. It includes both **business rules** and the **business workflow** (processes) that describes the way of transferring documents and data from one participant (individual or software system) to another.
- In developing information systems, business logic aims at:
 - *Modeling* real world **business objects** (such as inventories, customers, products);
 - Managing **business object storage solutions** (business objects mapping into the database tables);
 - Describing *how business objects interact with each other*.

Requirements engineering (4)

- In developing information systems, a **requirement** defines **an objective that it must be fulfilled**, in response to the needs of its users.
- It is part of the general purpose for which the computer system is developed.
- The requirements also dictate the way the system should respond to user interaction.
- Generally, developers should consider the following aspects related to the requirements system :
 - use of a **technical language**: requirements must always be specified using the **user's language**. The **jargon** of the analyzed field can also be listed and analyzed;
 - **relationship with the business goals**: each requirement must be clearly linked to business objectives.

Types of information systems requirements (1)

- Requirements engineering process aims to collect, develop, correct and adjust **large volumes of specifications** which **differ** in terms of **purpose** and **mode of expression**. Differentiations can be made between:
 - *Descriptive requirements*;
 - *Prescriptive requirements*.
- *Descriptive requirements* display properties the system should have, regardless of how it will work. Such properties are generated usually by laws of nature or physical constraints.
Examples of descriptive specifications:
 - ❑ Same book can not be borrowed by two subscribers simultaneously.
 - ❑ If a hotel room is under renovation, then it can not be occupied.

Types of information systems requirements(2)

- *Prescriptive requirements* display some properties the system should have, that will be or will be not met by the system, depending on the way the system will work. Examples of prescriptive specifications:
 - A subscriber cannot borrow more than three book at the same time
 - Goods have to be delivered within the time interval specified by the buyer.
 - Distinction between descriptive and prescriptive requirements is very important in the context of requirements engineering, as **we can negotiate, change or find alternatives to prescriptive specifications**, while for the descriptive specifications **these things are not possible**.

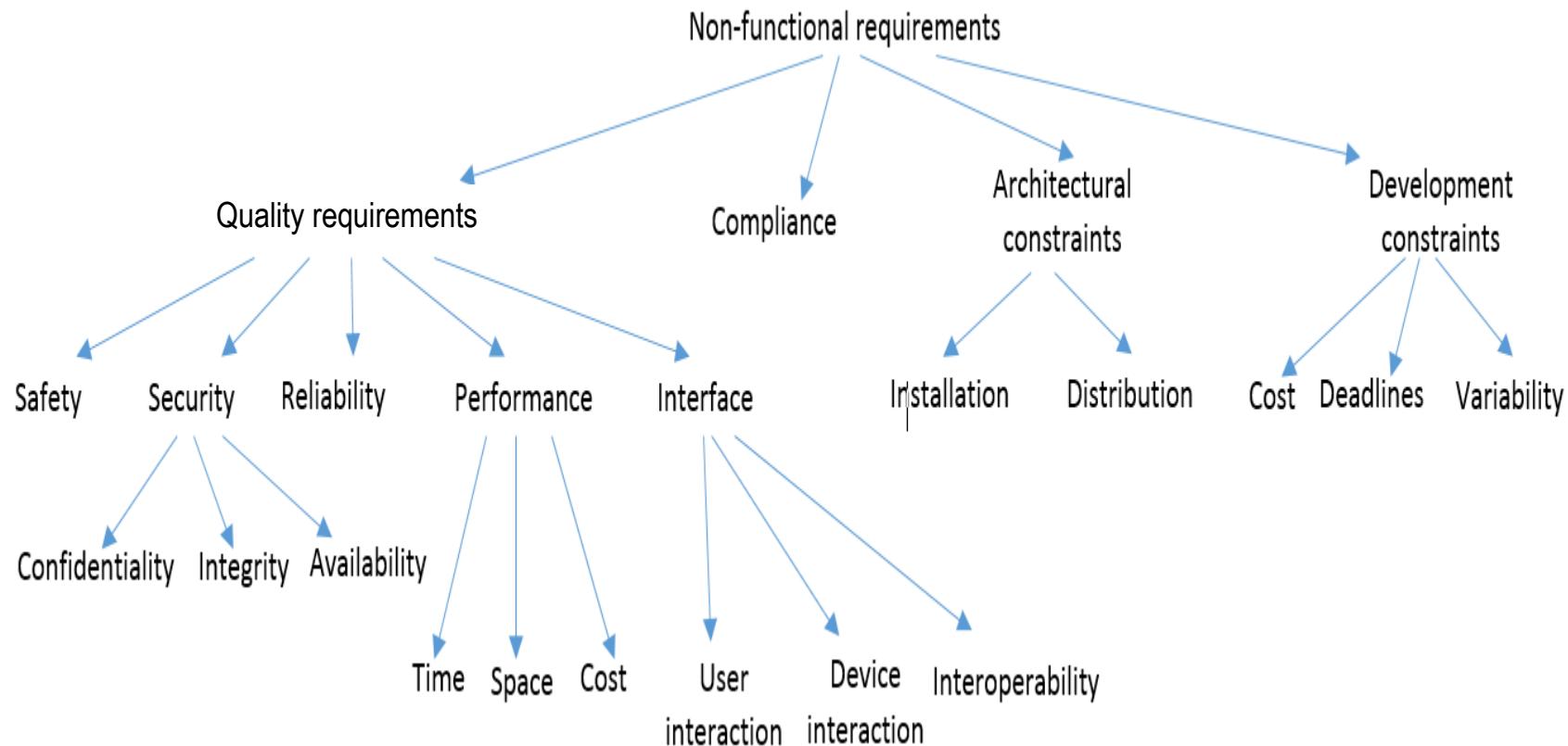
Types of information systems requirements (3)

- Considering their functionalities, requirements can be:
 - *functional*;
 - *non-functional*.
- **Functional requirements** define the functions of a computer system or of its components, through a set of *inputs*, *behavior* and a set of *outputs*. This category includes:
 - computations,
 - technical details,
 - information regarding data processing and manipulation,
 - functionalities, for example, the way a use case has to be built.

Types of information systems requirements (4)

- Non-functional requirements capture criteria that can be used to analyze aspects of the **system operation** and **not its behavior**.
- These **impose constraints** on the functional requirements at design or implementation level (e.g. performance, security or reliability requirements)
- Non-functional requirements are usually referred as ***qualities of the system***, but also as ***quality attributes***, ***quality objectives***, ***quality features*** or ***constraints***.

Non-functional requirements



Types of non-functional techniques

Non-functional requirements

A. **Quality requirements** define issues related to "how well" the system should operate. These include specifications related to:

- **Safety requirements** are quality requirements for preventing occurrence of accidents or environmental degradation.
- **Security requirements** describe system safeguards against undesirable behavior.
 - *Confidentiality requirements* indicate that certain information can not be disclosed to unauthorized parties.
 - *Integrity requirements* indicate that some information can be changed if this was done in a fair and authorized manner.
 - *Availability requirements* indicate that certain requirements and resources can be used by authorized persons when necessary.

Non-functional requirements

A. Quality requirements (continued)

- **Reliability requirements** constraint the computer system to work as expected and desired for long periods of time. Aside from exceptional circumstances, the system must provide services in a fair and robust manner.
- **Performance requirements** put conditions on the system operation manner, for example the maximum time required for executing an operation.
- **Interface requirements** indicate the way the computer system interacts with the environment, the users, and with other systems.

Non-functional requirements

- B. **Compliance requirements** impose the necessary conditions so that the system operates **in accordance with** national laws, international regulations, social norms, political and cultural constraints, standards etc.
- C. **Architectural requirements** impose **structural** constraints on the computer system, so that it can work properly in the environment where it will be implemented. It provides developers with guidance on the **suitable type of architecture** for the system.
- D. **Development requirements** are non-functional requirements that impose **the way the computer system should be developed**, and not the way the system should meet the functional requirements (time limitations, resource availability).

Techniques for requirements identification

1. Interviews

- Interviewing the beneficiary is **the most popular method** for requirement identification. Its success depends on the **involvement of all stakeholders**, including managers, stockholders, employees, etc. that will further be called **general users**.
- Focus on activities of the user within the organization
- The interview can reveal **new requirements or conflicting requirements**.

Techniques for requirements identification

1. Interviews - continued

- Conflicting requirements : confusing



The analyst may ask himself the following pertinent question : *"How can a business be competitive and profitable if there is no consensus, at least within it, on how it works?"*



A possible answer would be: *"...the detail level necessary for building a computer system is higher than the detail level necessary for successfully running a business."*

Techniques for requirements identification

2. Join Requirement Planning Sessions

- Join Requirement Planning - JRP can be equivalent with conducting **interviews of all users** (or at least a substantial part thereof) at the same time and in the same room.
- All people that shape the direction of system development are **brought together in one place** to provide details about what the system should do.
- There should be present:
 - a **mediator** to moderate these sessions, but also
 - a **person to document user proposals**, using, for example, a video projector and a visual modeling software.

Techniques for requirements identification

3. Use cases

- Use cases describe **interactions** between users and computer system and show **what** users should do with the system by identifying the **important functions**.
- A use case specifies a set of **interactions between system and external actors** (such as persons, hardware devices or other software systems), including also **versions or extensions** of the main behavior of the system.
- Use cases can be **one of the first forms of representation of the requirements** of a computer system.

Techniques for requirements identification

4. Social observation and analysis

- Observation based methods imply the existence of **an observer** that watches the system users and records information on the work they perform.
- Social observations can be
 - **direct** observations
 - **indirect** observations (video)
- Advantages: facilitate the collection of **quality data** and it is useful for analyzing **real processes and activities**.

Techniques for requirement identification

5. Prototyping

- A **prototype** is useful for **the end user** who will understand better what he wants or what is expected from the product. It is also useful for **developers** to test some techniques, developed algorithms, interfaces etc.
- There are two approaches:
 - **Testing/trial prototype** – it must be **fast**, if **not perfect**, it can be used to validate the interface, to customize the architecture to include requirements as well as possible, or to validate specific algorithms;
 - **Evolutive prototype** – it develops into the final product in order to be able to consider **all the quality features** of the final software product; Generally this prototype may not be fast, but can improve the model by ensuring **high quality of the software**.

Requirements in agile methodologies (SCRUM)

- The **user story** represents requirements in a simple written **narrative** told from the user's perspective, rather than a comprehensive document.
- User stories enable developers to **estimate and plan work**. User stories change and mature throughout the software development lifecycle.
- Developers compile the many user stories into a **product backlog**, i.e. a list of all requirements, ordered by users, according to **priorities**
- A **sprint backlog** is a list of tasks that the team must execute in a "sprint"/cycle (1-2 weeks).
- A "sprint" includes as stages: requirements identification, analysis, design, development and testing. At the end of a "sprint" there is a deliverable (eg a report/dashboard).

Requirements in agile methodologies – improvements (1)

1. Supplement user stories

- Complex or mission-critical projects especially demand more detail
- Some projects must have documentation to meet compliance or process standards



- Use cases
- Decision tables

2. Involve stakeholders regularly

- Projects can fail when stakeholders and developers fail to communicate.
- Stakeholders are the experts in a project's requirements



- Stakeholders: make suggestions and model and document requirements
- Developers: build and adjust the product

3. Prioritize requirements

- Developers estimate the time and money necessary to implement a requirement
- The list of requirements and estimates for the iteration = a stack of index cards



- The owner +developers prioritize the list of requirements
- The higher the priority the requirement, the more detailed the item should be

Requirements in agile methodologies – improvements (2)

4. Focus on executable requirements

- Test-first design process: model requirements, write code, and then refine and refactor it to implement
- Executable requirements focus on *what* something needs to do, and *how* that thing should work



Work through executable models upfront, to identify cross-requirement dependencies

5. Think layers, not slices

- Smaller requirements and feature sets -> user stories that are easier to estimate, prioritize and build.
- Larger, more complex requirements tend to create dependencies



- Compose user stories considering the software stack: the front end, middleware, back end and database

6. Prototype and update

- Prototyping is a useful practice to test ideas and encourage discussion with stakeholders



- Developers can update the prototype to refine the software.
- The resulting code often is usable for the build

Use case UML diagram

- Its role is to represent in graphical format the **functionalities** that have to be met by the computer system in the final stage.
- The model developed by use case diagrams and documents that briefly describe each use case is called **REQUIREMENTS MODEL**.
- Use case diagrams are made up of **actors** and **use cases**, on one side, and the **relationships** between them, on the other side.

Introduction

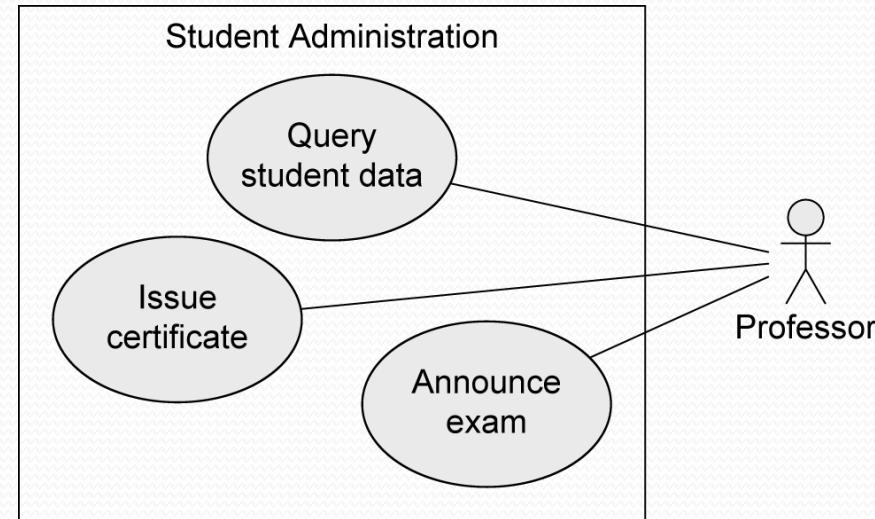
- The use case is a fundamental concept of many object-oriented development methods.
- Use case diagrams **express the expectations of the customers/stakeholders**
 - essential for a detailed design
- The use case diagram is used during the entire analysis and design process.
- We can use a use case diagram to answer the following questions:
 - **What** is being described? (The system.)
 - **Who** interacts with the system? (The actors.)
 - **What** can the actors do? (The use cases.)

Example: Student Administration System

- **System**
(what is being described?)
 - Student administration system

- **Actors**
(who interacts with the system?)
 - Professor

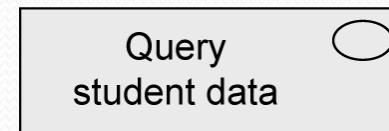
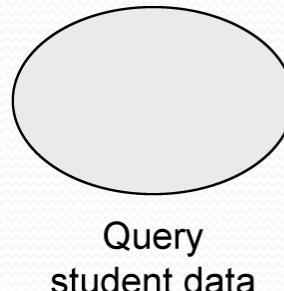
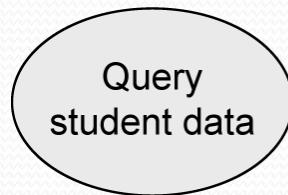
- **Use cases**
(what can the actors do?)
 - Query student data
 - Issue certificate
 - Announce exam



Use Case

A

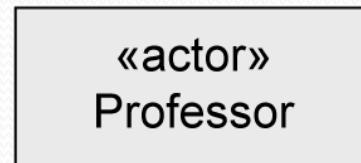
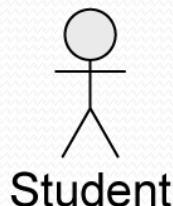
- Describes **functionality expected** from the system under development.
- Provides **tangible benefit** for one or more actors that communicate with this use case.
- Derived from collected **customer requirements**.
- Set of all use cases describes the functionality that a system shall provide.
 - Documents the functionality that a system offers.
- Alternative notations:





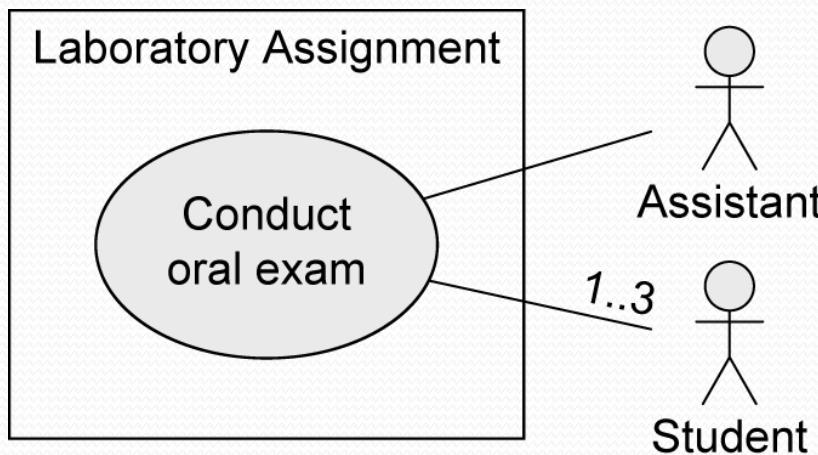
Actor (1/3)

- Actors interact with the system ...
 - by **using** use cases,
i.e., the actors initiate the execution of use cases.
 - by **being used** by use cases,
i.e., the actors provide functionality for the execution of use cases.
- Actors represent **roles** that users adopt.
 - Specific users can adopt and set aside multiple roles simultaneously.
- Actors are not part of the system, i.e., they are outside of the system boundaries.
- Alternative notations:



Actor (2/3)

- Usually **actor data** is also administered within the system. This data is modeled within the system in the form of objects and classes.
- Example: actor **Assistant**
 - The actor **Assistant** interacts with the system **Laboratory Assignment** by using it.
 - The class **Assistant** describes objects representing user data (e.g., name, ssNr, ...).



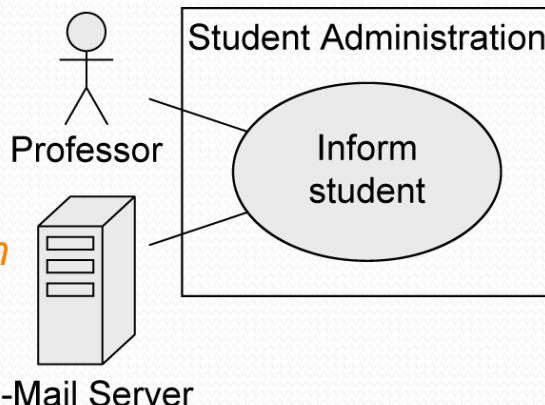
Actor (3/3)

- **Human**
 - E.g., **Student, Professor**
- **Non-human**
 - E.g., **E-Mail Server**
- **Primary**: has the main benefit of the execution of the use case
- **Secondary**: receives no direct benefit
- **Active**: initiates the execution of the use case
- **Passive**: provides functionality for the execution of the use case

Example:

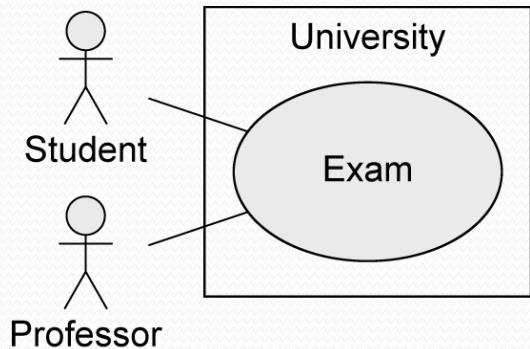
*Human
Primary
Active*

*Non-human
Secondary
Passive*



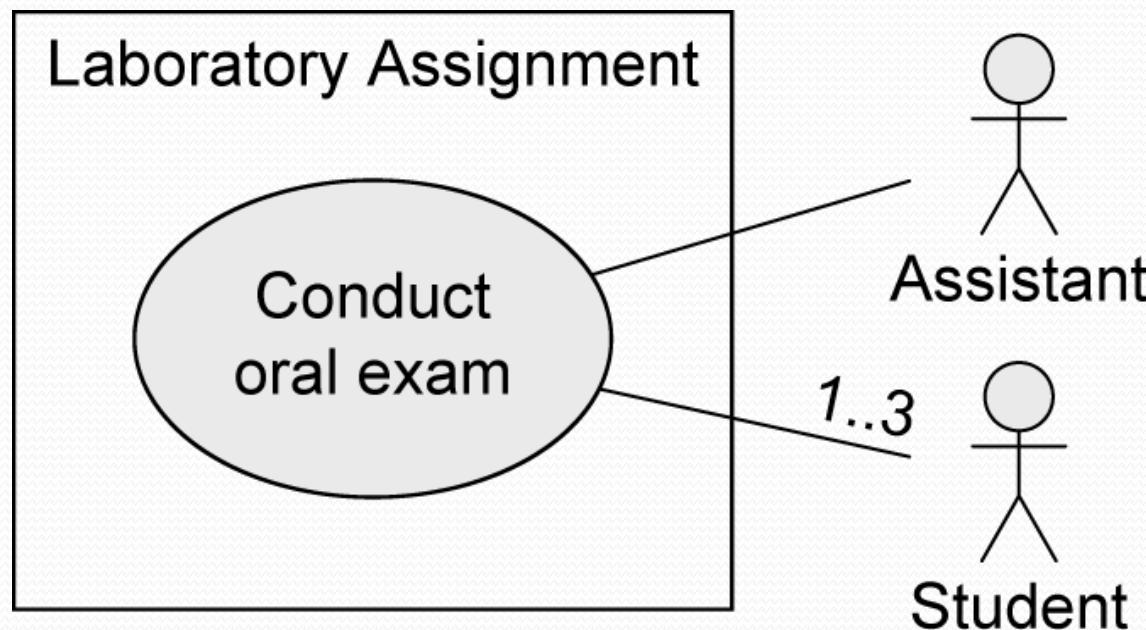
*Human
Primary
Active*

*Human
Secondary
Active*



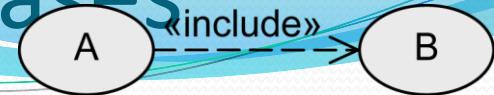
Relationships between Use Cases and Actors

- Actors are connected with use cases via solid lines (*associations*).
- Every actor must communicate with at least one use case.
- An association is always binary.
- **Multiplicities** may be specified.

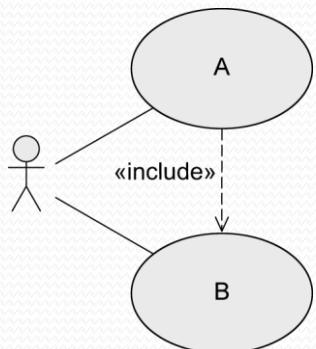


Relationships between Use Cases

«include» - Relationship



- The behavior of one use case (included use case) is integrated in the behavior of another use case (base use case)



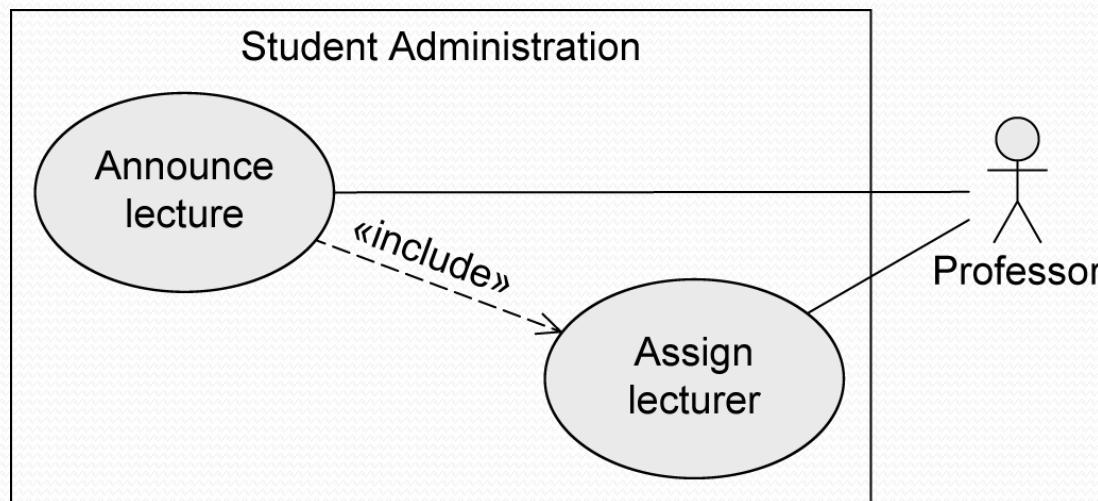
Base use case

requires the behavior of the included use case to be able to offer its functionality

Included use case

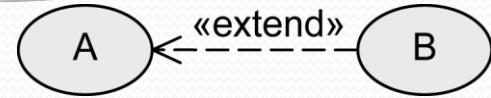
may be executed on its own

- Example:

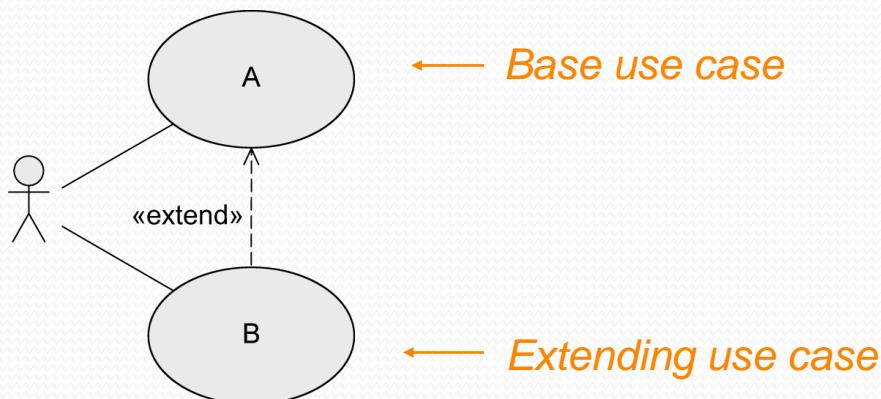


Relationships between Use Cases

«extend» - Relationship



- The behavior of one use case (extending use case) may be integrated in the behavior of another use case (base use case) but does not have to.
- Both use cases may also be executed independently of each other.

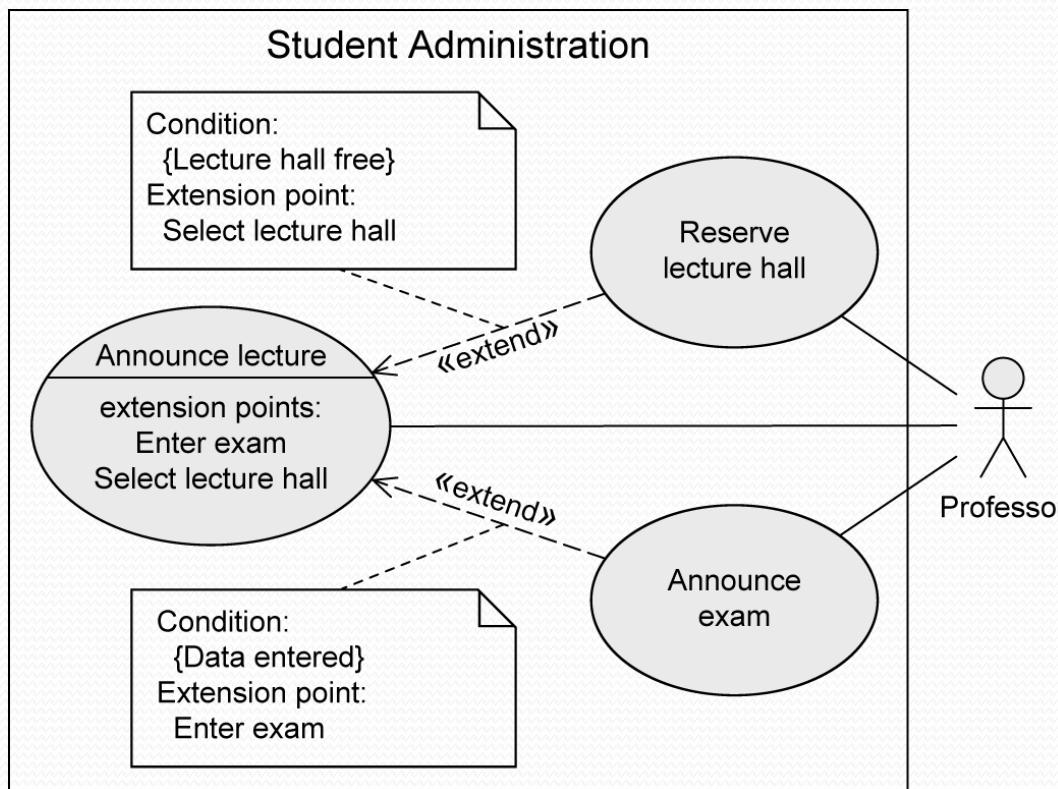


- **A decides** if B is executed.
- **Extension points** define at which point the behavior is integrated.
- **Conditions** define under which circumstances the behavior is integrated.

Relationships between Use Cases

«extend» - Relationship: Extension Points

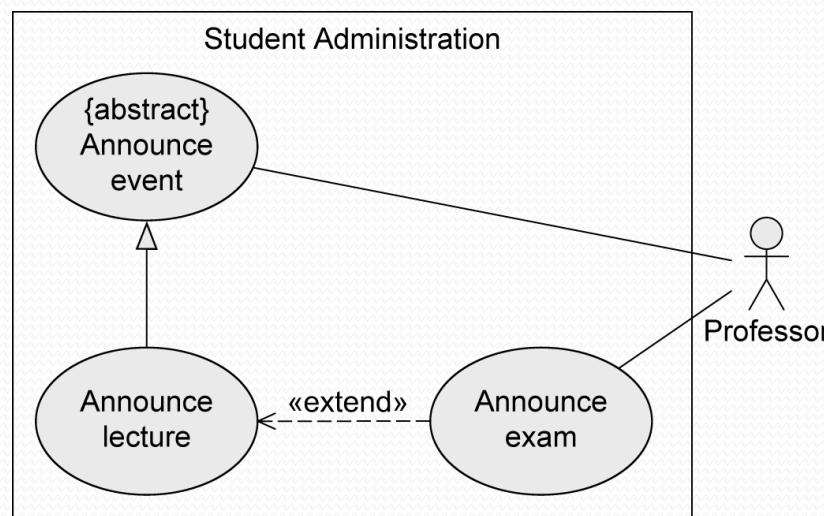
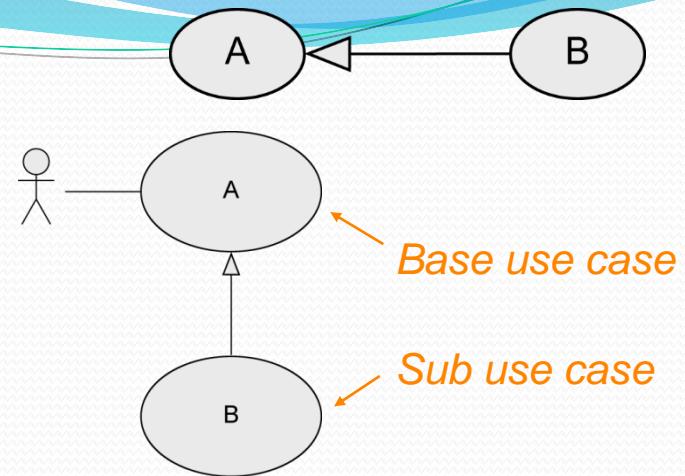
- Extension points are written directly within the use case.
- Specification of multiple extension points is possible.
- Example:



Relationships between Use Cases

Generalization of Use Cases

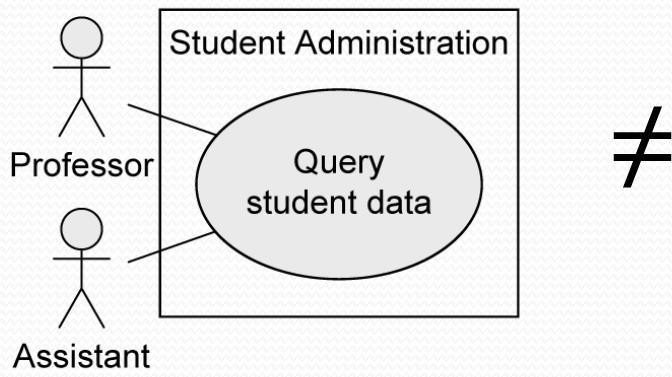
- Use case **A** generalizes use case **B**.
- **B** inherits the behavior of **A** and may either extend or overwrite it.
- **B** also inherits all relationships from **A**.
- **B** adopts the basic functionality of **A** but decides itself what part of **A** is executed or changed.
- **A** may be labeled **{abstract}**
 - Cannot be executed directly
 - Only **B** is executable
- Example:



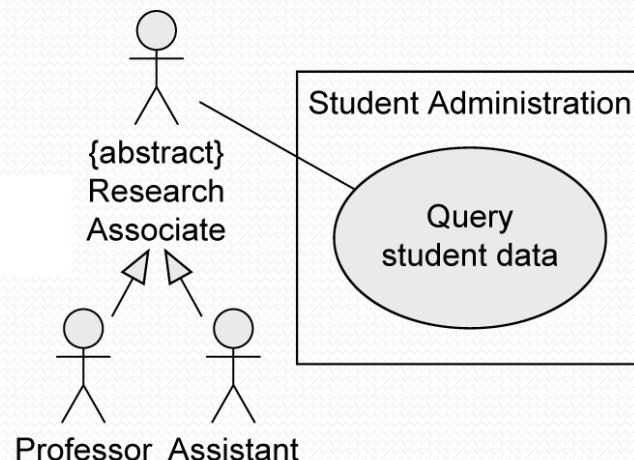
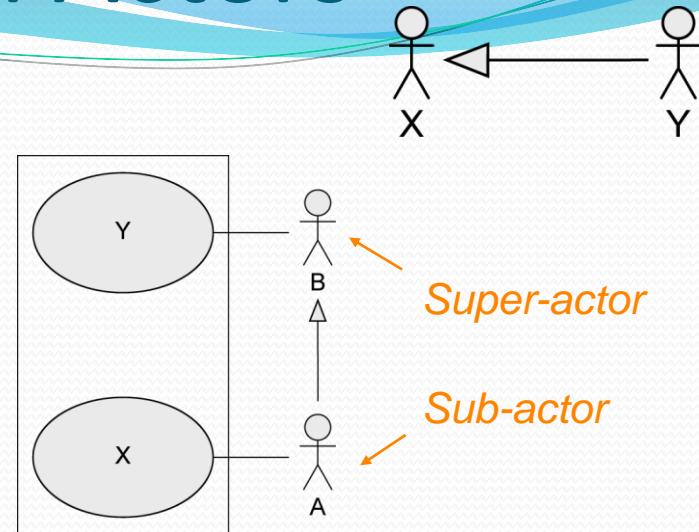
Relationships between Actors

Generalization of Actors

- Actor **A** inherits from actor **B**.
 - A** can communicate with **X** and **Y**.
 - B** can only communicate with **Y**.
 - Multiple inheritance* is permitted.
 - Abstract* actors are possible.
-
- Example:



Professor AND Assistant needed
for executing **Query student data**



Professor OR Assistant needed
for executing **Query student data**

Description of Use Cases

- Structured approach
 - **Name**
 - **Short description**
 - **Precondition**: prerequisite for successful execution
 - **Postcondition**: system state after successful execution
 - **Error situations**: errors relevant to the problem domain
 - **System state** on the occurrence of an error
 - **Actors** that communicate with the use case
 - **Trigger**: events which initiate/start the use case
 - **Standard process**: individual steps to be taken
 - **Alternative processes**: deviations from the standard process

[A. Cockburn: Writing Effective Use Cases, Addison Wesley, 2000]

Description of Use Cases - Example

- Name: **Reserve lecture hall**
- Short description: An employee reserves a lecture hall at the university for an event.
- Precondition: The employee is authorized to reserve lecture halls.
- Postcondition: A lecture hall is reserved.
- Error situations: There is no free lecture hall.
- System state in the event of an error: The employee has not reserved a lecture hall.
- Actors: **Employee**
- Trigger: Employee requires a lecture hall.
- Standard process:
 - (1) Employee logs in to the system.
 - (2) Employee selects the lecture hall.
 - (3) Employee selects the date.
 - (4) System confirms that the lecture hall is free.
 - (5) Employee confirms the reservation.
- Alternative processes: (4') Lecture hall is not free.
 - (5') System proposes an alternative lecture hall.
 - (6') Employee selects alternative lecture hall and confirms the reservation.

Importance of use case diagrams

- All processes that have to be carried on by the system are found in the form of a use case. Processes are then described textually or by a sequence of steps. For graphical modeling of scenarios, **activity diagram** can be used.
- Once the system behavior has been depicted, the use cases are further analyzed to identify how objects interact to model this behavior. **Sequence diagrams** and **communication diagrams** are used for modeling object interactions.
- Use case diagram is also used for testing whether the **system is consistent with the initial requirements**. It takes up all use cases to see if the system meets customer requirements.

Exemple of scenario

The project goal is to develop a software application for the **management of a hotel business unit**. In order to check in, a **customer** can request to reserve one or more rooms by e-mail or telephone. For this, he provides the **receptionist** with information on the period of accommodation and type of rooms required. Customers will get **discounts** if they reserve at least 3 rooms or if the period of accommodation exceeds 5 days. The receptionist checks availability and notifies the client of this and the estimated cost of accommodation. If there are no rooms available as requested, the receptionist can provide alternatives to the customer. The client may request a discount (additional or not) and the receptionist will decide the feasibility discount, assisted mandatory by the hotel manager. If the client agrees with the proposed price, they proceed to the reservation. For new customers, the receptionist asks **identification data**, which he introduces in the application.

Once at the hotel and if he has made a prior booking, the customer will provide his identification and / or booking number and the **check in** is finalized. If there is no reservation, the availability for the required period will be checked. When there is such a room, **accommodation** is made. At the end of the stay, the receptionist prepares **a list of all the services** used by the customer and their price. The list must be validated by the customer, then the **final invoice** is drawn up. The invoice can be **paid** partially or fully by bank transfer, cash or using a credit card. Also, before leaving the hotel, the customer is asked to complete **a form to evaluate** the services provided by the hotel premises

General requirements for hotel accommodation activity

The project aims at developing a computer system for a hotel that would provide support for its activities.

The **core activity** of the hotel is providing **accommodation, catering services and other services** of commercial agents affiliated to the hotel.

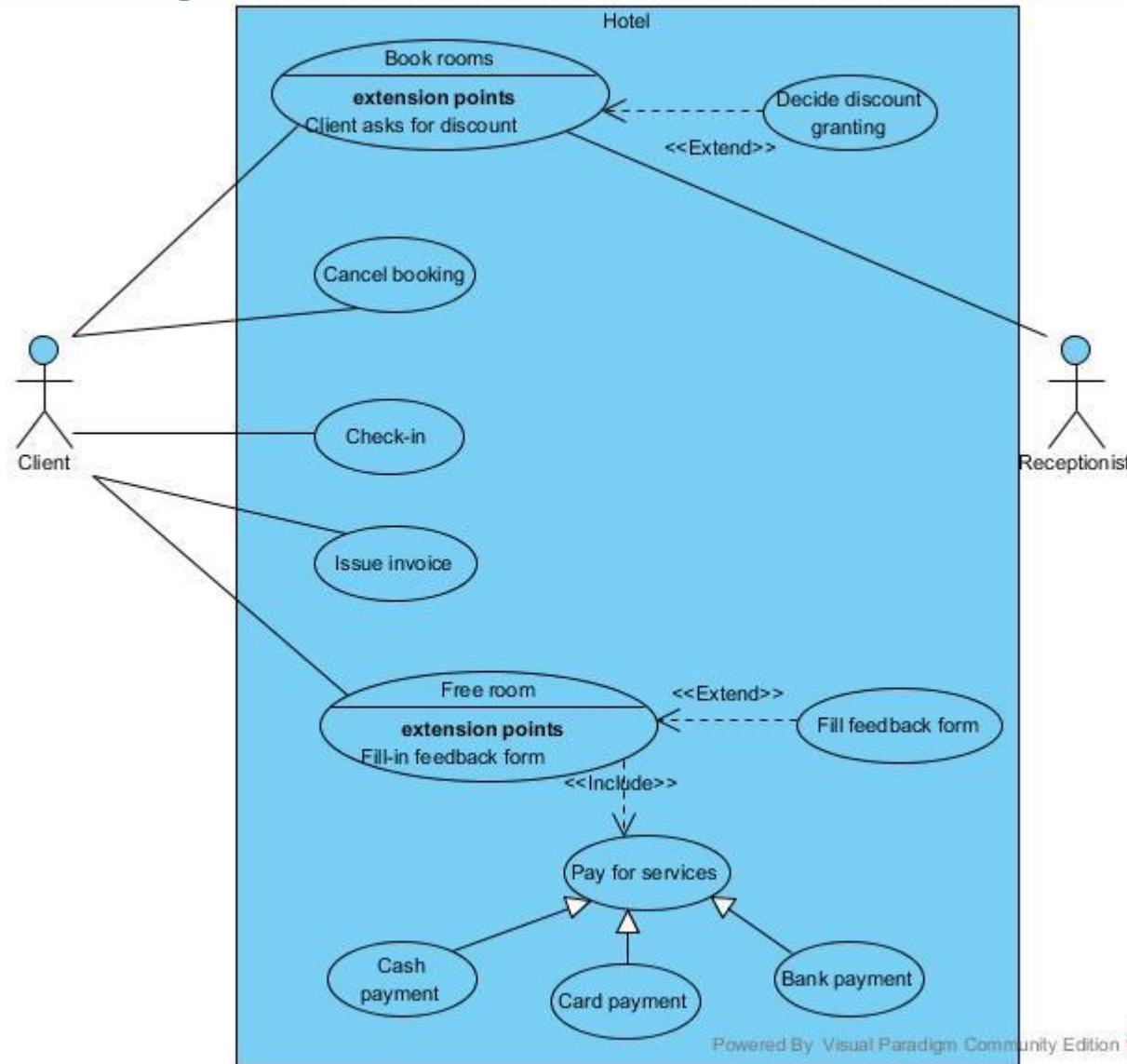
The IT system should achieve: **booking rooms** and **customer registration, management of room occupancy**, and **other services** used by customers.

The arrangements for booking, customer registration and corresponding payments derive from the marketing and management policy of the hotel.

The **basic requirements** that should be accomplished are:

- Reducing the time to fill in the forms for reservation and registration by automating the process of updating availability and allocation of rooms;
- Monitoring staff involved in bookings, registrations, and various changes or cancellations.
- Reducing time spent making lists of rooms to be cleaned, lists of arrivals, lists of room status;
- Increased promptness in responding to customer about the availability of the accommodation requested for a specific period;
- Creating a secure and effective system that protects against overbooking.

Use case diagram - example



Use case element	Description
Code	CU01
State	Draft
Scope	Manage hotel accommodation activity
Name	Book rooms
Primary actor	Client
Description	It involves booking for one or more rooms
Preconditions	The receptionist is logged in the system
Postconditions	The booking was accomplished and the client receives booking confirmation.
Trigger	Client asks for booking one ore more rooms by email or by phone.
Main course	<ol style="list-style-type: none"> 1. The client provides the receptionist with information on the period of accommodation and type of rooms required 2. The receptionist checks room availability. 3. The receptionist notifies the client that there are available rooms. [A alternate course: There are not available rooms according to client requirements] 4. The receptionist informs the client of the estimated costs. [Extention point: CU07 Decide discount granting] 5. The client confirms booking period and agrees estimated cost. [B alternate course: the client doesn't agree the booking conditions] 6. The receptionist asks the customer identification data. [C alternate course: customer data are already in the system] 7. The receptionist enters customer data in the system. 8. The receptionist makes room booking . 9. The client receives the confirmation of booking.
Alternate courses	<p>A: 1. The receptionist provides booking alternatives to the client. 2. The client selects an alternative and the scenario ends.</p> <p>B: 1. the client does not confirm booking and the scenario ends.</p> <p>C: 1. Go to step 8.</p>
Relationships	Extended by CU07 Decide discount granting
Frequency of use	Very often
Business rules	When the client asks for discount, the receptionist can grant it only if manager agrees with it .

Exemples of non-functional requirements

- Quality requirements

- i. Safety

- The system should stop working if the outside temperature drops below 4 degrees C.
 - The system should stop working in case of fire.
 - The system should stop working in case of obvious attacks.

- ii. Security

- Data access **permissions** can only be changed by the **system administrator**.
 - All system data must be **copied** every 24 hours and backups must be stored in a safe place that is not in the same building as the system.
 - All external communications between data server and client must be **encrypted**.

- iii. Reliability

- The system must have an availability of 999/1000 or 99%. This is a requirement of reliability which implies that out of every 1,000 service requests, 999 must be met.

- iv. Performance

- The system will process at least X transactions per second.

Exemples of non-functional requirements

- **Compliance requirements**

- Legal and regulatory requirements: all changes to user data must be registered and stored at least 6 years.
- License requirements:
 - “Update order” process will be licensed for 300 concurrent users.
 - PostalCode_Address file will be licensed for a year, starting each April.

- **Architectural requirements**

- Data persistence will be ensured by a relational database.
- This database will be Oracle 12g.
- The system will run 24 hours a day, 7 days a week..

Question no. 1

Which of the following use cases are correct use cases when you want to design a use case diagram for an online book shop? Select one or more:

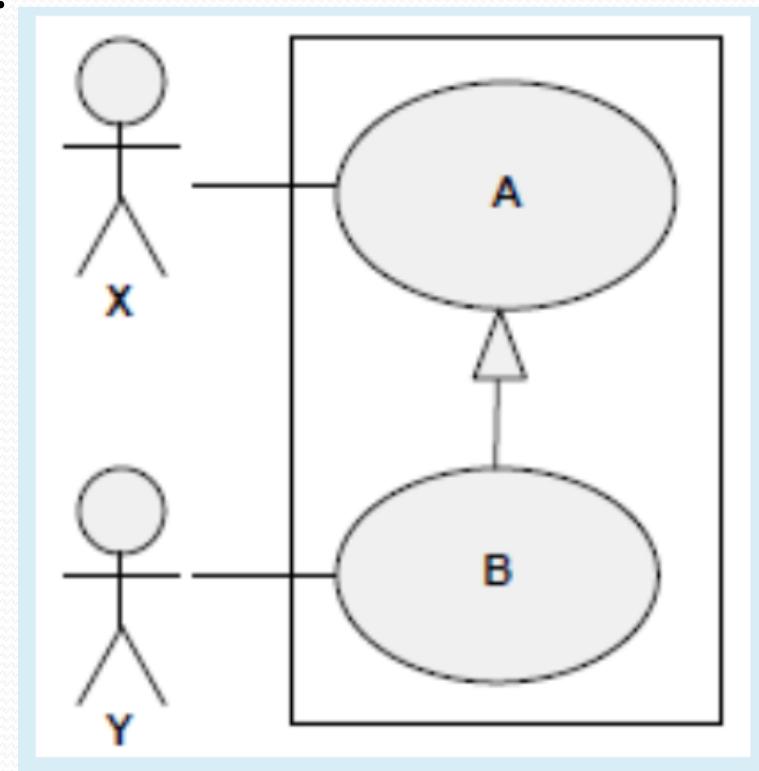
- a. Look for a book
- b. Order a book
- c. Cancel order
- d. Do not order a book
- e. Enter book title
- f. Login
- g. Enter credit card details

Question no. 2

- The following Use Case Diagram was modeled according to UML2 standard. Which combinations of actors communicate with Use Case A?

Select one or more:

- a. X
- b. Y
- c. X \wedge Y
- d. Y \wedge Y
- e. X \wedge X
- f. X \wedge X \wedge X
- g. X \wedge X \wedge Y



Question no. 3

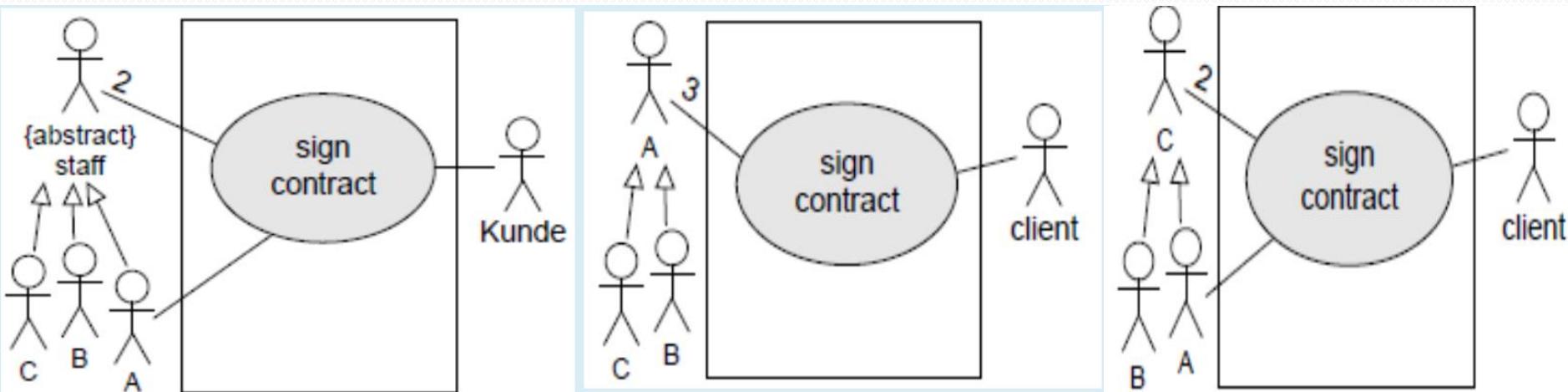
- Which of the following statements characterize use cases?
- Select one or more:
 - a. Use cases specify the functionalities and the behavior that the system which is being developed should have.
 - b. Use cases specify the procedural process within a system.
 - c. Use cases are derived from the clients' requests.
 - d. Use cases are suitable for modeling interfaces between two systems.

Question no. 4

- Which of the following questions are useful when identifying the involved actors of a use case diagram?
- Select one or more:
 - a. Who needs the system's support for their everyday work?
 - b. Who or what is interested in the systems' results?
 - c. Which hardware is available for the users of the system?

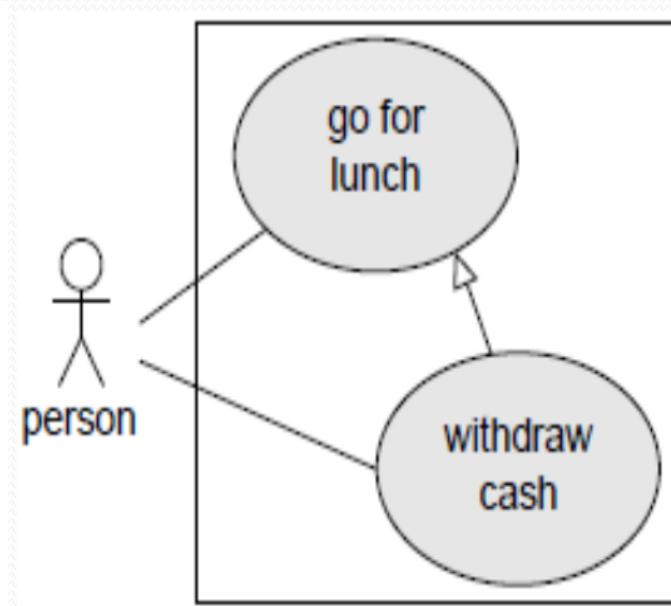
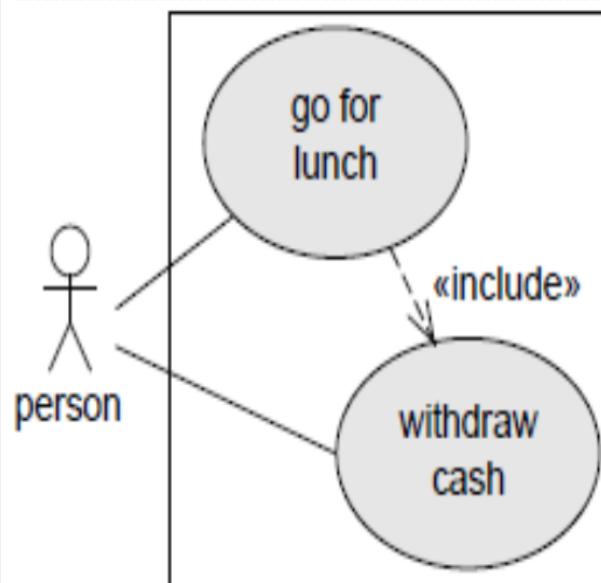
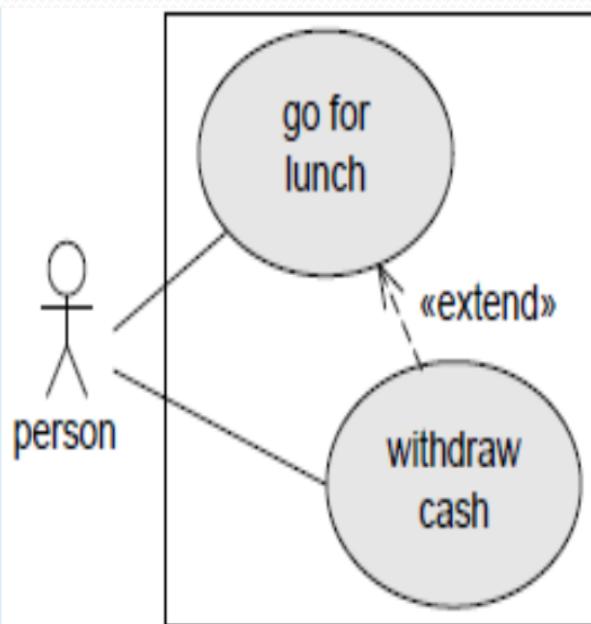
Question no 5

- There are 3 different types of staff members namely A, B and C. For a valid contract, the client, one staff member of Type A and two other staff members of either type (A, B or C) have to sign the contract.
(Possible employee combinations: AAA, AAB, AAC, ABB, ACC, ABC)



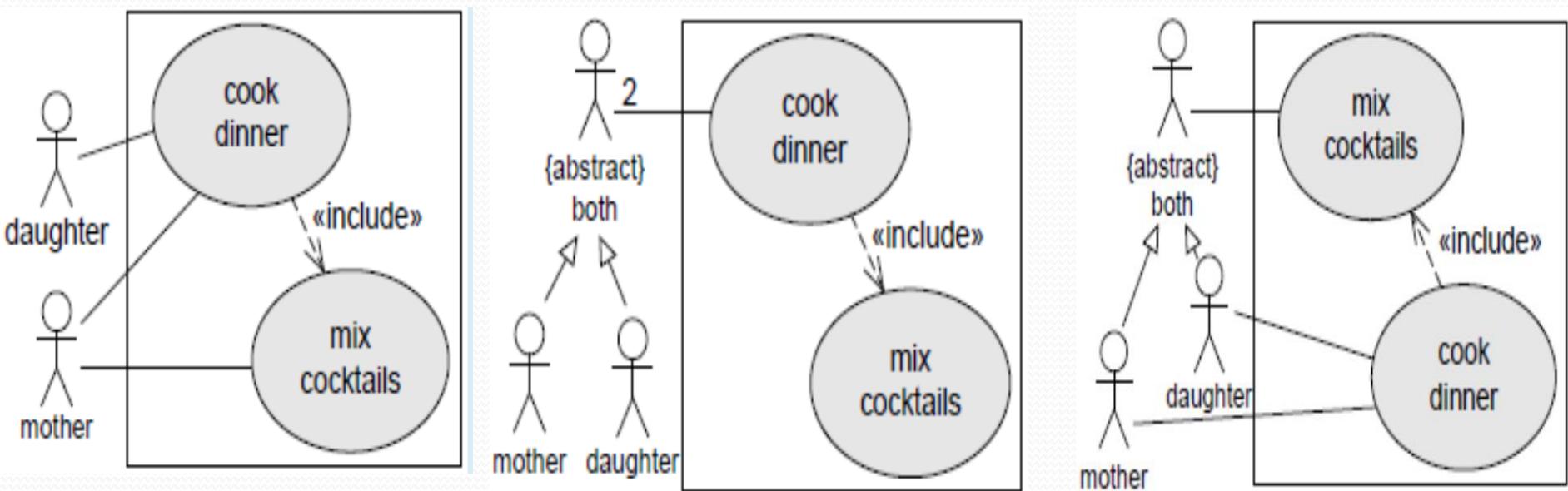
Question no. 6

- A person goes for lunch. In the course of that it might be necessary that the person withdraws cash from an ATM. Select one or more:



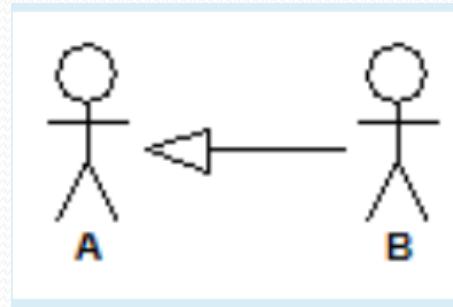
Question no. 7

- A mother cooks dinner together with her daughter. In the course of that, the mother also always has to mix the cocktails. Select one or more:



Question no. 8

- Which of the following statements about the given diagram clipping are true? Select one or more:



- a. A can execute the same use cases as B.
- b. A inherits all of B's associations.
- c. B inherits all of A's associations.
- d. B can execute the same use cases as A.

Information system design

Lecture 4

Prof. Ramona Bologa

Agenda

1. Information system analysis
2. Class diagram
3. Object diagram

5th Lecture - Object oriented analysis of computer systems

- During **system analysis stage**, **specifications** and **use cases** are analyzed identifying the most **important concepts** the system works with and their **relationships**.
- They are graphically represented by a **structure diagram** of the classes for the analyzed system.
 1. It is initiated **class diagram** representation, that will be finished in the following stages. Class diagram represents graphically the **static structure** of the system, including identified classes, packages and their relationships. A **package** may contain classes, interfaces, components, collaborations, use cases, diagrams or other packages.
 2. It is initiated **object diagram** building, that models **instances** of the elements contained in class diagrams. These diagrams contain a set of objects and the relationships between them at some point.

Object oriented analysis of computer systems

3. To highlight the statuses an object or an event can pass through (received messages, errors, conditions) **state chart diagrams** are built. They represent the **lifecycle of objects**, subsystems and systems. Object states change when receiving **events** or **signals**.
4. **Activity diagrams** is developed to show **actions** and **results** of those actions and to emphasize the **workflows**.
5. **Interaction diagrams** are developed to show interactions between objects: sequence diagram and communication diagrams.
 - **Sequence diagrams** describe the way objects interact and communicate with each other, by focusing on messages that are sent and received
 - **Communication diagrams** allow the representation of both the interactions and connections between a set of collaborating objects. It is recommended when the visualization of space coordinate is useful.

Class diagram

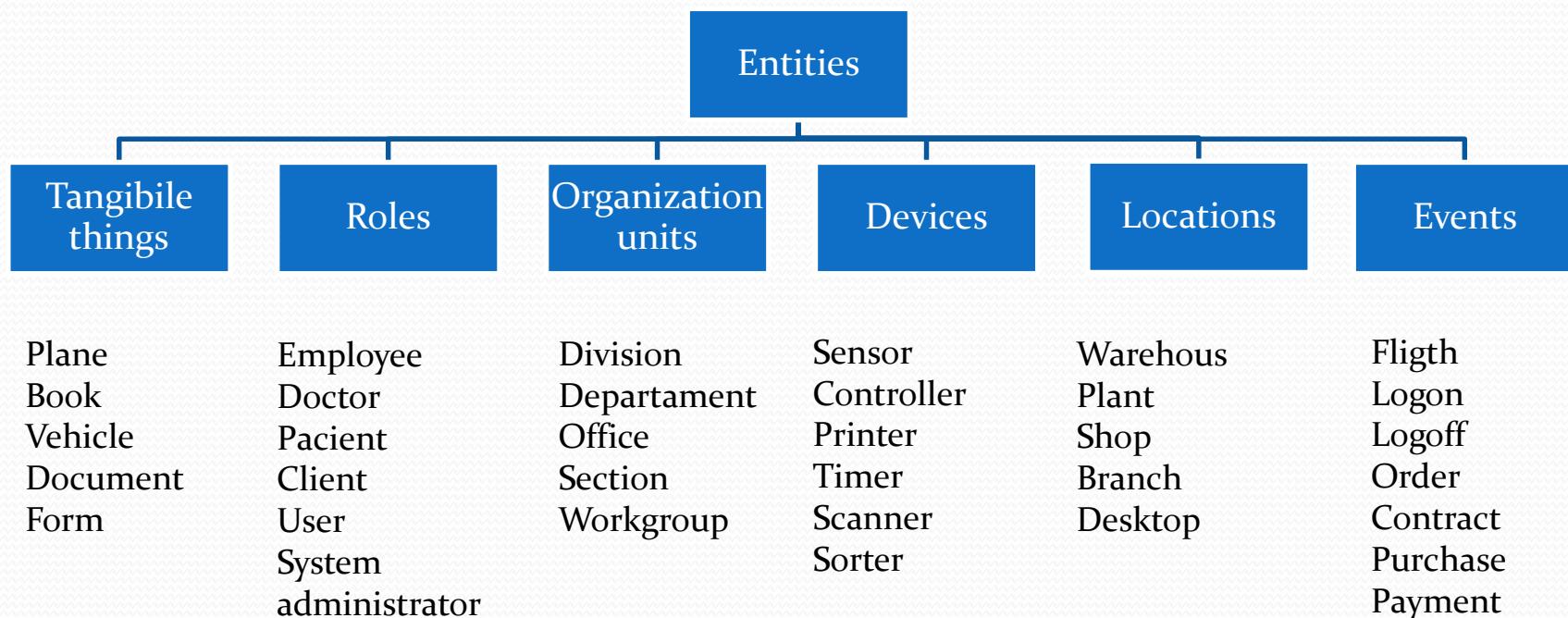
- The class diagram is **the most important diagram** of object oriented analysis and design. The purpose of class diagram is to present the static nature of classes, showing their attributes, operations and associations.
- Most modeling tools **generate source code** starting **from the class diagram alone**.
- The other UML diagrams provide different **points of view** for identifying attributes, operations and relationships between classes. They help at class diagram validation and may serve to clarify specific issues.

Class identification techniques

- a) **Brainstorming technique** – a list of control is used to include all **entity types** that occur frequently. Brainstorming sessions are performed to identify **domain classes** for each of those types.
- b) **Noun technique** – all the **nouns** that occur in the system description are identified, and it is decided whether the noun is a class or the field, an attribute or an unimportant element

a. Brainstorming technique

- We have some elements like ...

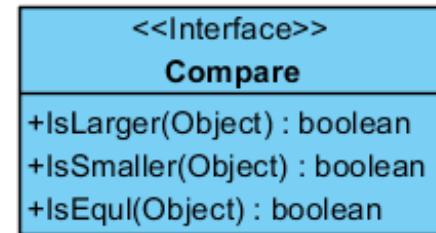
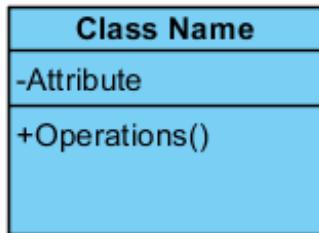


b. Noun technique

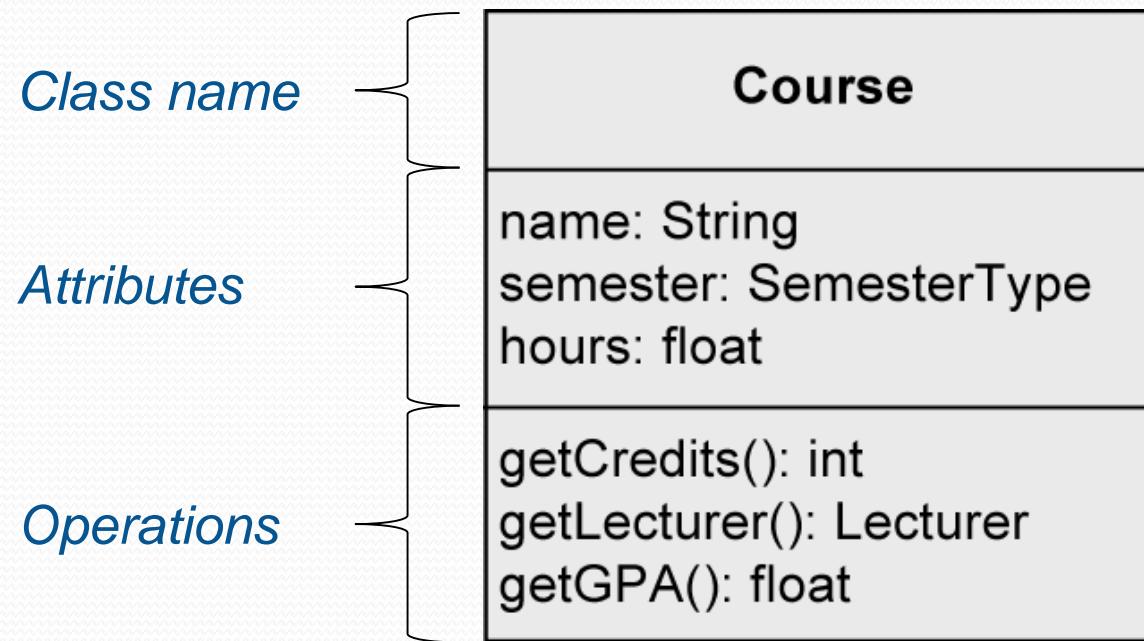
1. All the nouns are identified starting from use cases, actors and other information about the computer system.
2. The necessary **information is collected** from existing systems, procedures, forms and reports.
3. The noun list is then refined
 - Does it belong to the analyzed system scope?
 - Is it necessary to keep more than one element of this type?
 - Is it a synonym of an already identified noun?
 - Is it an attribute of a registered noun?
 - Is it just a system output obtained from an already identified noun?
 - Is it just an input resulting from the recording of an already identified noun?
4. It is created the **main list of nouns** and notes for each noun to be included / excluded / discussed.
5. The list of users, beneficiaries, team members is presented for **validation**

Defining a class

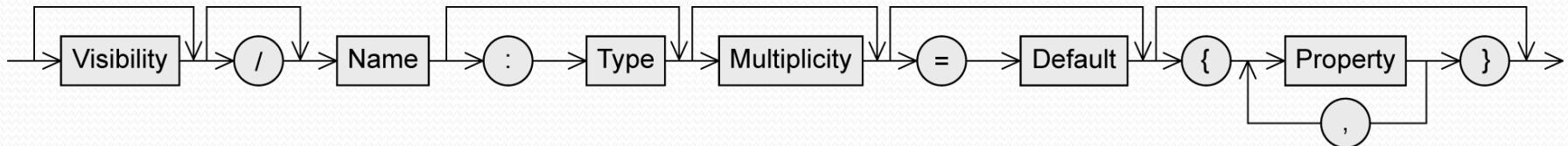
- Set of objects that have the same characteristics and constraints.
- The characteristics of a class are **attributes** and **operations**.
- **Abstract classes** can not be instantiated. Their role is to enable other classes to inherit them, for reuse of characteristics.
- **An interface** describes a set of characteristics and public obligations. Actually, it specifies a contract. Any instance that implements the interface must provide the services provided by the contract.



Defining a class



Attribute Syntax – Visibility, derived attribute



Person

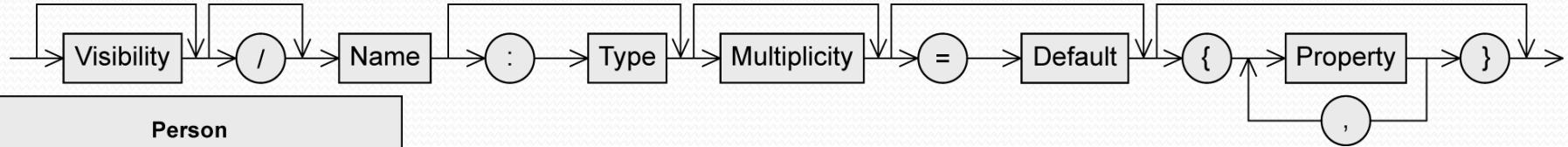
```
+ firstName: String  
+ lastName: String  
- dob: Date  
# address: String[1..*] {unique, ordered}  
- ssNo: String {readOnly}  
- /age: int  
- password: String = "pw123"  
- personsNumber: int
```

Person

```
firstName: String  
lastName: String  
dob: Date  
address: String[1..*] {unique, ordered}  
ssNo: String {readOnly}  
/age: int  
password: String = "pw123"  
personsNumber: int
```

- Who is permitted to access the attribute
 - + ... public: everybody
 - - ... private: only the object itself
 - # ... protected: class itself and subclasses
 - ~ ... package: classes that are in the same package
- / Attribute value is derived from other attributes
 - **age**: calculated from the date of birth

Attribute Syntax – Name, type



Person

```
firstName: String  
lastName: String  
dob: Date  
address: String[1..*] {unique, ordered}  
ssNo: String {readOnly}  
/age: int  
password: String = "pw123"  
personsNumber: int
```

Person

```
firstName: String  
lastName: String  
dob: Date  
address: String[1..*] {unique, ordered}  
ssNo: String {readOnly}  
/age: int  
password: String = "pw123"  
personsNumber: int
```

- Name of the attribute

- Type

- User-defined classes
- Data type
 - Primitive data type
 - Pre-defined: Boolean, Integer, UnlimitedNatural, String
 - User-defined: «**primitive**»
 - Composite data type: «**datatype**»
 - Enumerations: «**enumeration**»

«**primitive**»
Float

round(): void

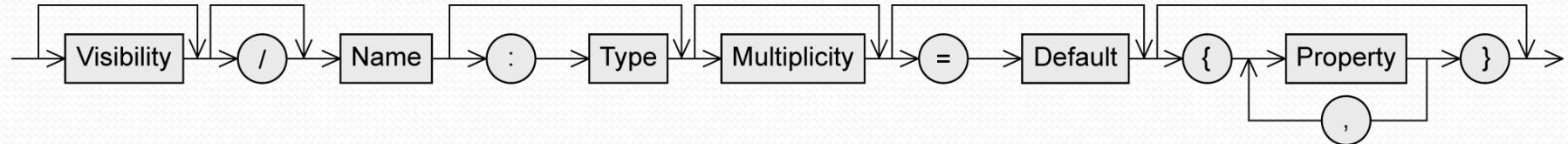
«**datatype**»
Date

day
month
year

«**enumeration**»
AcademicDegree

bachelor
master
phd

Attribute Syntax – Multiplicity, default value



Person

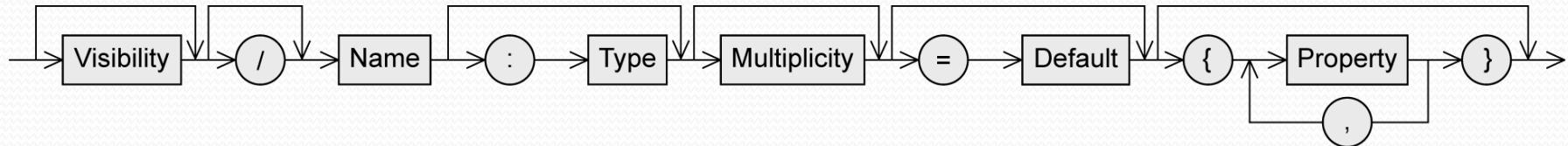
```
firstName: String  
lastName: String  
dob: Date  
address: String[1..*] {unique, ordered}  
ssNo: String {readOnly}  
/age: int  
password: String = "pw123"  
personsNumber: int
```

Person

```
firstName: String  
lastName: String  
dob: Date  
address: String[1..*] {unique, ordered}  
ssNo: String {readOnly}  
/age: int  
password: String = "pw123"  
personsNumber: int
```

- Number of values an attribute may contain
- Default value: 1
- Notation: **[min..max]**
 - no upper limit: **[*]** or **[0 .. ***
- Default value
 - Used if the attribute value is not set explicitly by the user

Attribute Syntax – Properties



Person

```
firstName: String  
lastName: String  
dob: Date  
address: String[1..*] {unique, ordered}  
ssNo: String {readOnly}  
age: int  
password: String = "pw123"  
personsNumber: int
```

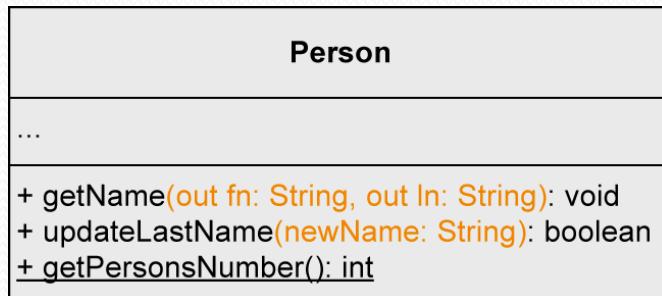
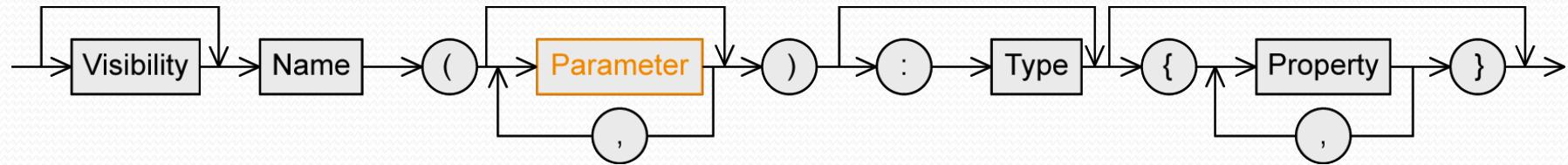
Pre-defined properties

- {readOnly} ... value cannot be changed
- {unique} ... no duplicates permitted
- {non-unique} ... duplicates permitted
- {ordered} ... fixed order of the values
- {unordered} ... no fixed order of the values

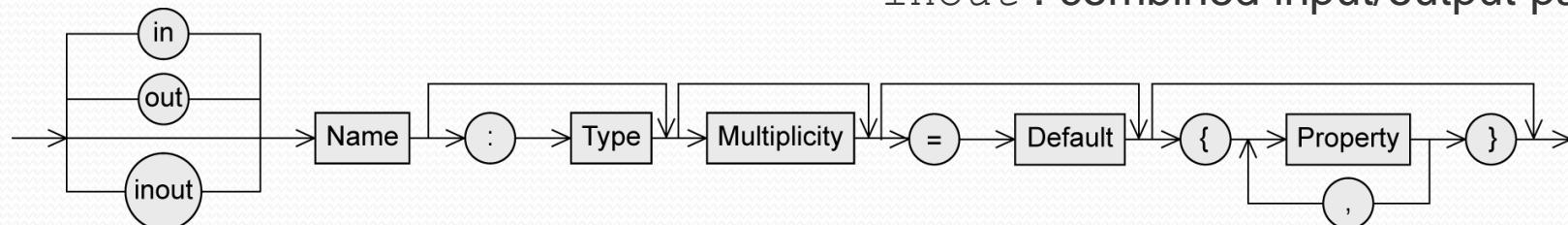
Attribute specification

- Set: {unordered, unique}
- Multi-set: {unordered, non-unique}
- Ordered set: {ordered, unique}
- List: {ordered, non-unique}

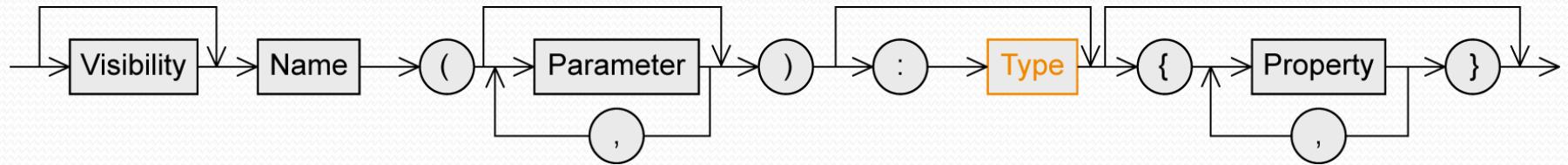
Operation Syntax - Parameters



- Notation similar to attributes
- Direction of the parameter
 - **in** ... input parameter
 - When the operation is used, a value is expected from this parameter
 - **out** ... output parameter
 - After the execution of the operation, the parameter has adopted a new value
 - **inout** : combined input/output parameter



Operation Syntax - Type



Person

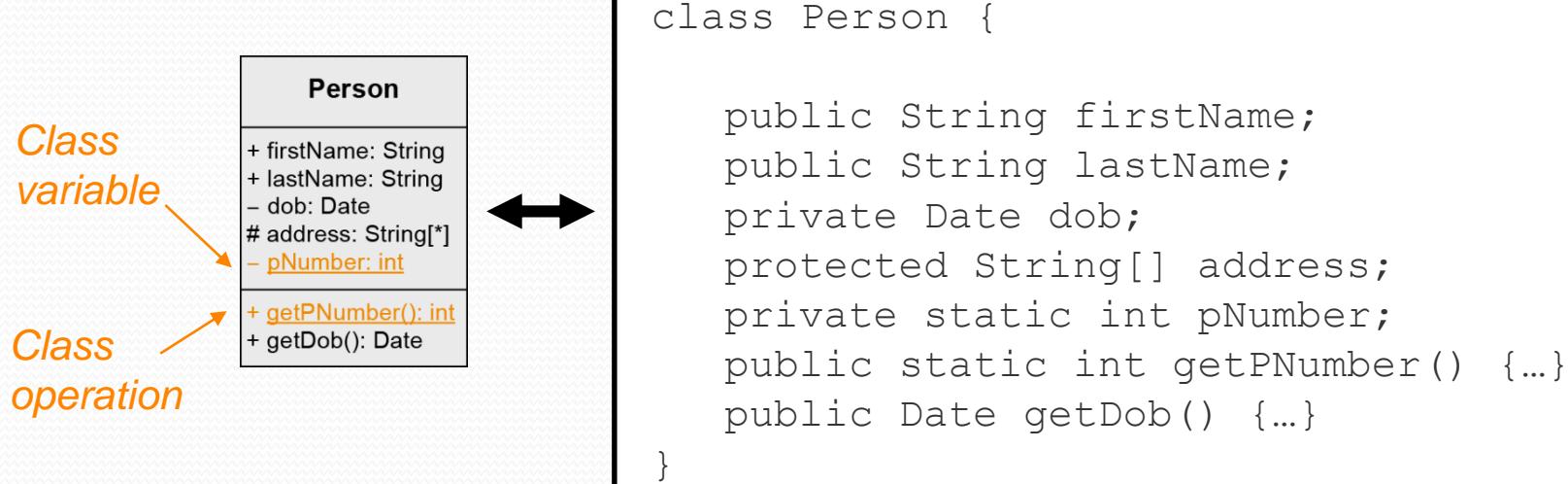
...

```
getName(out fn: String, out ln: String): void  
updateLastName(newName: String): boolean  
getPersonsNumber(): int
```

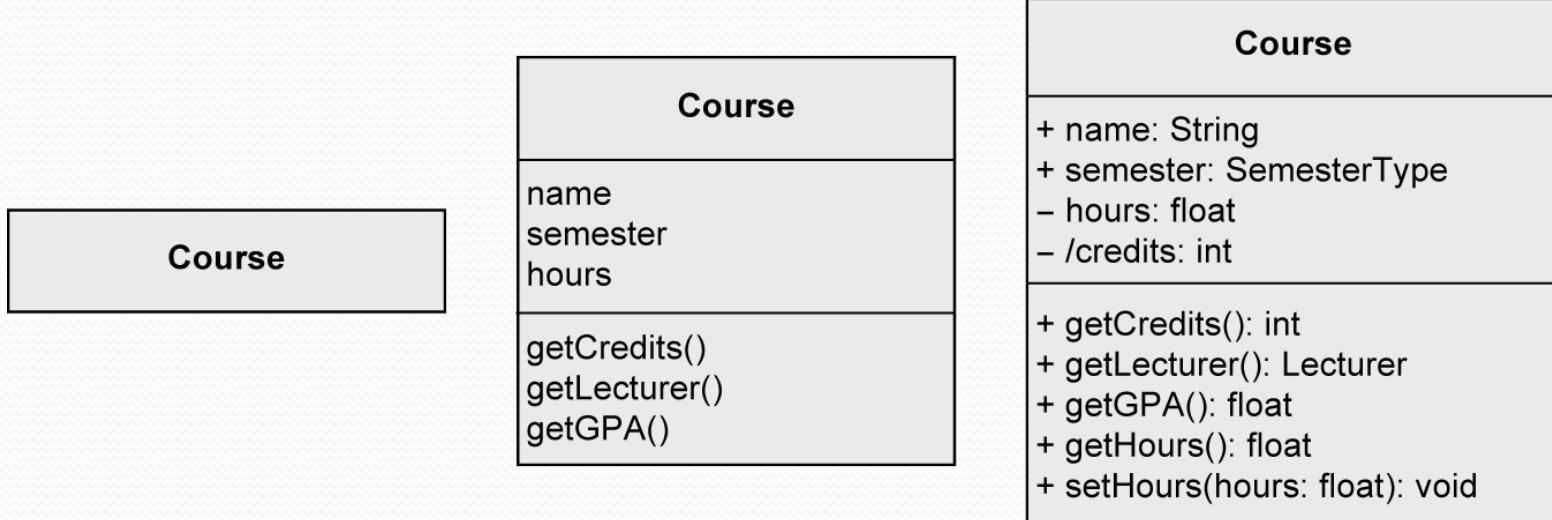
- Type of the return value

Class Variable and Class Operation

- Instance variable (= instance attribute): attributes defined on instance level
- Class variable (= class attribute, static attribute)
 - Defined only once per class, i.e., shared by all instances of the class
 - E.g. counters for the number of instances of a class, constants, etc.
- Class operation (= static operation)
 - Can be used if no instance of the corresponding class was created
 - E.g. constructors, counting operations, math. functions ($\sin(x)$), etc.
- Notation: underlining name of class variable / class operation



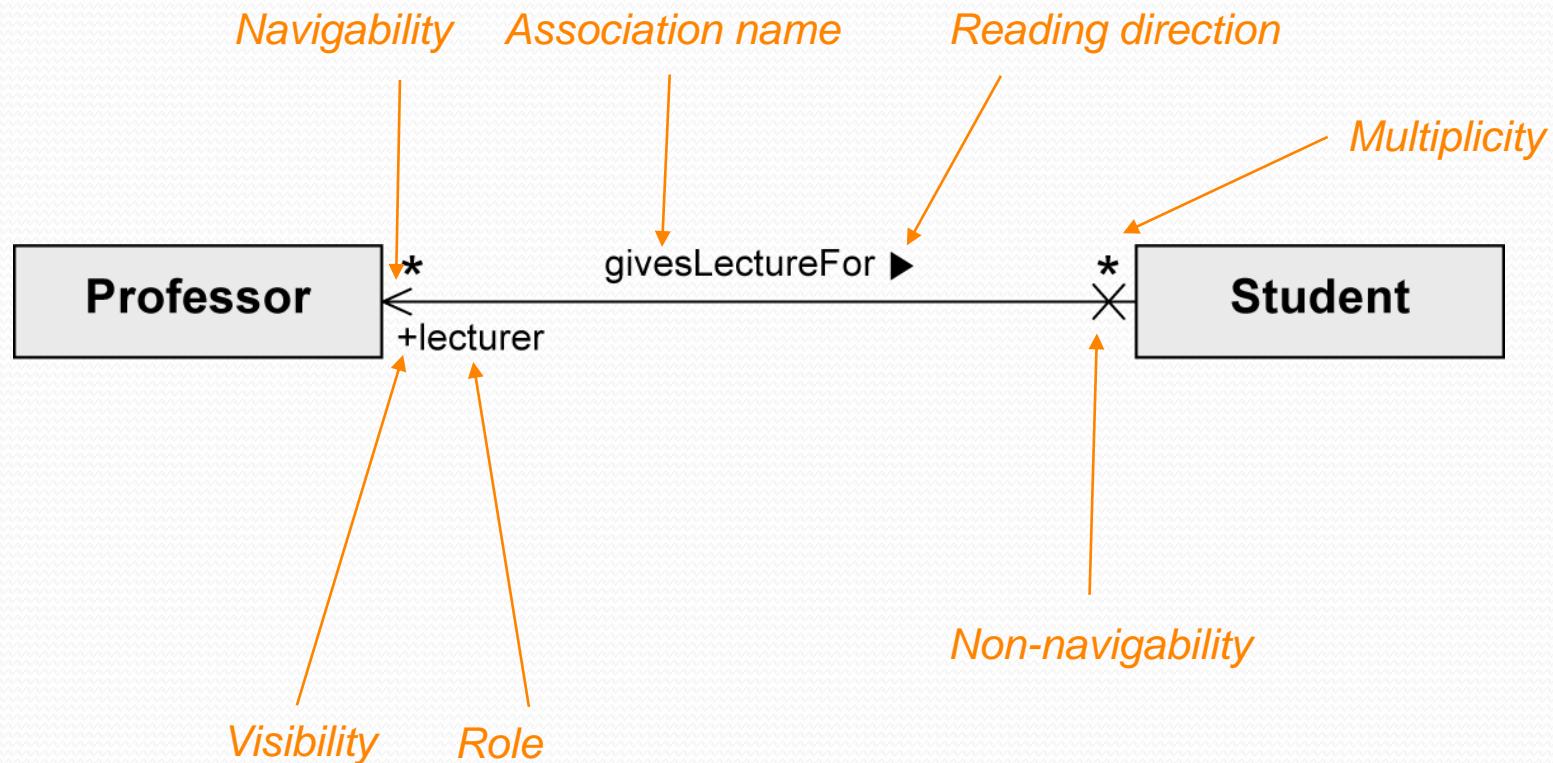
Specification of Classes: Different Levels of Detail



Association

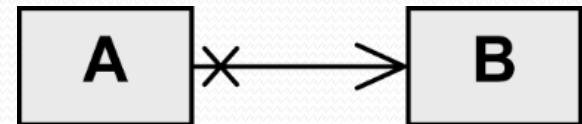


- Connects instances of two classes with one another



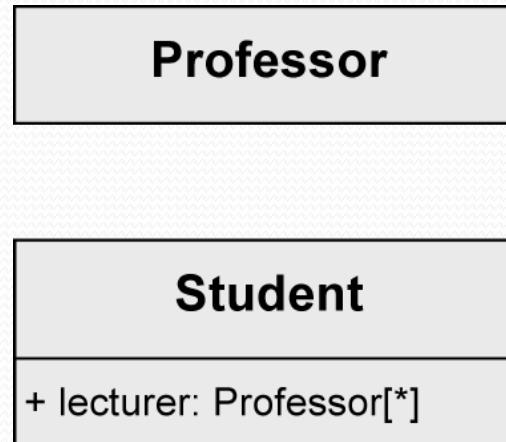
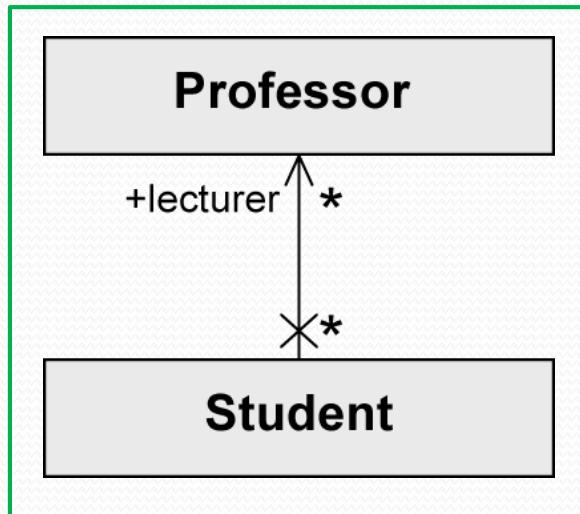
Binary Association - Navigability

- Navigability: an object knows its partner objects and can therefore access their visible attributes and operations
 - Indicated by open arrow head
- Non-navigability
 - Indicated by cross
- Example:
 - **A** can access the visible attributes and operations of **B**
 - **B** cannot access any attributes and operations of **A**
- Navigability undefined
 - Bidirectional navigability is assumed



Binary Association as Attribute

Preferable



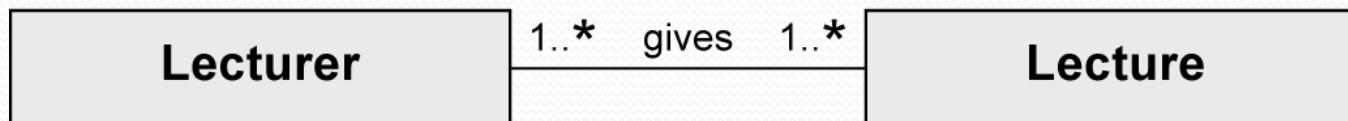
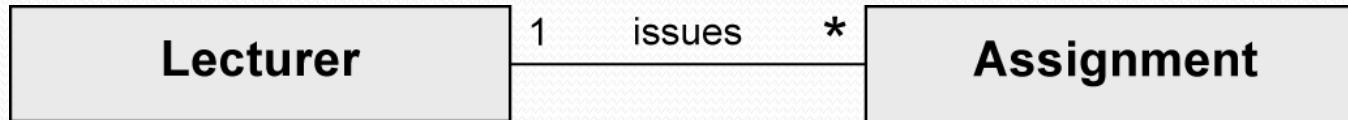
- Java-like notation:

```
class Professor { ... }

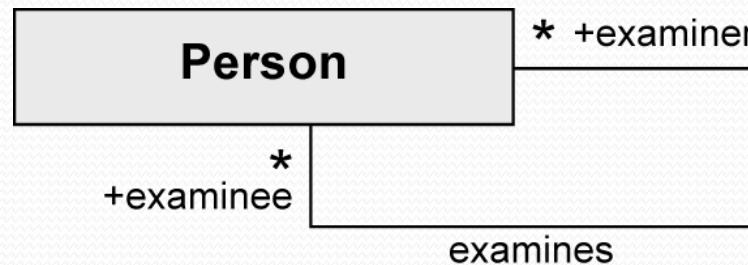
class Student{
    public Professor[] lecturer;
    ...
}
```

Binary Association – Multiplicity and Role

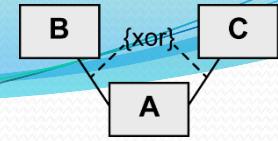
- Multiplicity: Number of objects that may be associated with exactly one object of the opposite side



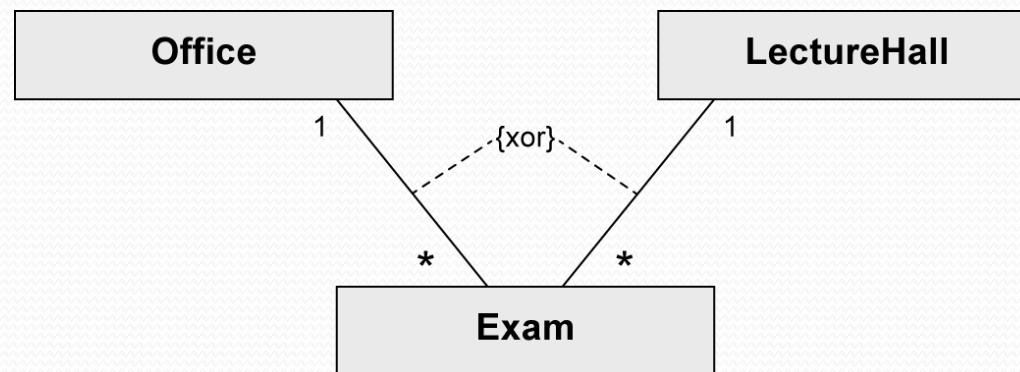
- Role: describes the way in which an object is involved in an association relationship



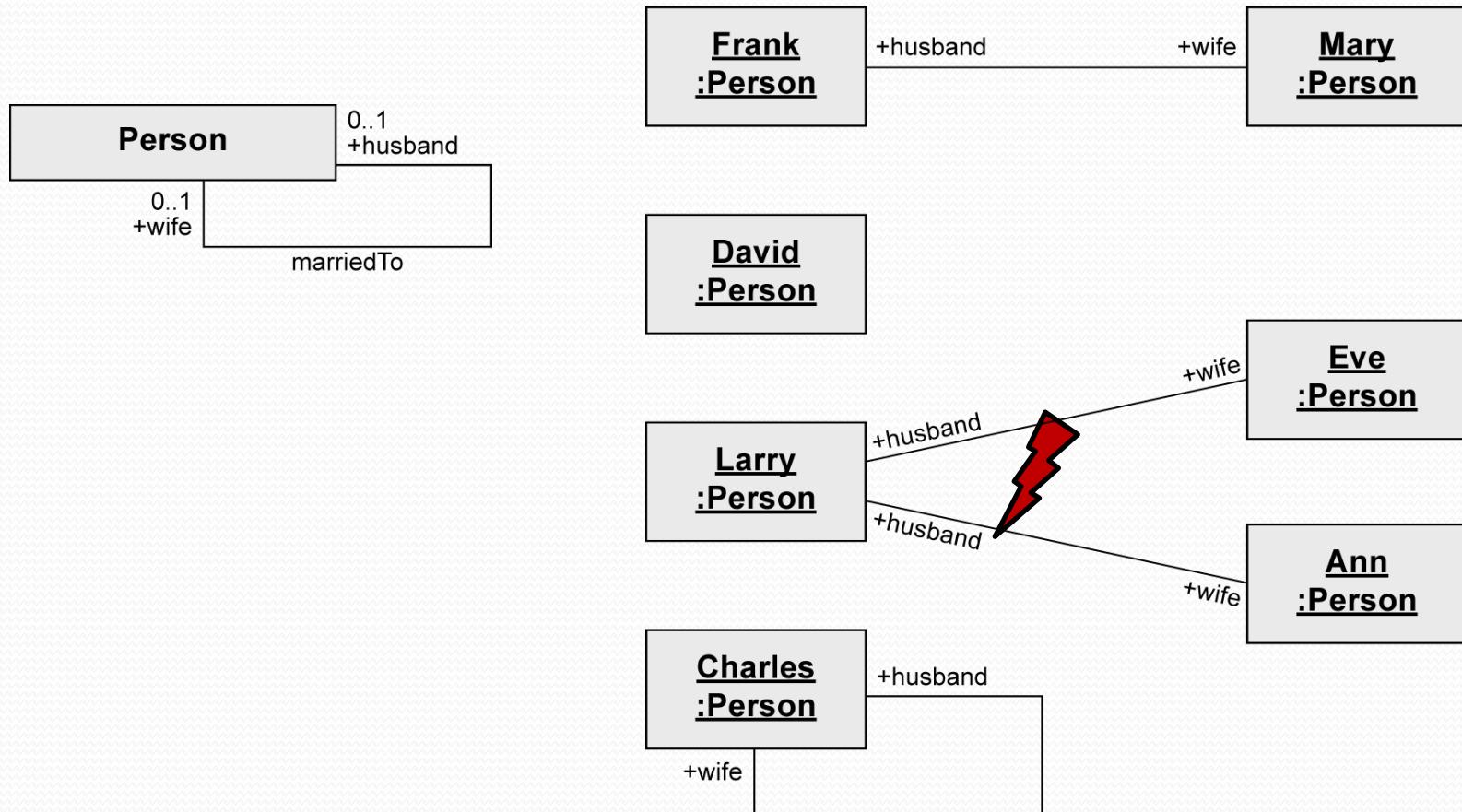
Binary Association – xor constraint



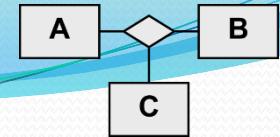
- “exclusive or” constraint
- An object of class **A** is to be associated with an object of class **B** or an object of class **C** but not with both.



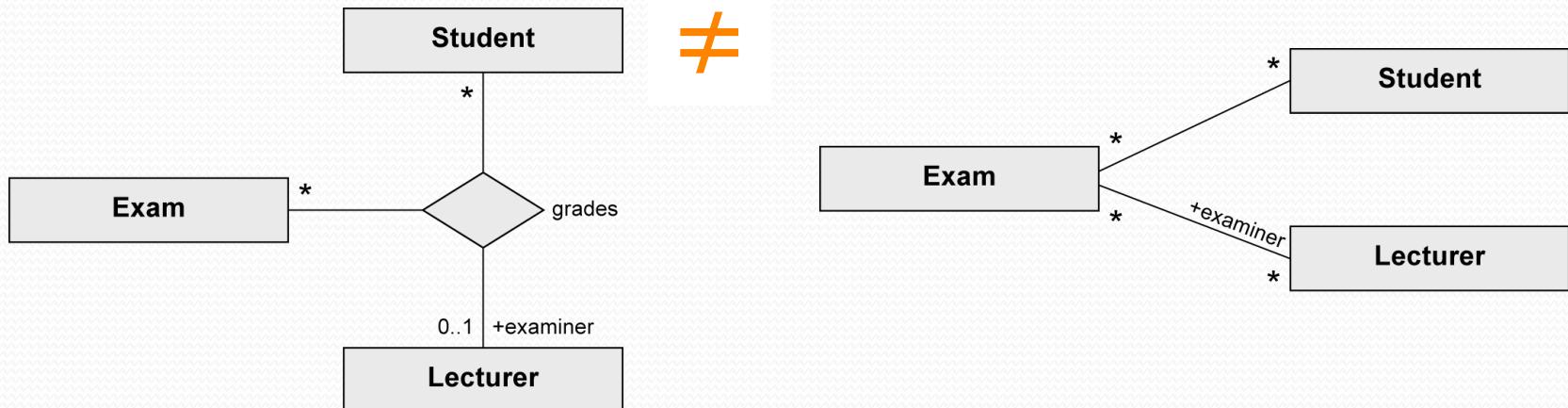
Unary Association - Example

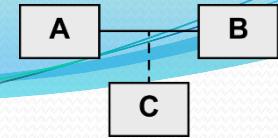


n-ary Association



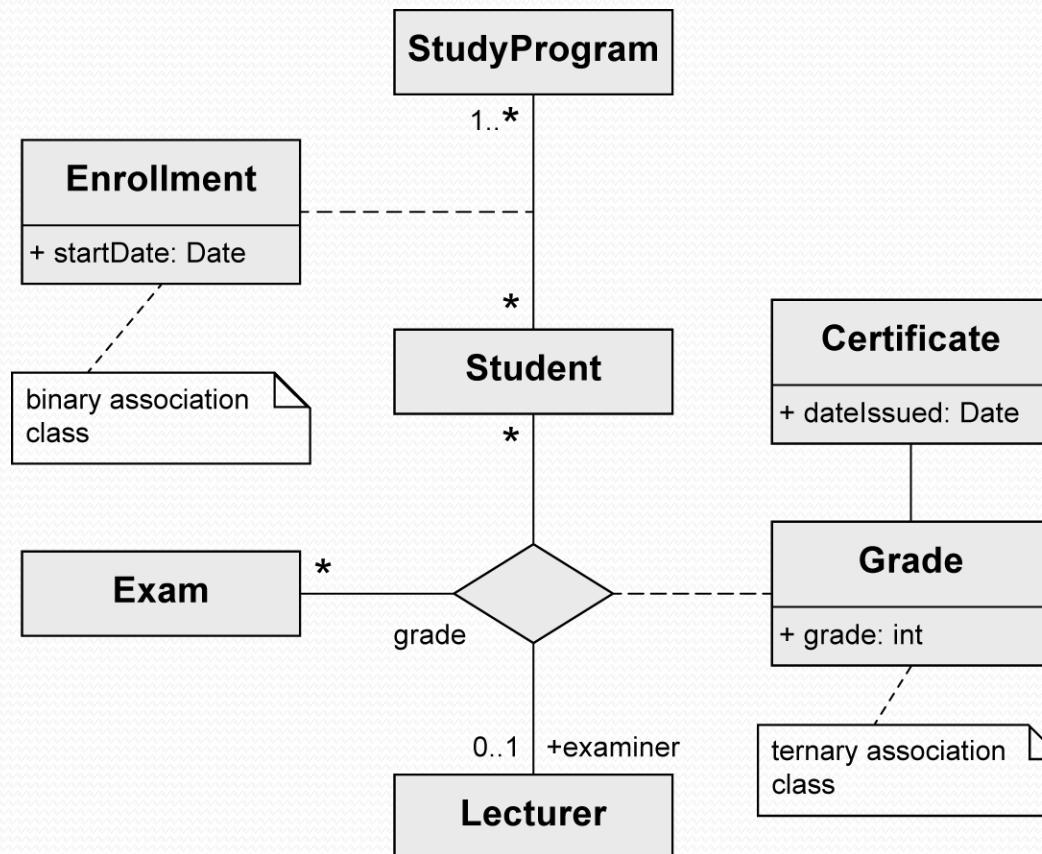
- More than two partner objects are involved in the relationship.
- No navigation directions
- Example
 - (Student, Exam) → (Lecturer)
 - One student takes one exam with one or no lecturer
 - (Exam, Lecturer) → (Student)
 - One exam with one lecturer can be taken by any number of students
 - (Student, Lecturer) → (Exam)
 - One student can be graded by one Lecturer for any number of exams





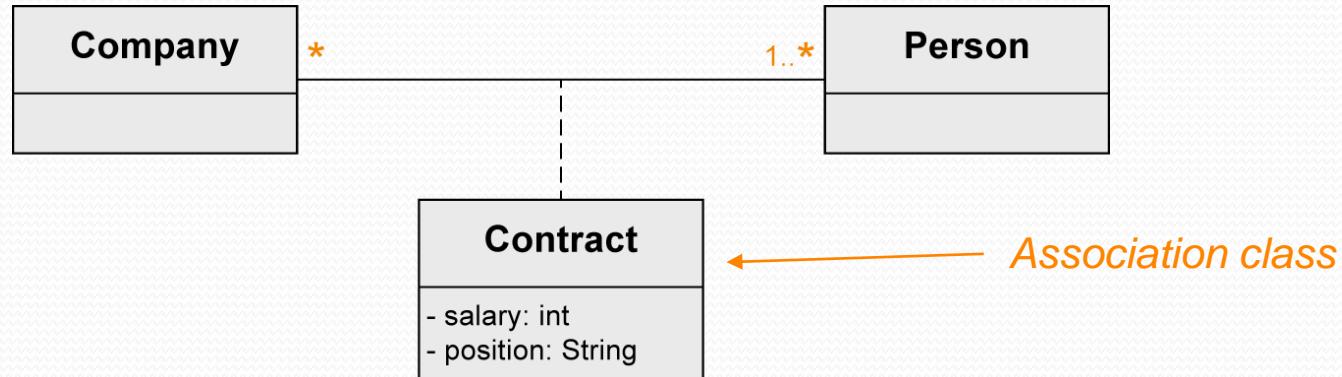
Association Class

- Assign attributes to the relationship between classes rather than to a class itself

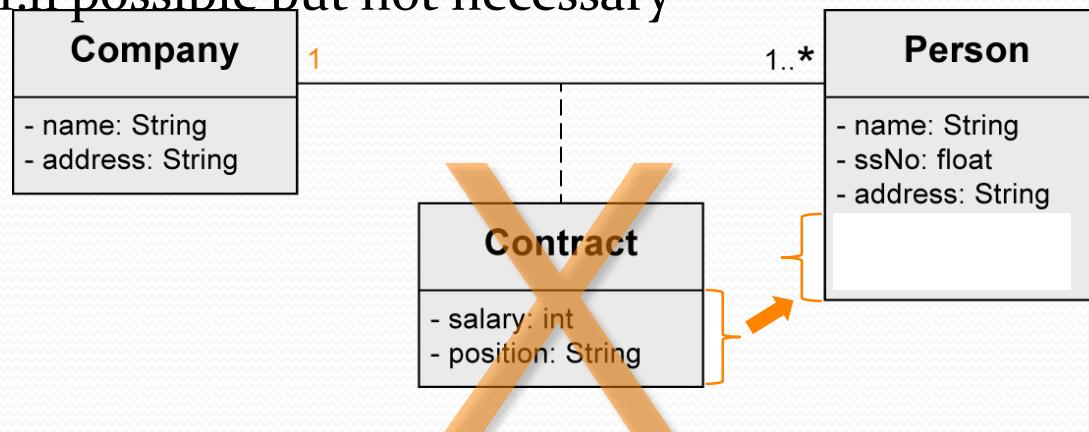


Association Class

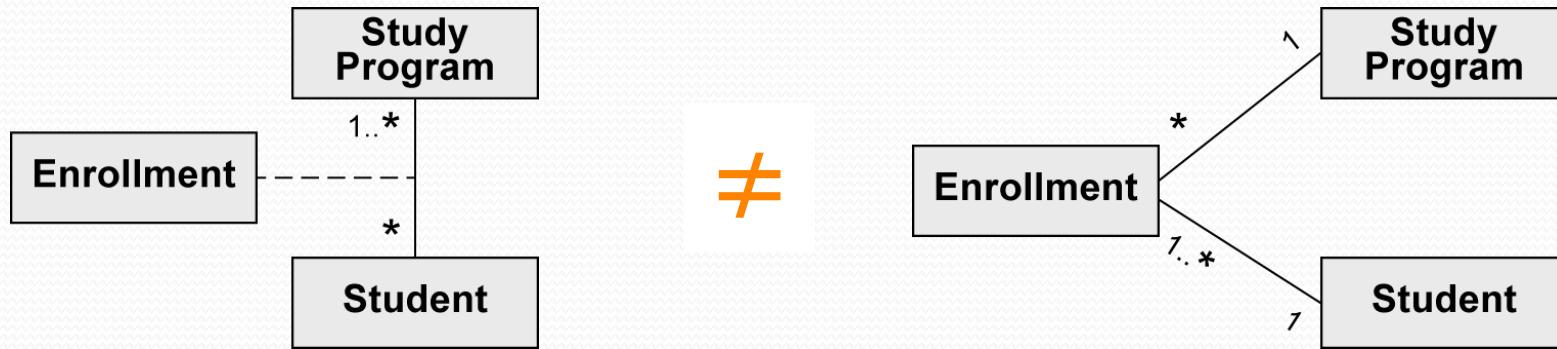
- Necessary when modeling n:m Associations



- With 1:1 or 1:n possible but not necessary



Association Class vs. Regular Class



A *Student* can enroll for one particular *StudyProgram* only once

A *Student* can have **multiple** *Enrollments* for one and the same *StudyProgram*

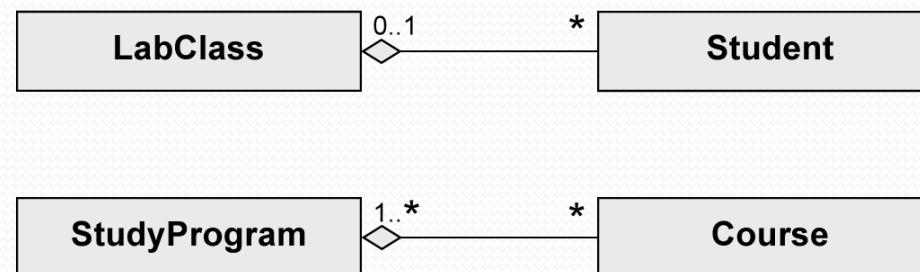
Aggregation

- Special form of association
- Used to express that a class is part of another class
- Properties of the aggregation association:
 - **Transitive**: if **B** is part of **A** and **C** is part of **B**, **C** is also part of **A**
 - **Asymmetric**: it is not possible for **A** to be part of **B** and **B** to be part of **A** simultaneously.
- Two types:
 - Shared aggregation
 - Composition



Shared Aggregation

- Expresses a weak belonging of the parts to a whole
 - = Parts also exist independently of the whole
- Multiplicity at the aggregating end may be >1
 - = One element can be part of multiple other elements simultaneously
- Spans a directed acyclic graph
- Syntax: Hollow diamond at the aggregating end
- Example:
 - Student** is part of **LabClass**
 - Course** is part of **StudyProgram**



Composition

- Existence dependency between the composite object and its parts
- One part can only be contained in at most one composite object at one specific point in time
 - Multiplicity at the aggregating end max. 1
-> The composite objects form a tree
- If the composite object is deleted, its parts are also deleted.
- Syntax: Solid diamond at the aggregating end
- Example: **Beamer** is part of **LectureHall** is part of **Building**

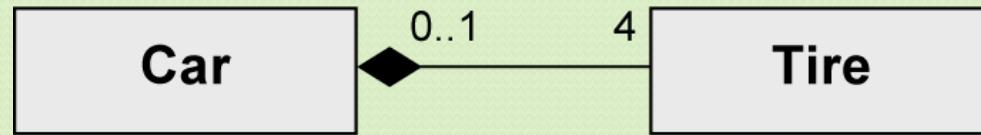


*If the Building is deleted,
the LectureHall is also deleted*

*The Beamer can exist without the
LectureHall, but if it is contained in the
LectureHall while it is deleted, the Beamer
is also deleted*

Shared Aggregation and Composition

- Which model applies?



A Tire can exist without a Car. A Tire belongs to one Car at most.

Yes

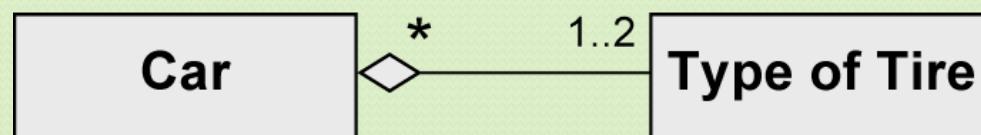


A Tire cannot exist without a Car.

No



A Tire can belong to multiple Cars



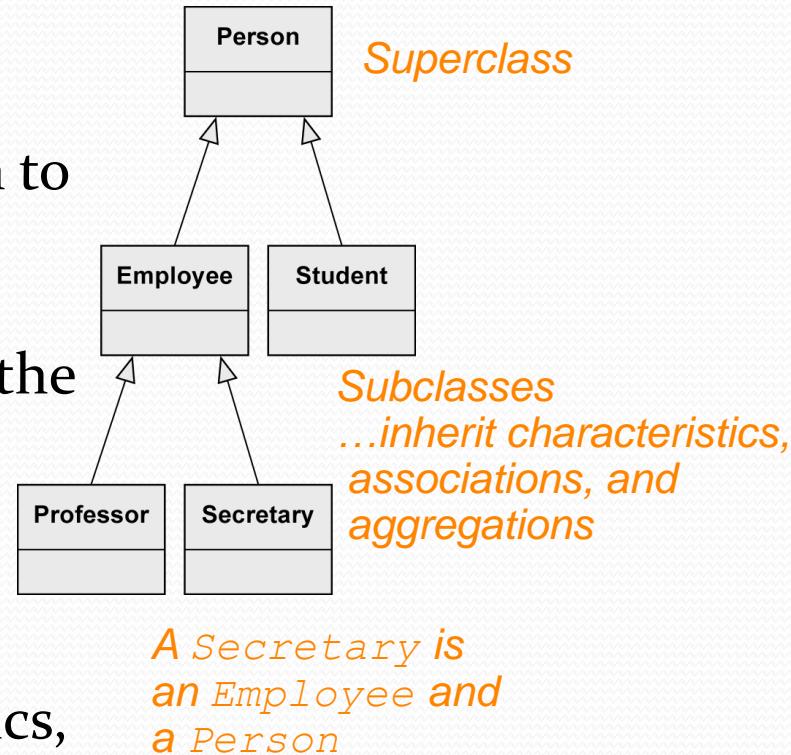
A Car has one or two types of Tires. Several Cars may have the same Type of Tires.

Yes

Generalization



- Characteristics (attributes and operations), associations, and aggregations that are specified for a general class (superclass) are passed on to its subclasses.
- Every instance of a subclass is simultaneously an indirect instance of the superclass.
- Subclass inherits all characteristics, associations, and aggregations of the superclass except private ones.
- Subclass may have further characteristics, associations, and aggregations.
- Generalizations are transitive.



{abstract}

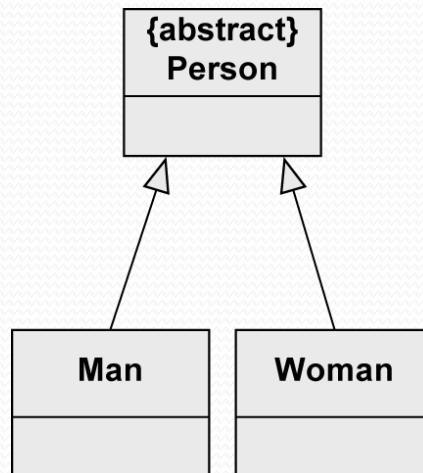
A

Generalization – Abstract Class

- Used to highlight common characteristics of their subclasses.
- Used to ensure that there are no direct instances of the superclass.
- Only its non-abstract subclasses can be instantiated.
- Useful in the context of generalization relationships.
- Notation: keyword **{ abstract }** or class name in italic font.

{abstract}
Person

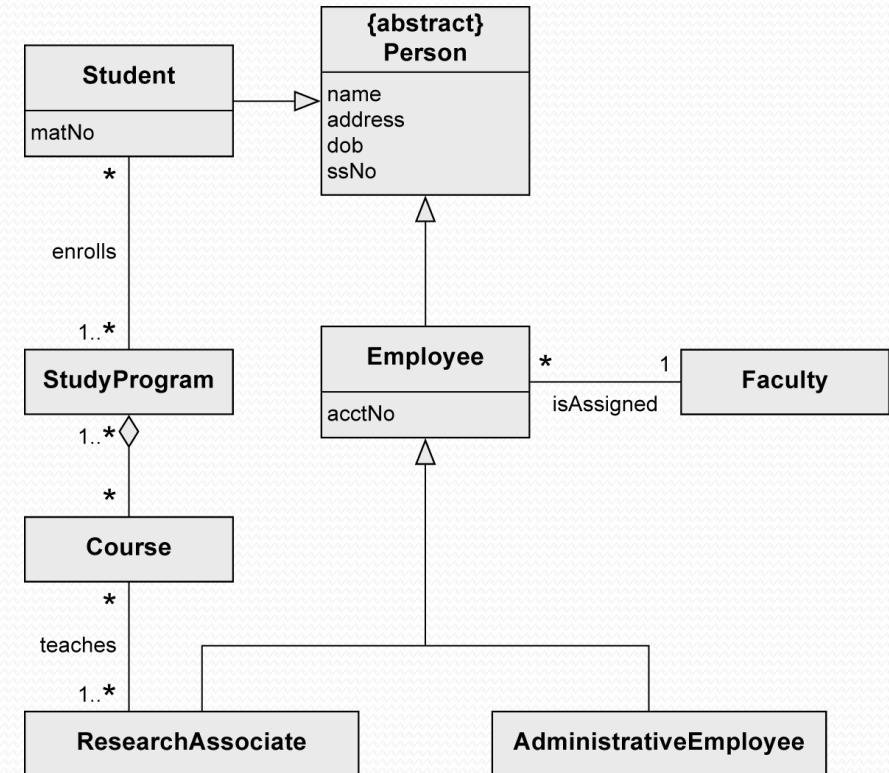
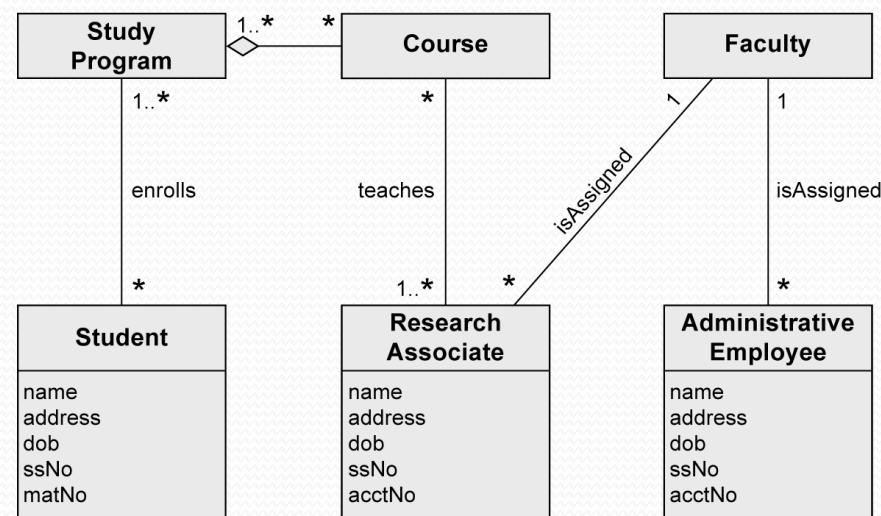
Person



No Person-object possible

Two types of Person: Man and Woman

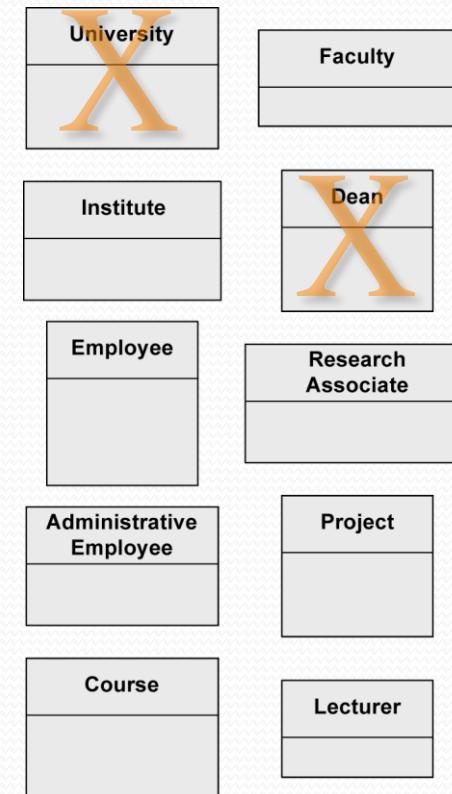
With and Without Generalization



Example – Step 1: Identifying Classes

- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known. Employees have a social security number, a name, and an email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known. Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.

We model the system „University“

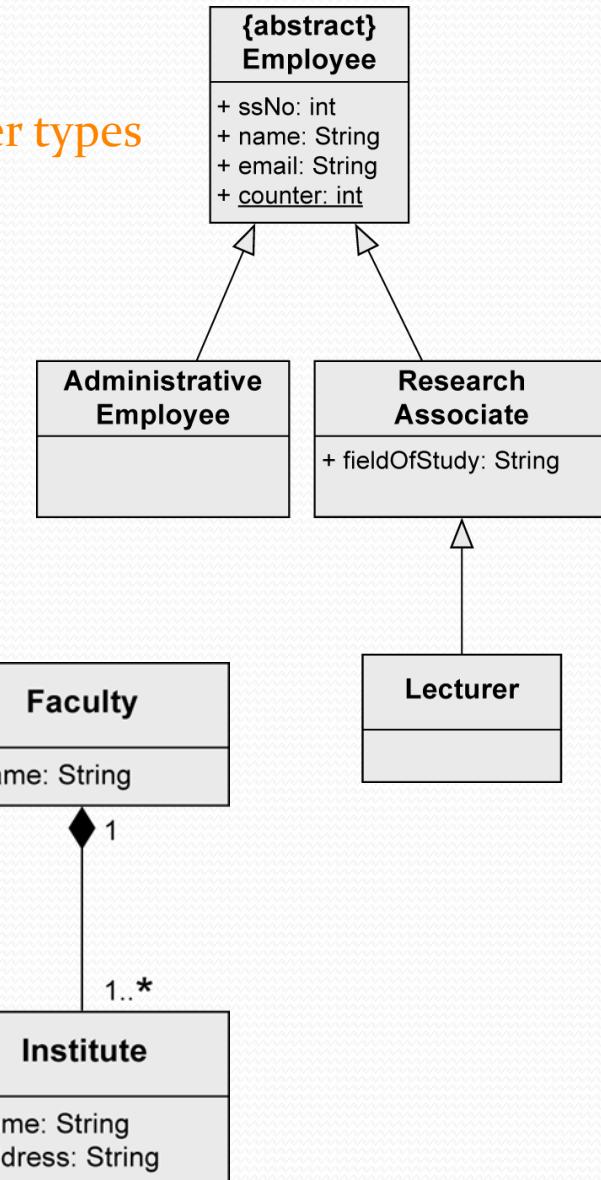


Dean has no further attributes than any other employee

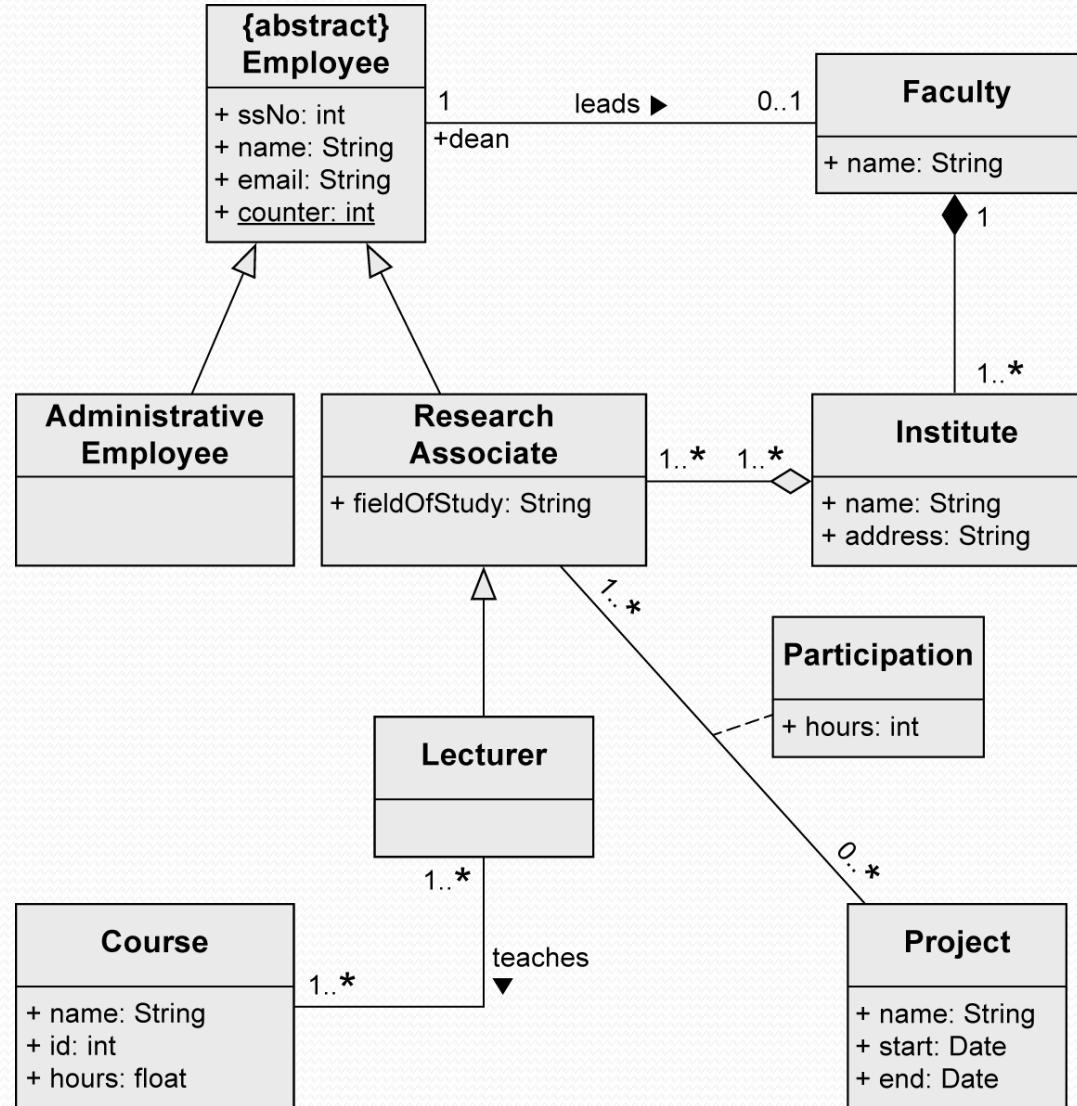
Example – Step 2: Identifying Relationships

- Three kinds of relationships:
 - Association
 - Generalization
 - Aggregation
- Indication of a generalization
- “There is a distinction between research and administrative personnel.”*
- “Some research associates hold courses. Then they are called lecturers.”*
- “A university consists of multiple faculties which are composed of various institutes.”*

Abstract, i.e., no other types of employees



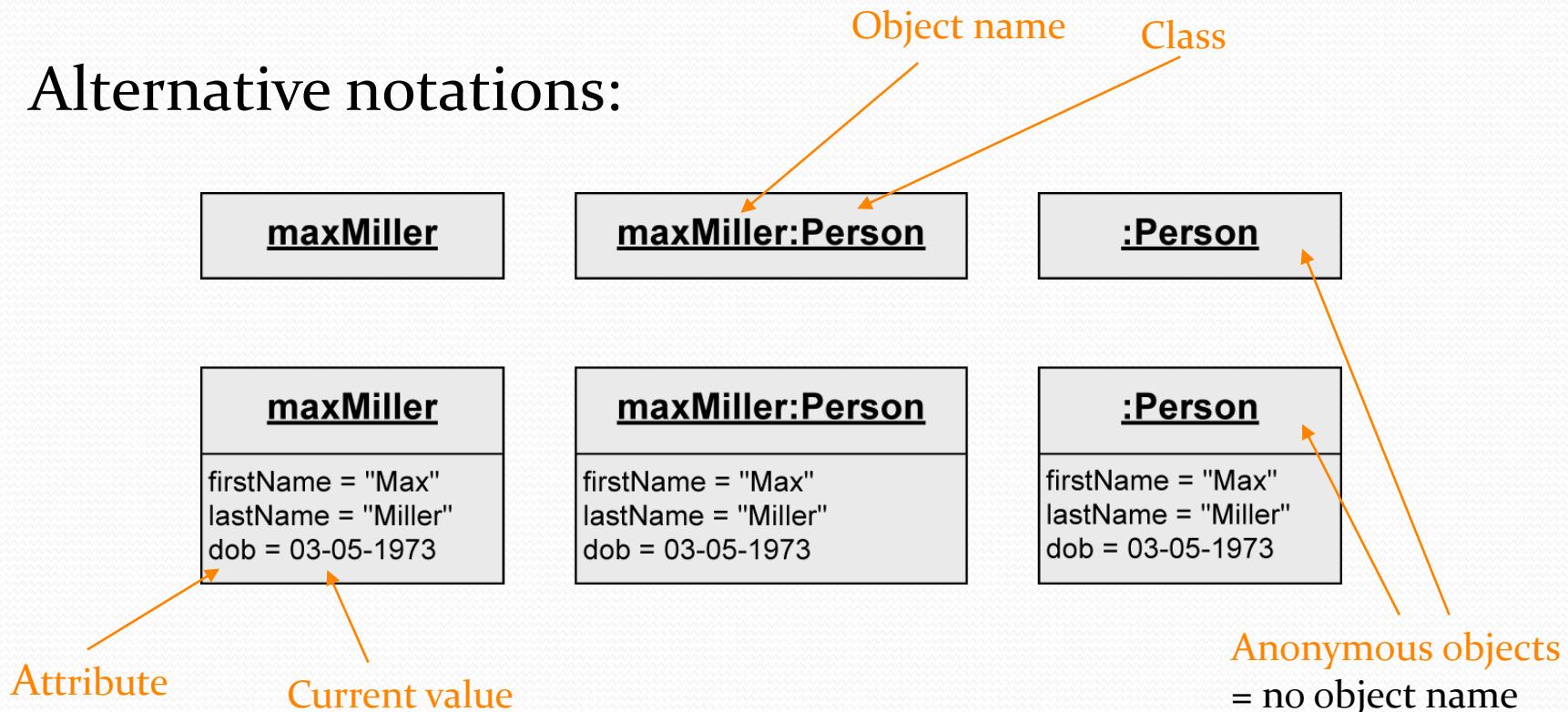
Example – Complete Class Diagram



Object

- Individuals of a system

- Alternative notations:

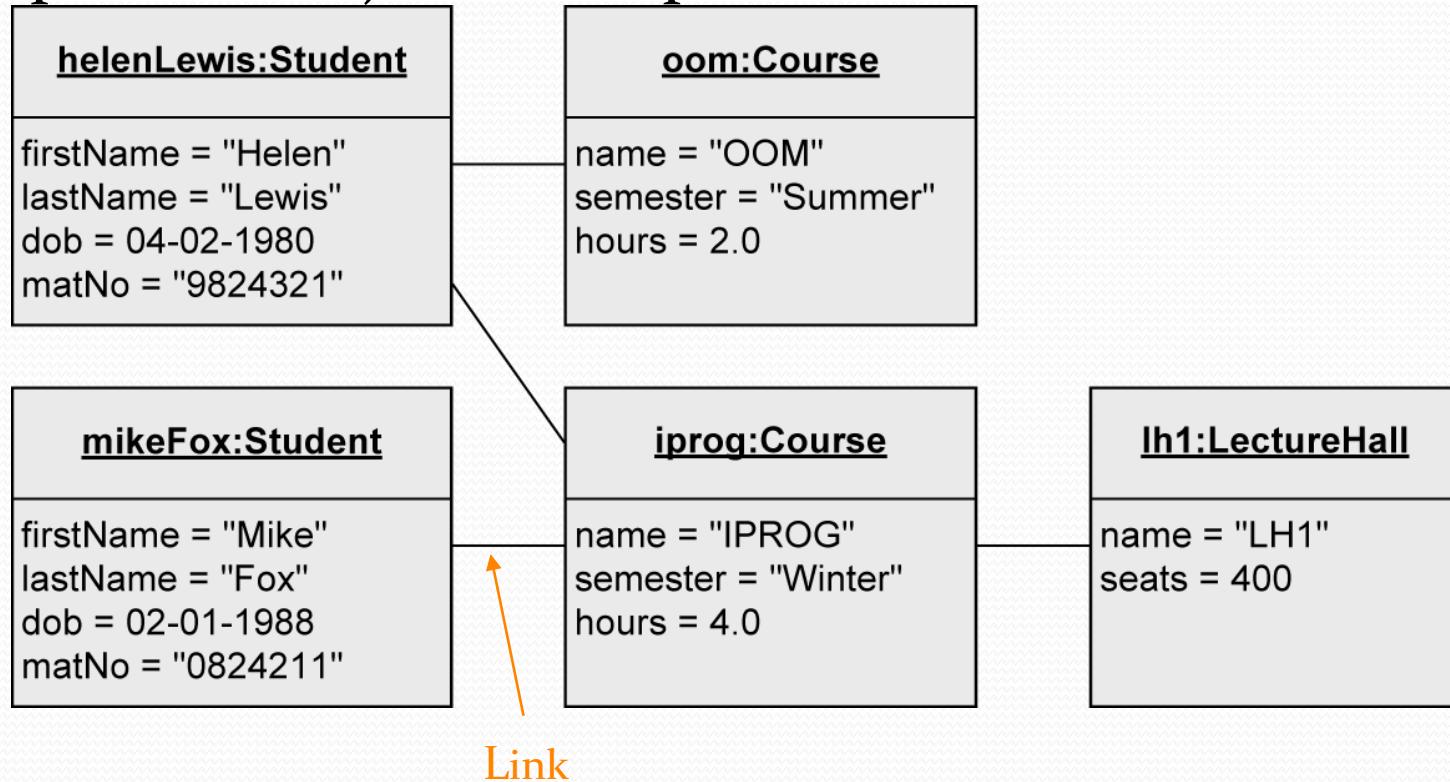


Object Diagram

o1

o2

- Objects of a system and their relationships (links)
- Snapshot of objects at a specific moment in time

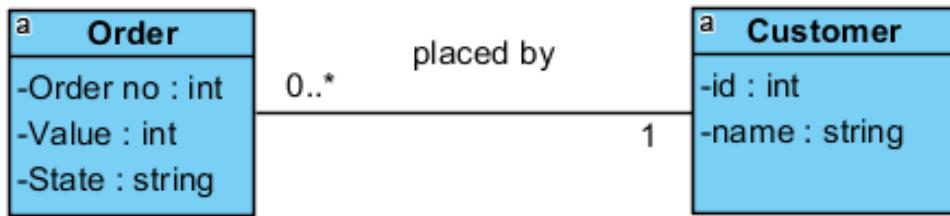


Class and Object diagram notation – a comparison

Class diagram	Object diagram
The class has three compartments: name, attributes and operations.	An object has only two compartments : names and attributes.
The class name is specified alone in the first compartment.	The format of an object's name includes class names . These notations are found in other diagrams that represent objects
The second section describes the properties as attributes.	The second section defines values for each attribute, in order to test the model.
Operations show up in the description of the class.	Operations are not included in objects because they are identical for each object of the class.
Classes are connected by associations that have a name, multiplicity, constraints and roles. Classes are an abstraction of objects, so it is necessary to specify how many classes are participating in an association.	The objects are connected by a relationship that can have a name, roles, but not multiplicities . Objects are singular entities, all links are one-on-one, and multiplicities are irrelevant.

Object diagram in Visual Paradigm

- Define the **class diagram**, in which the classes have attributes



- Define an **object** in the **Object diagram** (Instance Specification).
- Select the class to which the object belongs: Right click on the object -> **Select Classifier**-> tick and select the appropriate class
- Optionally, the object is given a **name**.
- Define values for attributes: Right click on the object -> **Slots**, Slots Define (for attributes we want to give values to) -> **Edit Values**-> Add -> Text (insert desired value).
- We create **links** (Link) between objects.

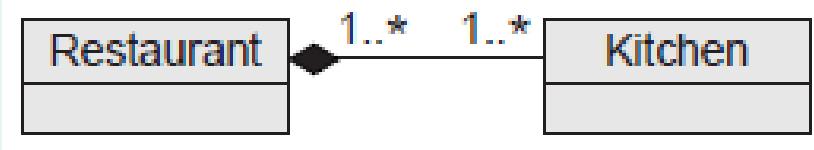
Every restaurant has at least one kitchen, one kitchen is part of exactly one restaurant.

Choose one or more:

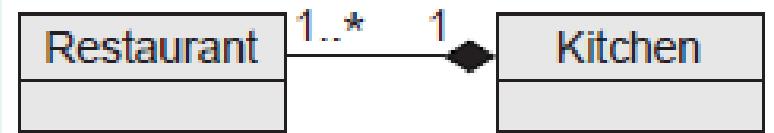
i.



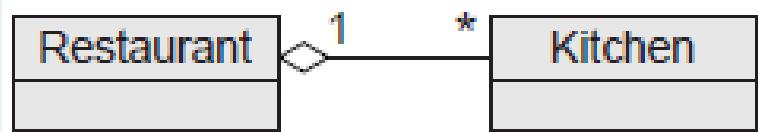
ii.



iii.



iv.



a. i

b. iii

c. iv

d. i + iv

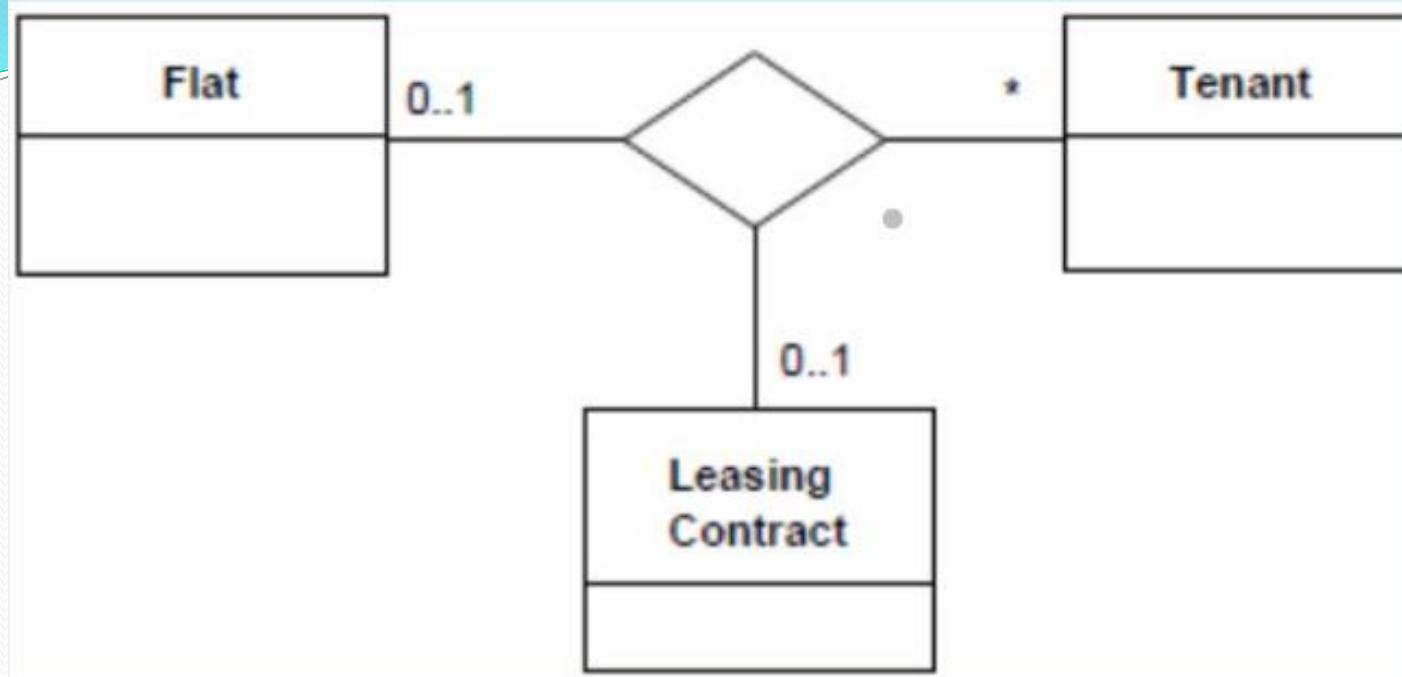
Enumerations....

Select one or more:

Selectați unul sau mai multe:

- a. ... are a user-definable data type which instances form a list of named literal values.
- b. ... have the keyword <enumeration>.
- c. ... may have operations.

You are given the following clipping of a UML2 class diagram. Which of the following statements are true? Select one or more:



- a. One flat can be rented by multiple tenants with the same leasing contract.
- b. One tenant can rent multiple flats with the same leasing contract.
- c. One flat can be rented by multiple tenants with different leasing contracts
- d. One tenant can rent multiple flats with different leasing contracts.

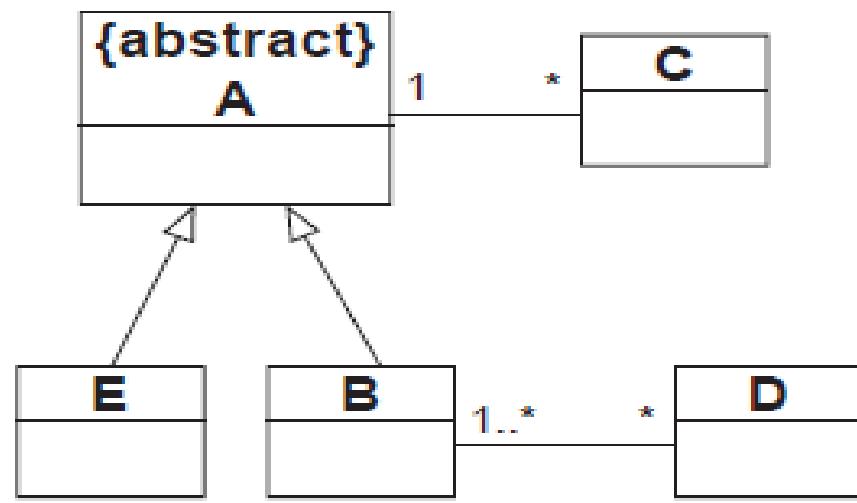
Which of the following statements about shared aggregations is true?

Selectați răspunsul corect:

- a. The multiplicity of a shared aggregation may be ≥ 1
- b. Chains of shared aggregation links may form a cycle
- c. A shared aggregation is shown by a solid-filled diamond on the end of an association line
- d. Shared aggregations are used to express an is-a relationship

What techniques can be used for identifying class operations?

- i. Class - responsibility cards
 - ii. Activity diagrams
 - iii. Sequence diagrams
 - iv. Business process diagrams
-
- a. i+iii
 - b. i+ii
 - c. ii+iv
 - d. i+ii+iii

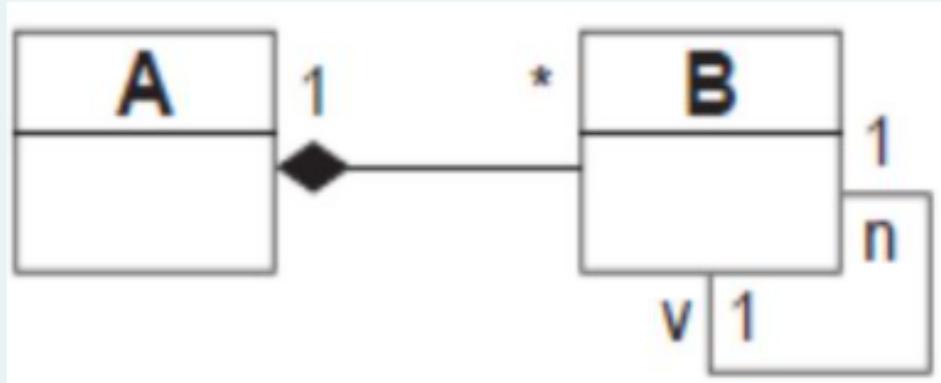


Choose one or more:

- i. One object of B is associated with `1..*` objects of D.
- ii. One object of a subclass of A is associated with `*` objects of C.
- iii. One object of D is associated with at least one object of B.

- a. ii+iii
- b. i+ii+iii+iv
- c. ii+iii+iv
- d. ii+iv

You are given the following clipping of a UML2 class diagram. Which of the following statements are true?



Selectați unul sau mai multe:

- a. One object of B is associated with two other objects of B.
- b. If an instance of B is deleted, all contained instances of A are deleted as well.
- c. one object of B is contained in exactly one object of A.
- d. One object of A may be associated with one object of B.
- e. The diamond near A is called composition.

Information system design

Lecture 4

Prof. Ramona Bologa

Dynamic analysis of the computer system

- 1.** Activity diagrams
- 2.** State chart diagram

1. Activity diagram - Content

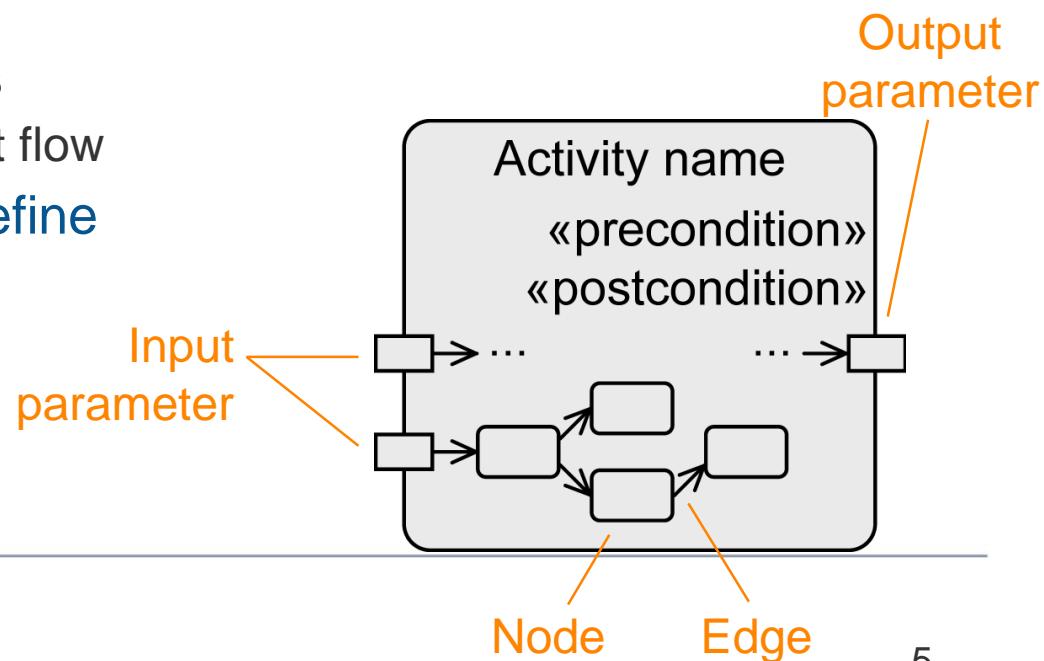
- Introduction
- Activities
- Actions
- Edges
 - Control flow
 - Object flow
- Initial node, activity final node, flow final node
- Alternative paths
- Concurrent paths
- Object nodes
- Event-based actions and call behavior actions
- Partitions

Introduction

- Focus of activity diagram: **procedural processing aspects**
- Flow-oriented language concepts
- Based on
 - languages for defining business processes
 - established concepts for describing concurrent communicating processes (token concept as found in petri nets)
- Concepts and notation variants cover **broad area of applications**
 - Modeling of object-oriented and non-object-oriented systems

Activity

- Specification of user-defined behavior at different levels of granularity
- Examples:
 - Definition of the behavior of an operation in the form of individual instructions
 - Modeling the course of actions of a use case
 - Modeling the functions of a business process
- An activity is a directed graph
 - Nodes: actions and activities
 - Edges: for control and object flow
- Control flow and object flow define the execution
- Optional:
 - Parameter
 - Pre- and postconditions

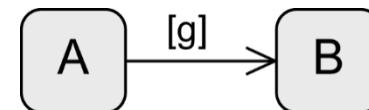
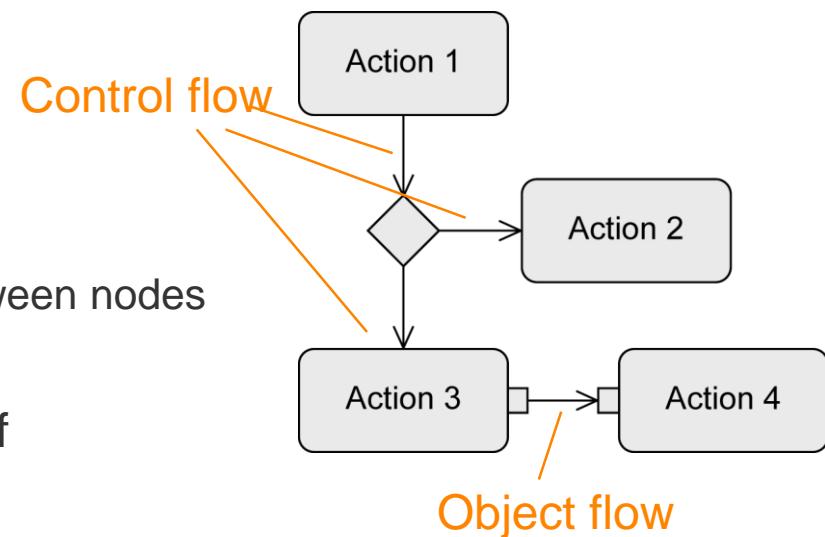


Action

- **Basic element** to specify user-defined behavior
- **Atomic** but can be aborted
- No specific rules for the description of an action
 - Definition in natural language or in any programming language
- Process input values to produce output values
- Special notation for predefined types of actions, most importantly
 - Event-based actions
 - Call behavior actions

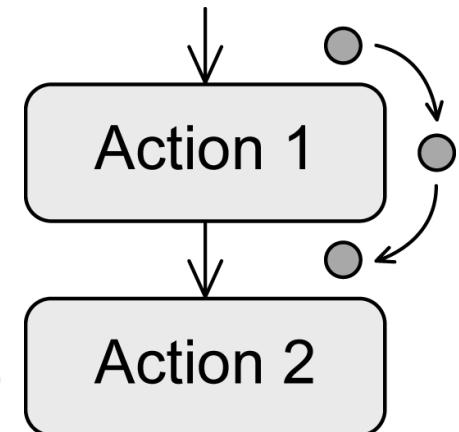
Edges

- Connect activities and actions to one another
- Express the execution order
- Types
 - Control flow edges
 - Define the order between nodes
 - Object flow edges
 - Used to exchange data or objects
 - Express a data/causal dependency between nodes
- Guard (condition)
 - Control and object flow only continue if guards in square brackets evaluate to true



Token

- **Virtual coordination mechanism** that describes the execution exactly
 - No physical component of the diagram
 - Mechanism that grants the execution permission to actions
- If an action receives a token, the action can be executed
- When the action has completed, it passes the token to a subsequent action and the execution of this action is triggered
- Guards can prevent the passing of a token
 - Tokens are stored in previous node
- Control token and object token
 - **Control token**: "execution permission" for a node
 - **Object token**: transport data + "execution permission"



Beginning and Termination of Activities

● Initial node

- Starts the execution of an activity
- Provides tokens at all outgoing edges
- Keeps tokens until the successive nodes accept them
- Multiple initial nodes to model concurrency

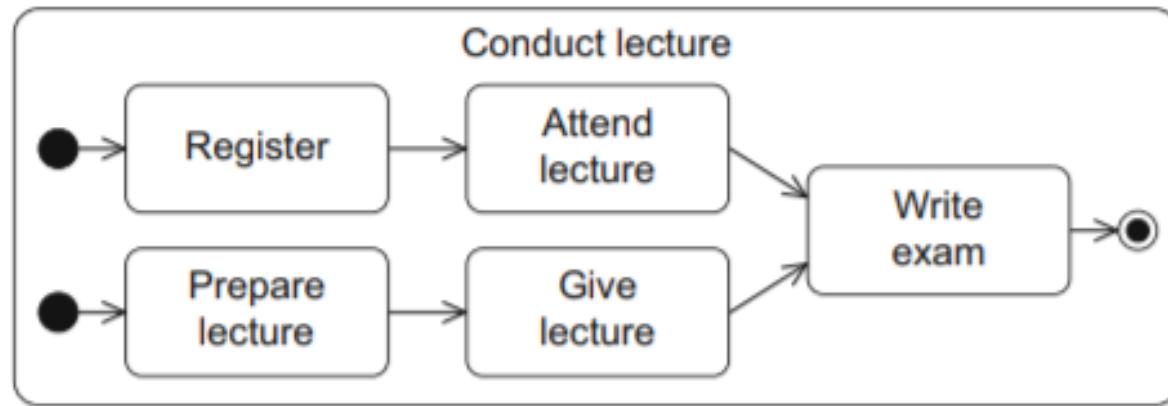
○ Activity final node

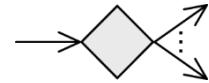
- Ends all flows of an activity
- First token that reaches the activity final node terminates the entire activity
 - Concurrent subpaths included
- Other control and object tokens are deleted
 - Exception: object tokens that are already present at the output parameters of the activity

⊗ Flow final node

- Ends one execution path of an activity
- All other tokens of the activity remain unaffected

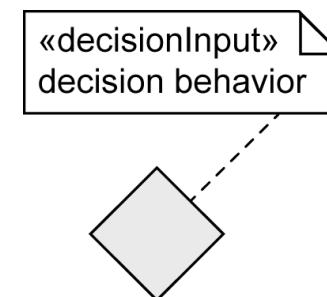
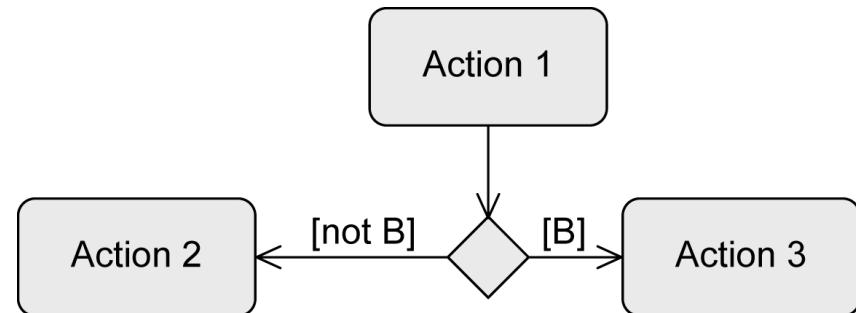
Example with multiple initial nodes

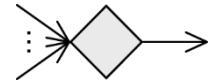




Alternative Paths – Decision Node

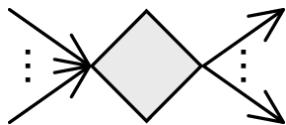
- To define alternative branches
- „Switch point“ for tokens
- Outgoing edges have guards
 - Syntax: [Boolean expression]
 - Token takes **one** branch
 - Guards must be mutually exclusive
 - Predefined: [else]
- Decision behavior
 - Specify behavior that is necessary for the evaluation of the guards
 - Execution must not have side effects



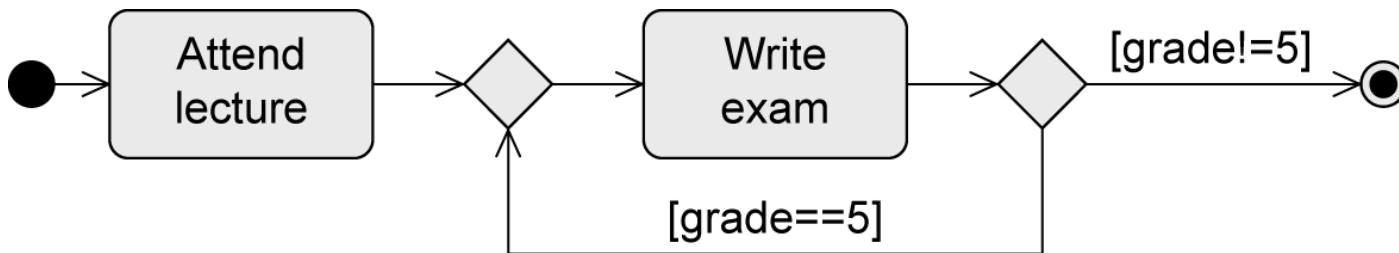


Alternative Paths – Merge Node

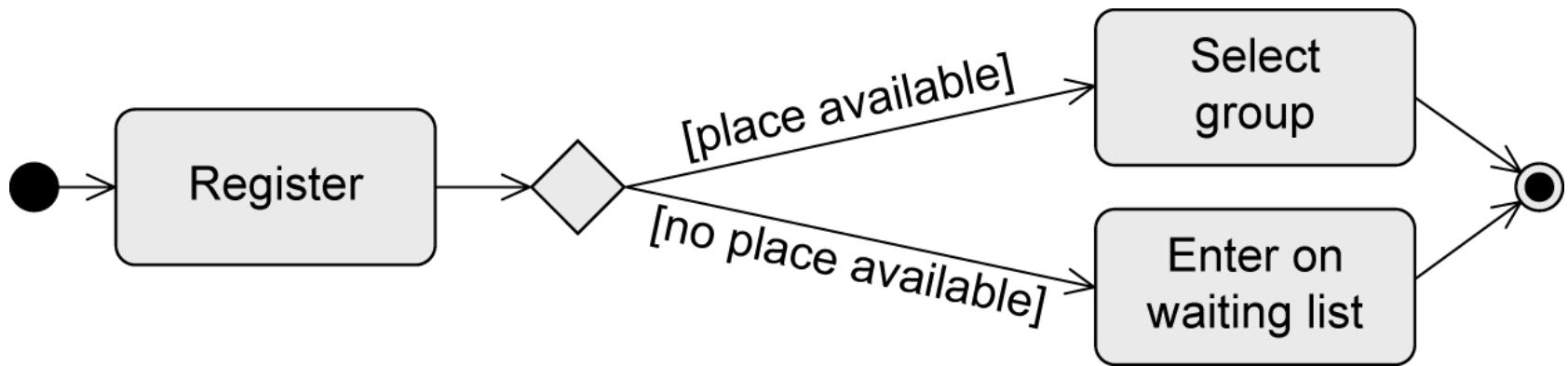
- To bring **alternative** subpaths together
- Passes token to the next node
- Combined decision and merge node

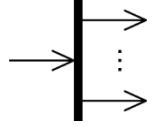


- Decision and merge nodes can also be used to model loops:



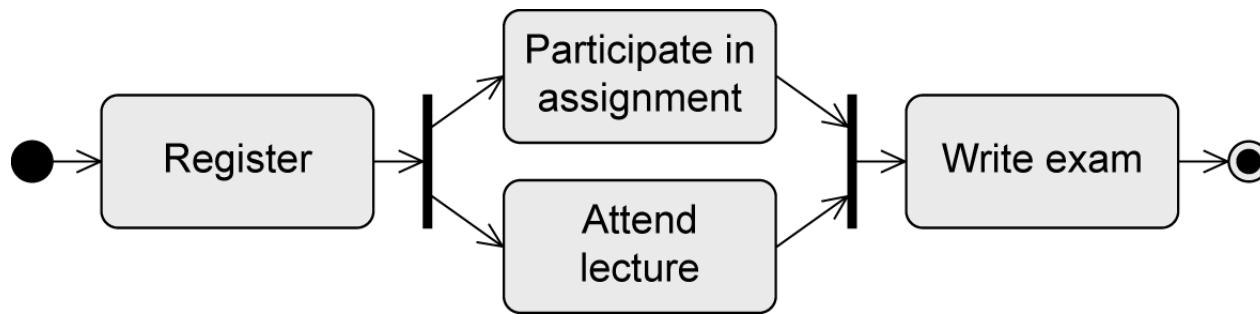
Example: Alternative Paths

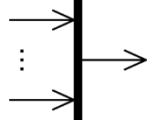




Concurrent Paths – Parallelization Node

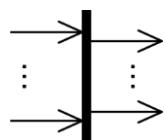
- To split path into concurrent subpaths
- Duplicates token for all outgoing edges
- Example:



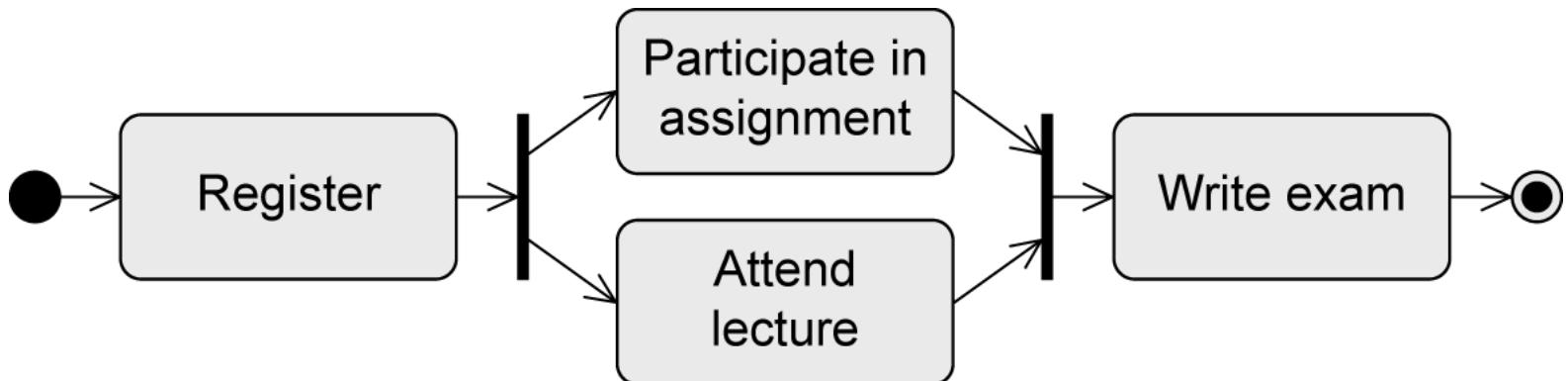


Concurrent Paths – Synchronization Node

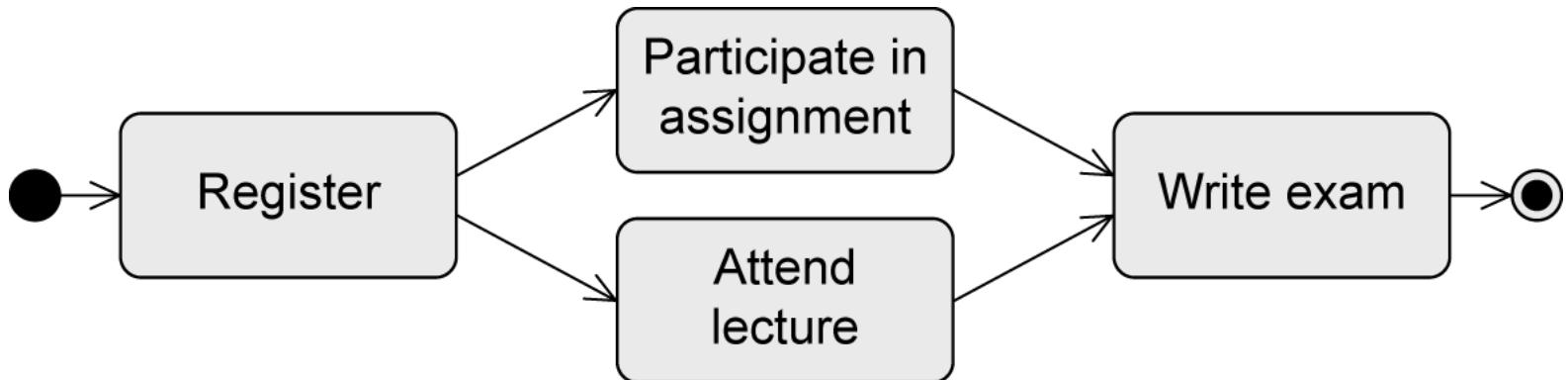
- To merge concurrent subpaths
- Token processing
 - Waits until tokens are present at all incoming edges
 - Merges all control tokens into one token and passes it on
 - Passes on all object tokens
- Combined parallelization and synchronization node:



Example: Equivalent Control Flow

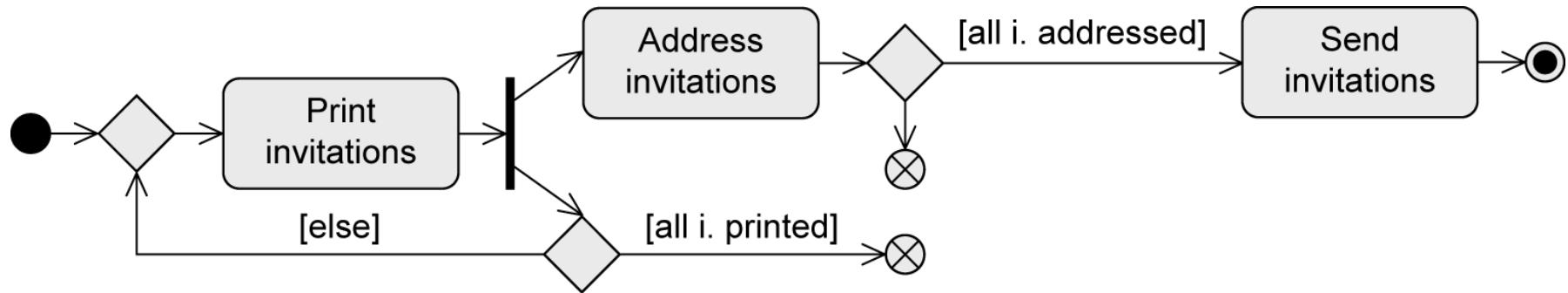


... equivalent to ...

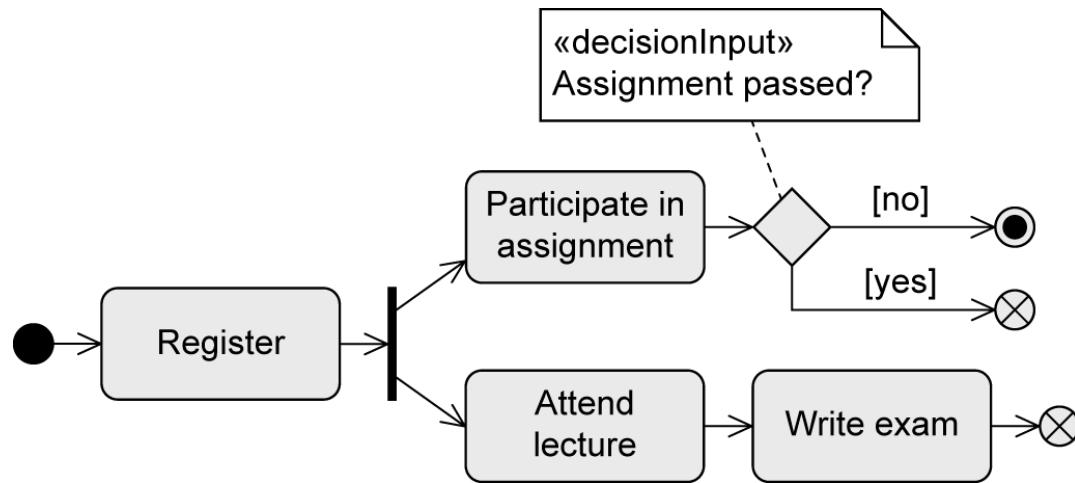


Example: Create and Send Invitations to a Meeting

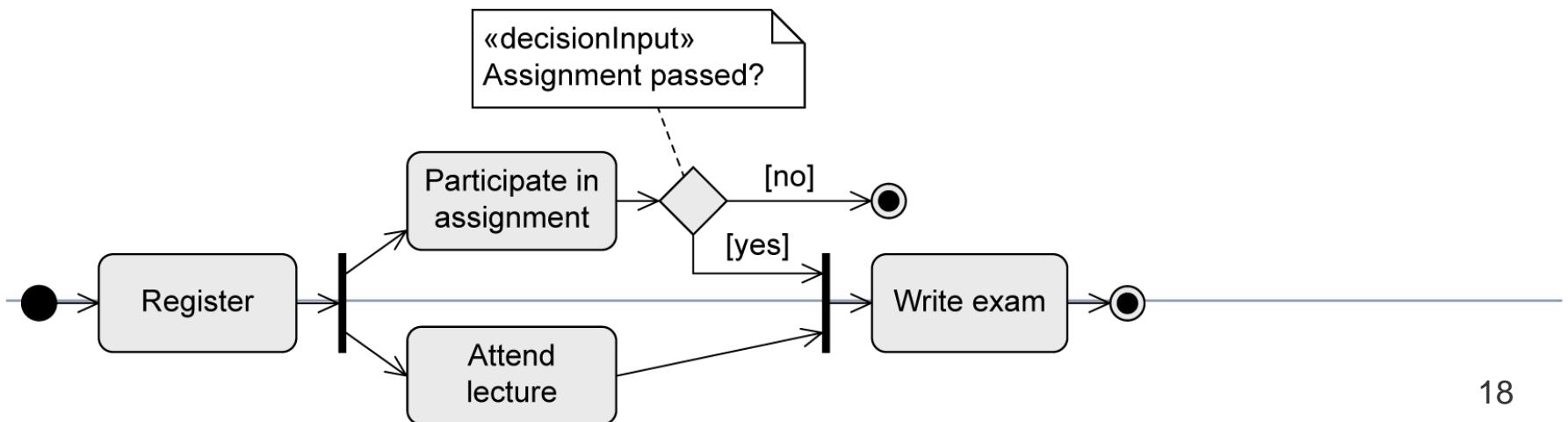
- While invitations are printed, already printed invitations are addressed.
- When all invitations are addressed, then the invitations are sent.



Example: Conduct Lecture (Student Perspective)



NOT equivalent ... why?

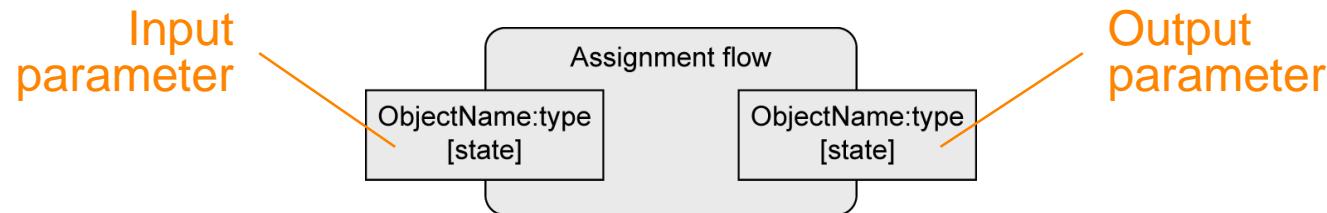


Object Node

- Contains object tokens
- Represents the exchange of data/objects
- Is the source and target of an object flow edge
- Optional information: type, state



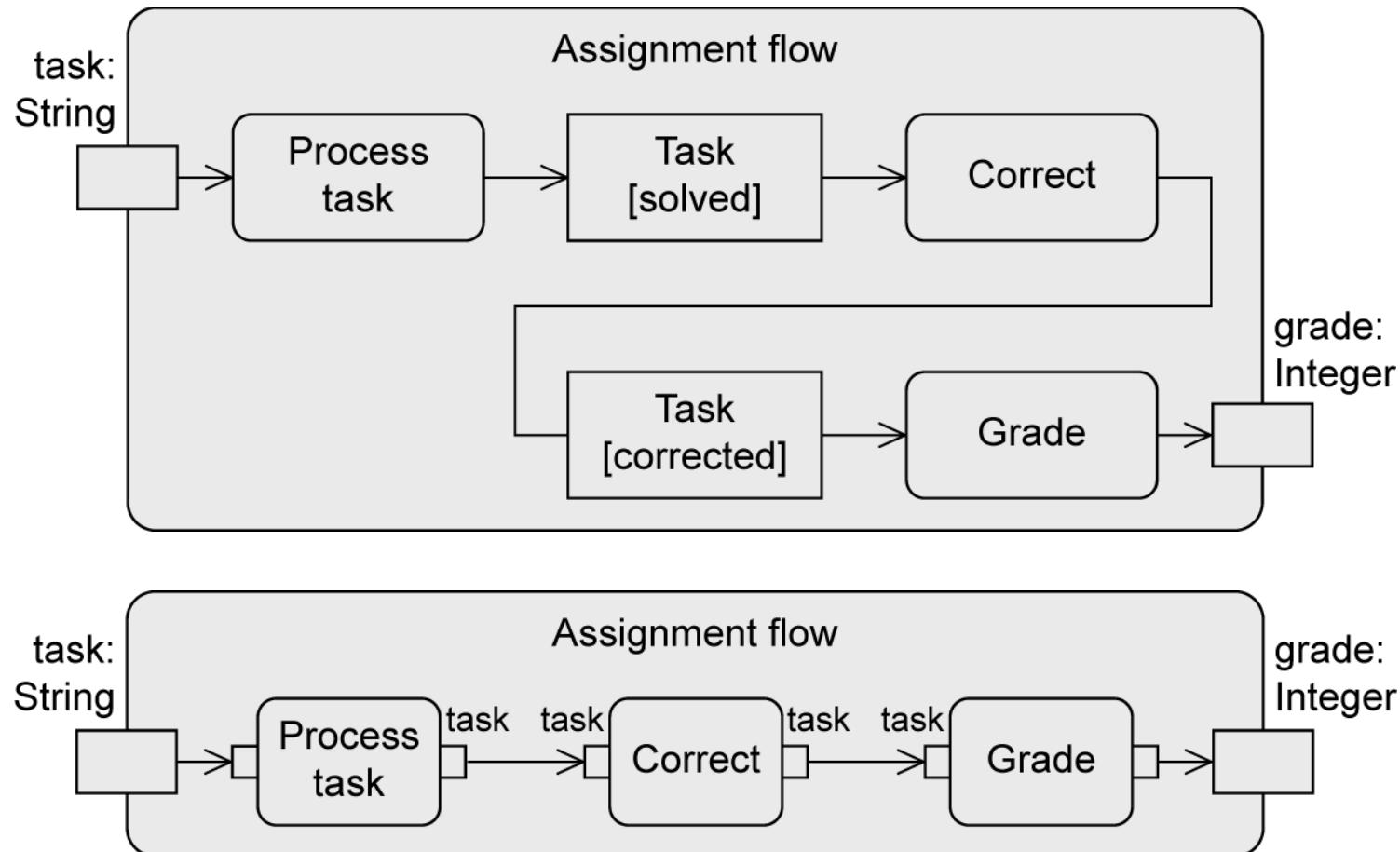
- Notation variant: object node as parameter
 - For **activities**



- For **actions** (“pins”)

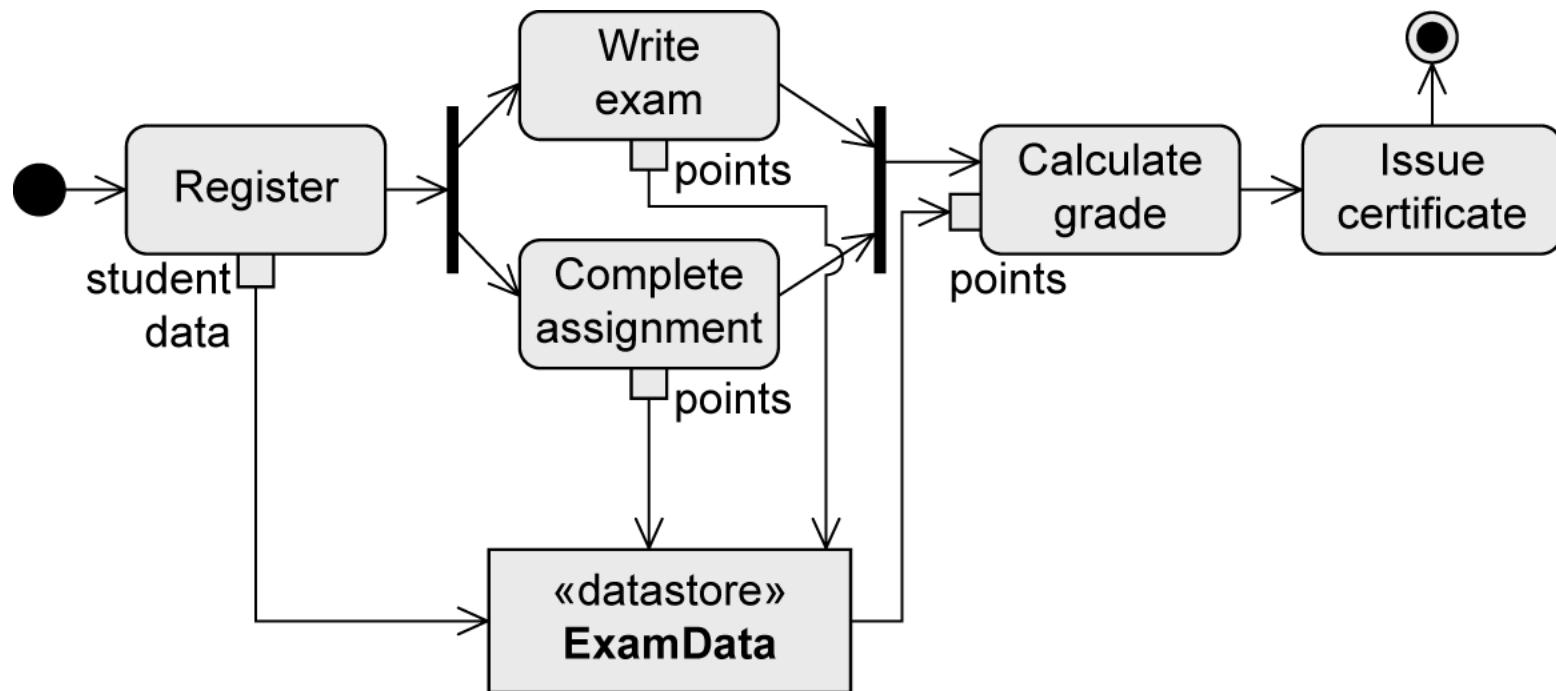


Example: Object Node

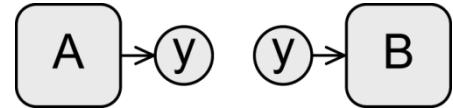


Data Store

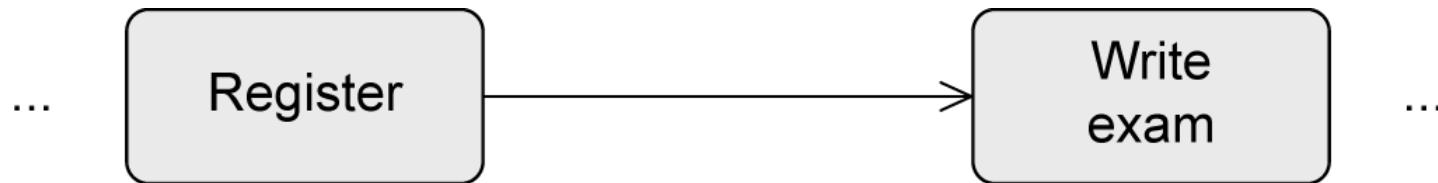
- For saving and passing on object tokens
- Permanent memory
- Saves object tokens permanently, passes copies to other nodes



Connector



- Used if two consecutive actions are far apart in the diagram
- Without connector:

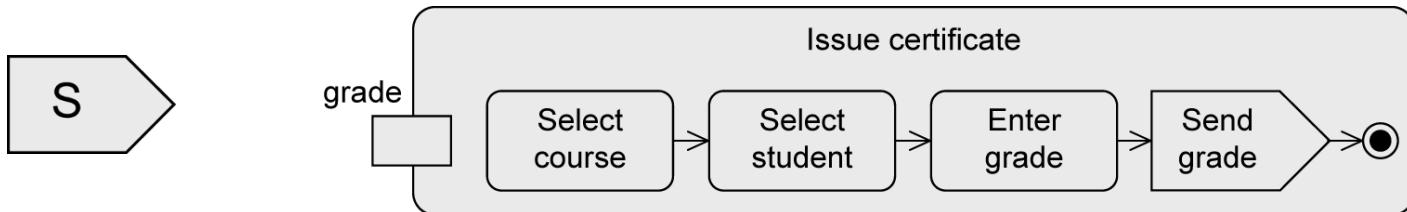


- With connector

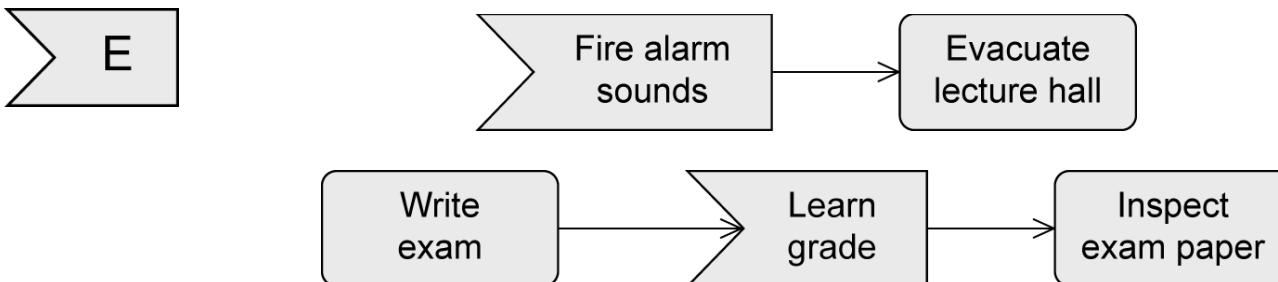


Event-Based Actions

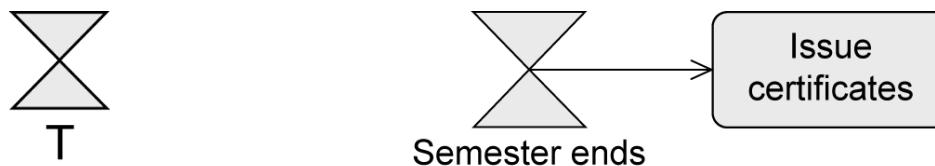
- To send signals
 - Send signal action



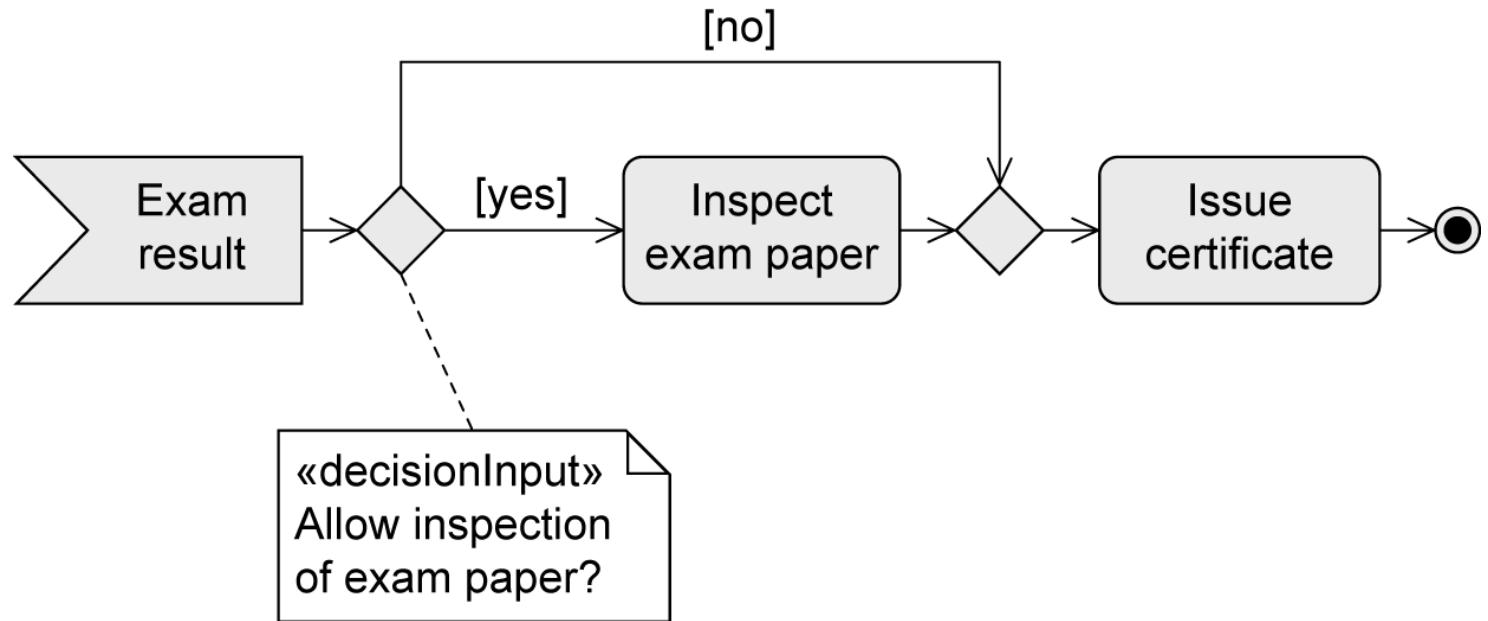
- To accept events
 - Accept event action



- Accept time event action



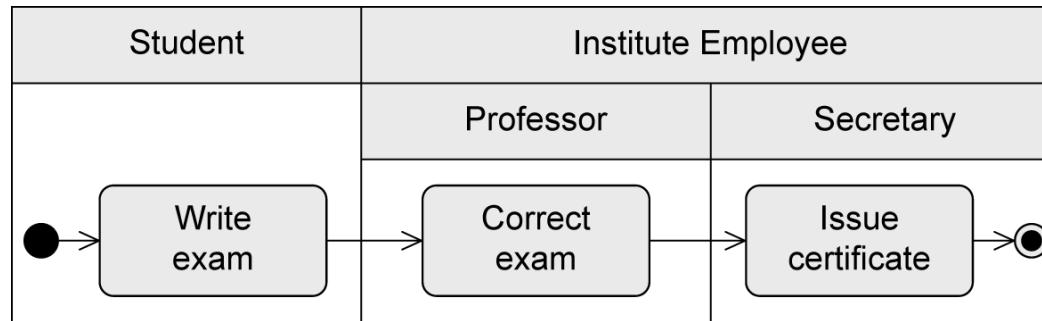
Example: Accept Event Action



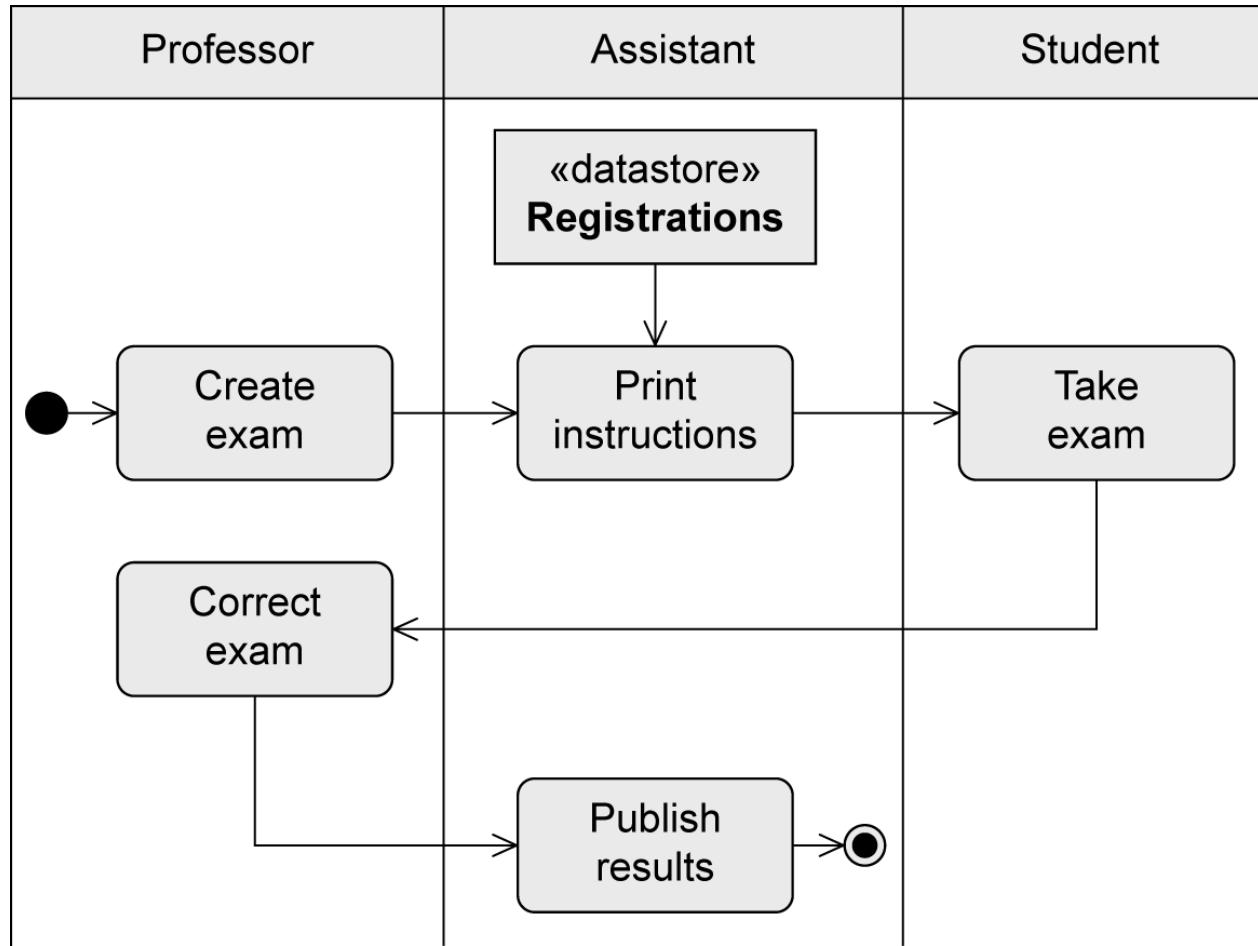
A	B	A	
	B		

Partition

- “Swimlane”
- Graphically or textual
- Allows the grouping of nodes and edges of an activity due to responsibilities
- Responsibilities reflect organizational units or roles
- Makes the diagram more structured
- Does not change the execution semantics
- Example: partitions **Student** and **Institute Employee** (with subpartitions **Professor** and **Secretary**)

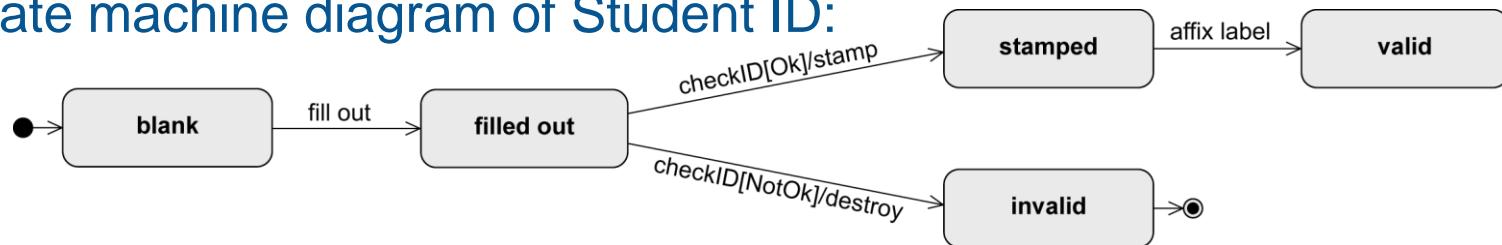


Example: Partitions

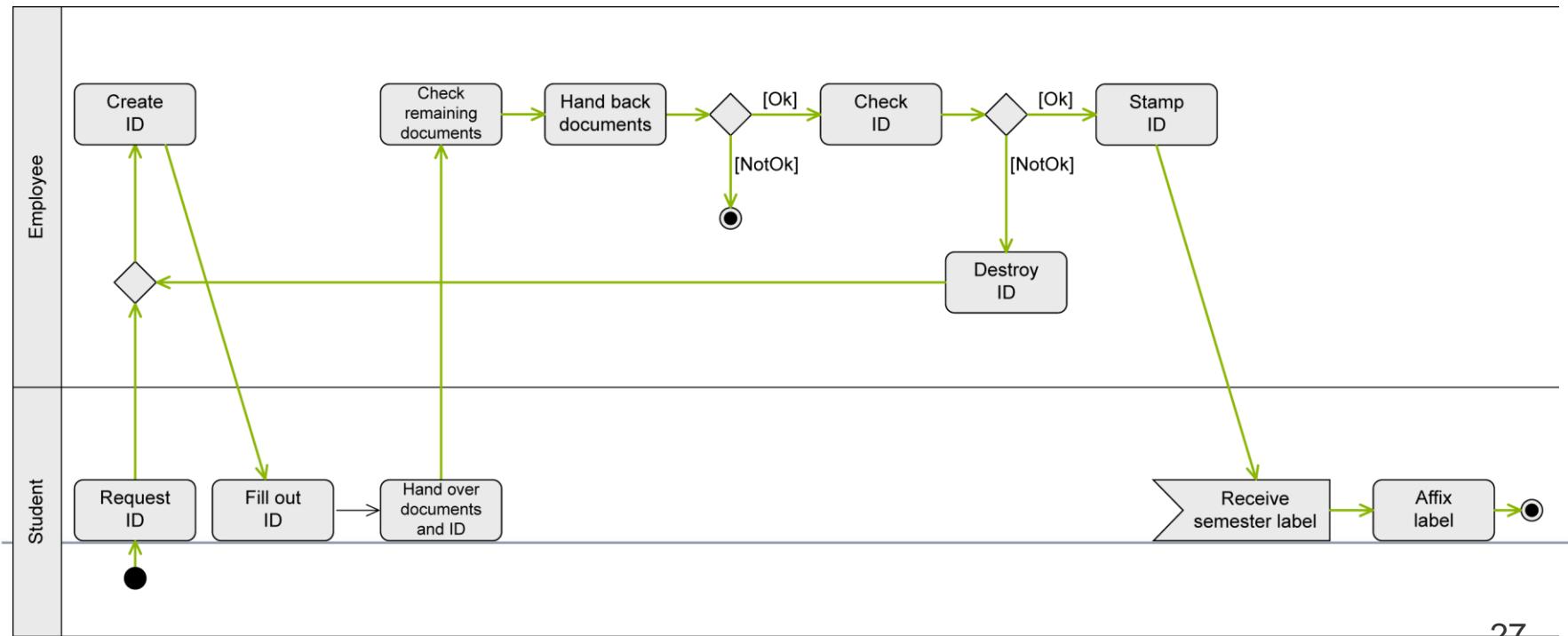


Example: Issue Student ID on Paper (1/2)

- State machine diagram of Student ID:



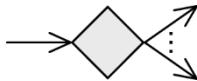
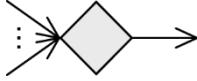
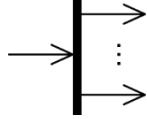
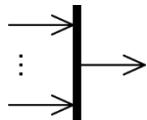
- Activity diagram – control flow:



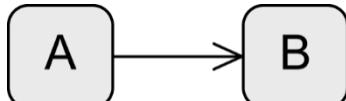
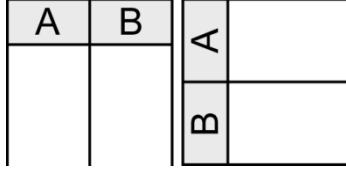
Notation Elements (1/5)

Name	Notation	Description
Action node		Represents an action (atomic!)
Activity node		Represents an activity (can be broken down further)
Initial node		Start of the execution of an activity
Activity final node		End of ALL execution paths of an activity

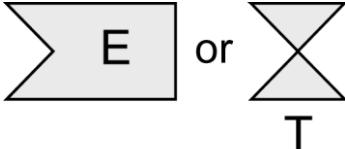
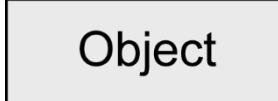
Notation Elements (2/5)

Name	Notation	Description
Decision node		Splitting of one execution path into alternative execution paths
Merge node		Merging of alternative execution paths into one execution path
Parallelization node		Splitting of one execution path into concurrent execution paths
Synchronization node		Merging of concurrent execution paths into one execution path

Notation Elements (3/5)

Name	Notation	Description
Flow final node	⊗	End of ONE execution path of an activity
Edge		Connection between the nodes of an activity
Partition		Grouping of nodes and edges within an activity

Notation Elements (4/5)

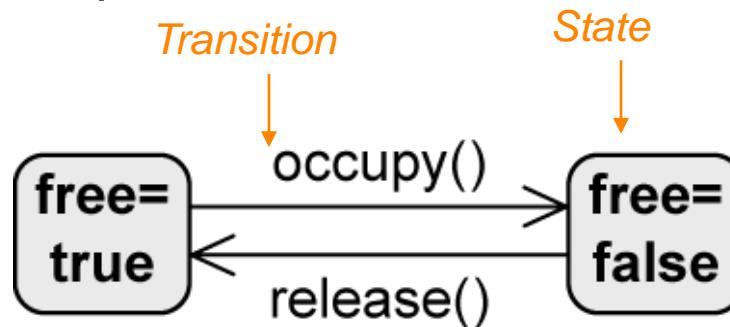
Name	Notation	Description
Send signal action		Transmission of a signal to a receiver
Asynchronous accept (timing) event action		Wait for an event E or a time event T
Object node		Contains data or objects
Parameter for activities		Contains data and objects as input and output parameters
Parameter for actions (pins)		

State machine diagrams

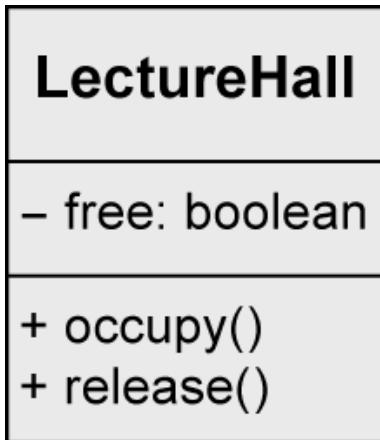
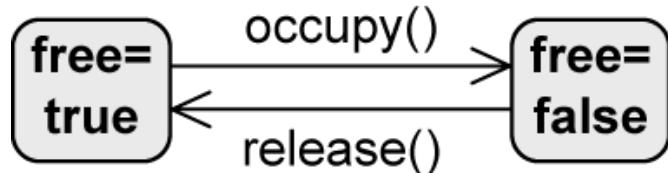
- Introduction
- States
- Transitions
- Types of events
- Types of states
- Entry and exit points

Introduction

- Every object takes a finite number of different states during its life
- According to UML, a state is „a condition during an object's life when it satisfies some criterion, performs some action, or waits for an event”.
- State machine diagram is used as follows:
 - To model the possible states of a system or object
 - To show how state transitions occur as a consequence of events
 - To show what behavior the system or object exhibits in each state
- Example: high-level description of the behavior of a lecture hall



Example: Lecture Hall with Details

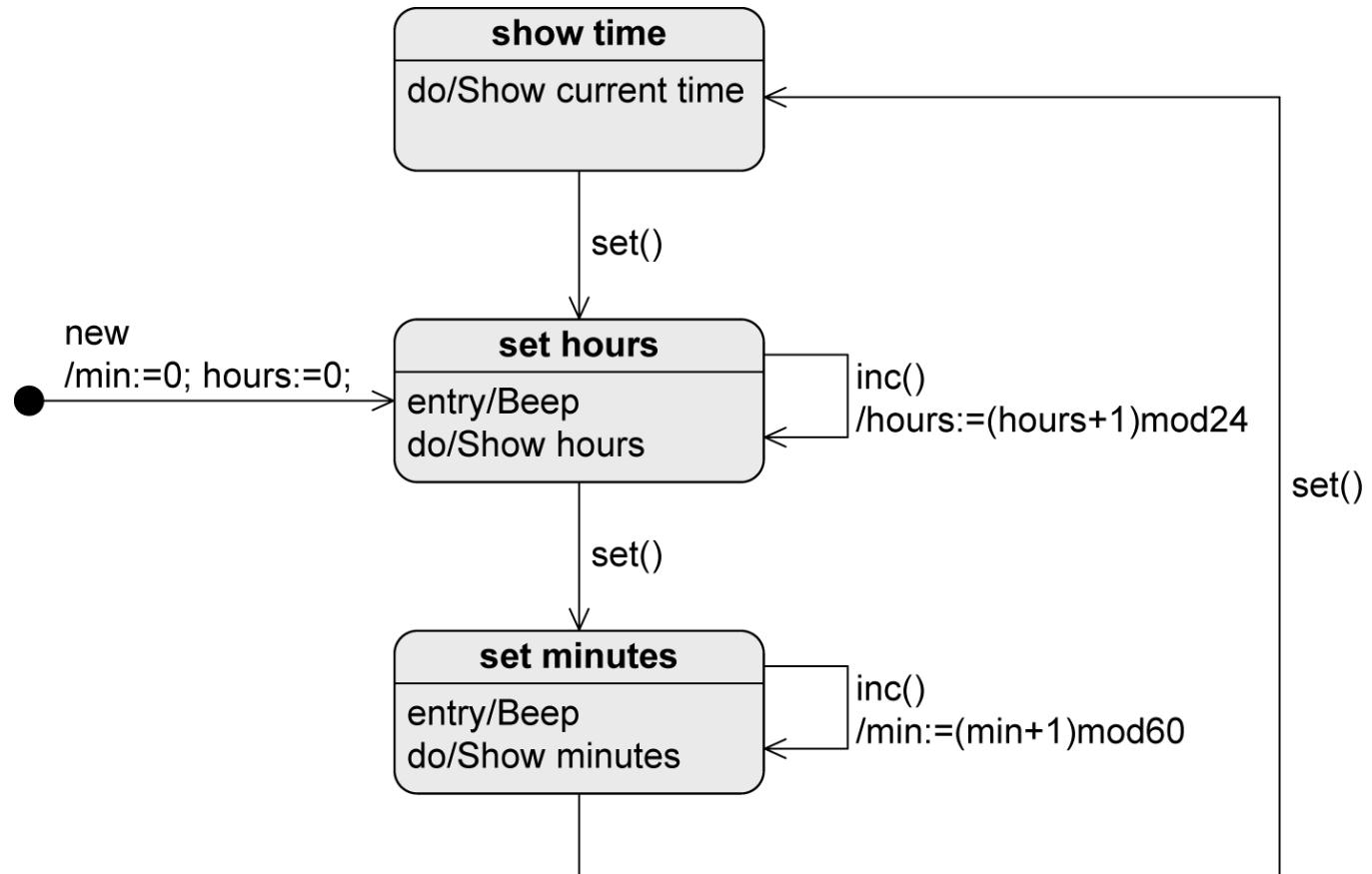


```
class LectureHall {  
    private boolean free;  
  
    public void occupy() {  
        free=false;  
    }  
    public void release() {  
        free=true;  
    }  
}
```

00 : 00 • set
• inc

Example: Digital Clock

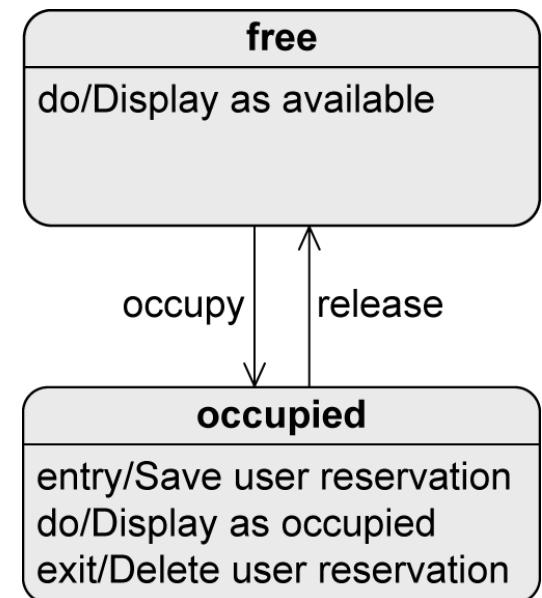
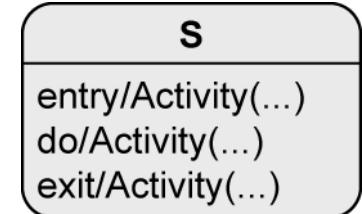
DigitalClock
- min: int
- hours: int
+ set(): void
+ inc(): void

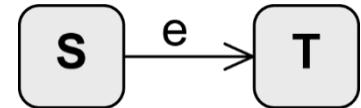


S

State

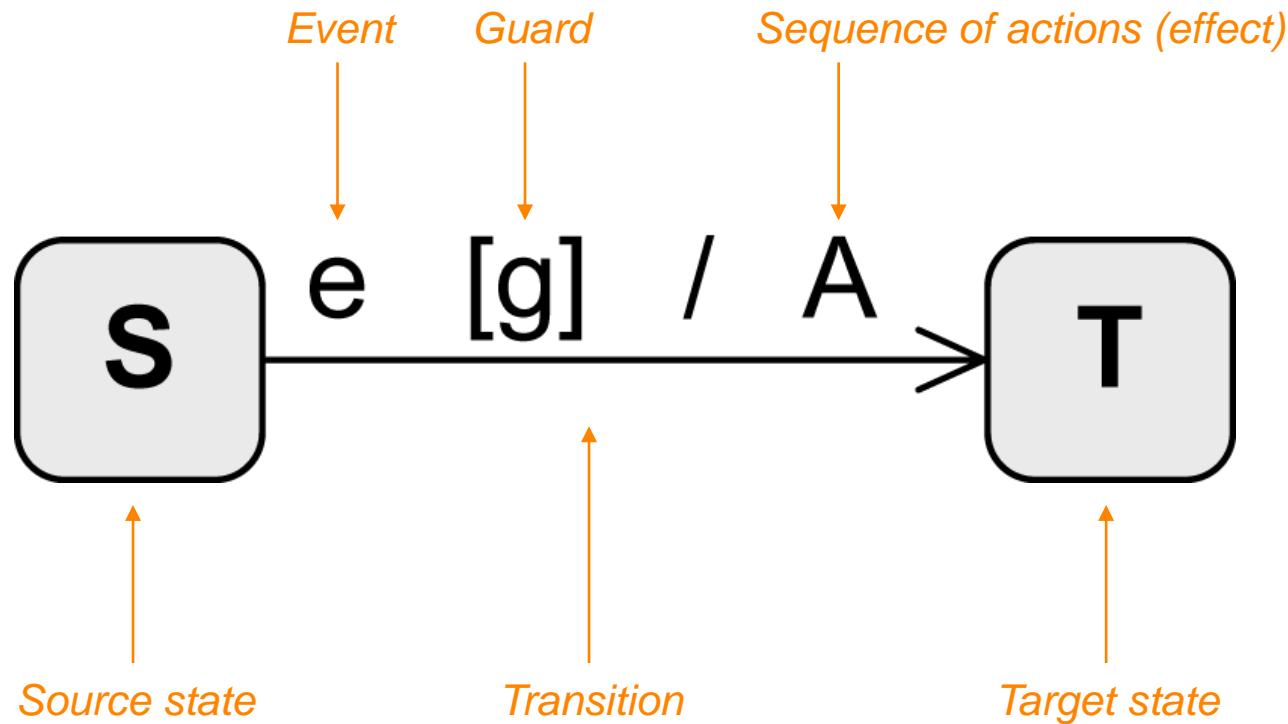
- States = nodes of the state machine
- When a state is active
 - The object is in that state
 - All internal activities specified in this state can be executed
 - An activity can consist of multiple actions
- entry / Activity(...)
 - Executed when the object enters the state
- exit / Activity(...)
 - Executed when the object exits the state
- do / Activity(...)
 - Executed while the object remains in this state



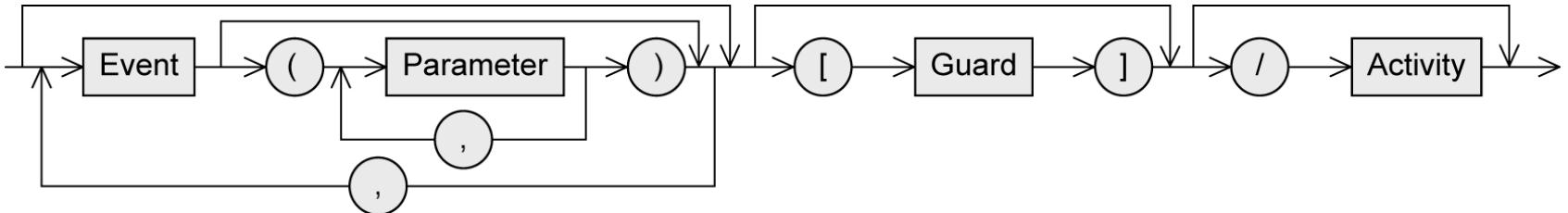


Transition

- Change from one state to another



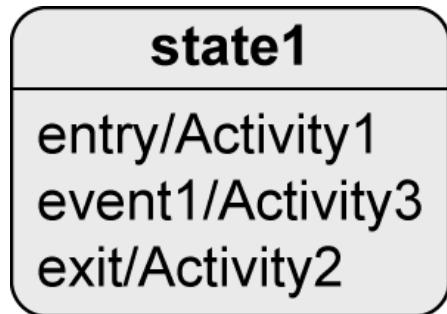
Transition – Syntax



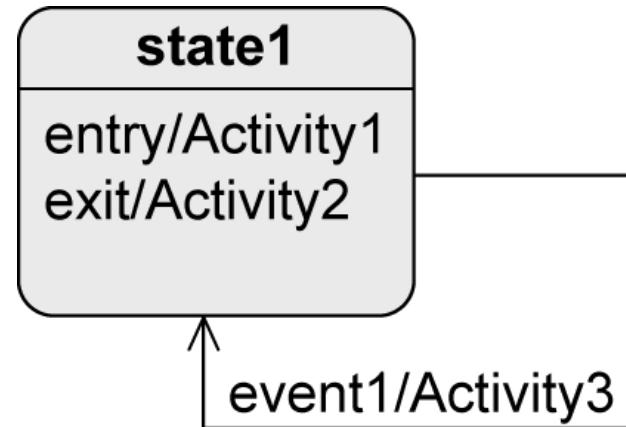
- **Event (trigger)**
 - Exogenous stimulus
 - Can trigger a state transition
- **Guard (condition)**
 - Boolean expression
 - If the event occurs, the guard is checked
 - If the guard is true
 - All activities in the current state are terminated
 - Any relevant exit activity is executed
 - The transition takes place
 - If the guard is false
 - No state transition takes place, the event is discarded
- **Activity (effect)**
 - Sequence of actions executed during the state transition

Transition – Types (1/2)

Internal transition



External transition

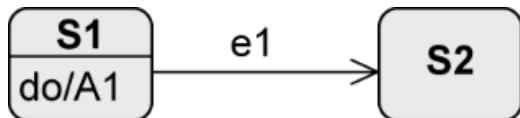


- If **event1** occurs
 - Object remains in **state1**
 - **Activity3** is executed

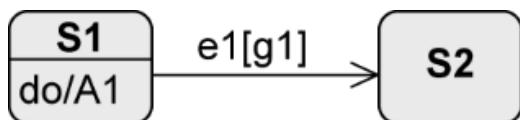
- If **event1** occurs
 - Object leaves **state1** and **Activity2** is executed
 - **Activity3** is executed
 - Object enters **state1** and **Activity1** is executed

Transition – Types (2/2)

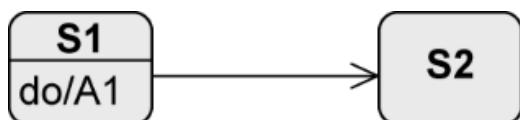
- When do the following transitions take place?



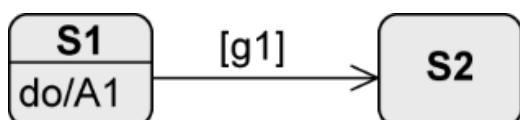
If **e1** occurs, **A1** is aborted and the object changes to **S2**



If **e1** occurs and **g1** evaluates to true, **A1** is aborted and the object changes to **S2**



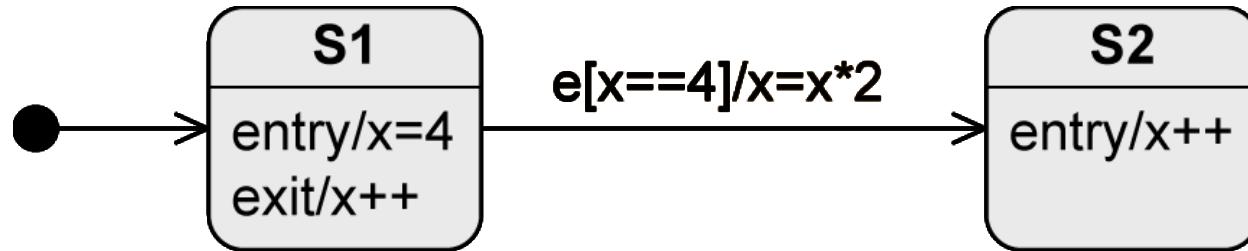
As soon as the execution of **A1** is finished, a **completion event** is generated that initiates the transition to **S2**



As soon as the execution of **A1** is finished, a completion event is generated; if **g1** evaluates to true, the transition takes place; If not, this transition can never happen

Transition – Sequence of Activity Executions

- Assume **S1** is active ... what is the value of **x** after **e** occurred?



S1 becomes active, **x** is set to the value **4**

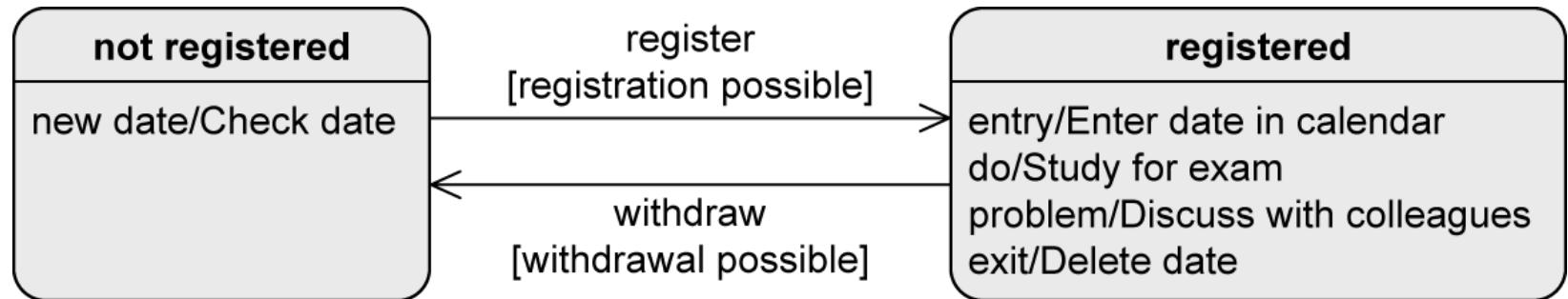
e occurs, the guard is checked and evaluates to true

S1 is left, **x** is set to **5**

The transition takes place, **x** is set to **10**

S2 is entered, **x** is set to **11**

Example: Registration Status of an Exam



Event – Types (1/2)

- **Signal event**

Receipt of a signal

- E.g., `rightmousedown`, `sendSMS(message)`

- **Call event**

Operation call

- E.g., `occupy(user, lectureHall)`, `register(exam)`

- **Time event**

Time-based state transition

- Relative: based on the time of the occurrence of the event
 - E.g., `after(5 seconds)`
 - Absolute
 - E.g., `when(time==16:00)`, `when(date==20150101)`

Event – Types (2/2)

- **Any receive event**

Occurs when any event occurs that does not trigger another transition from the active state

- Keyword `all`

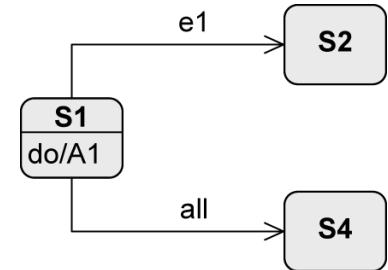
- **Completion event**

Generated automatically when everything to be done in the current state is completed

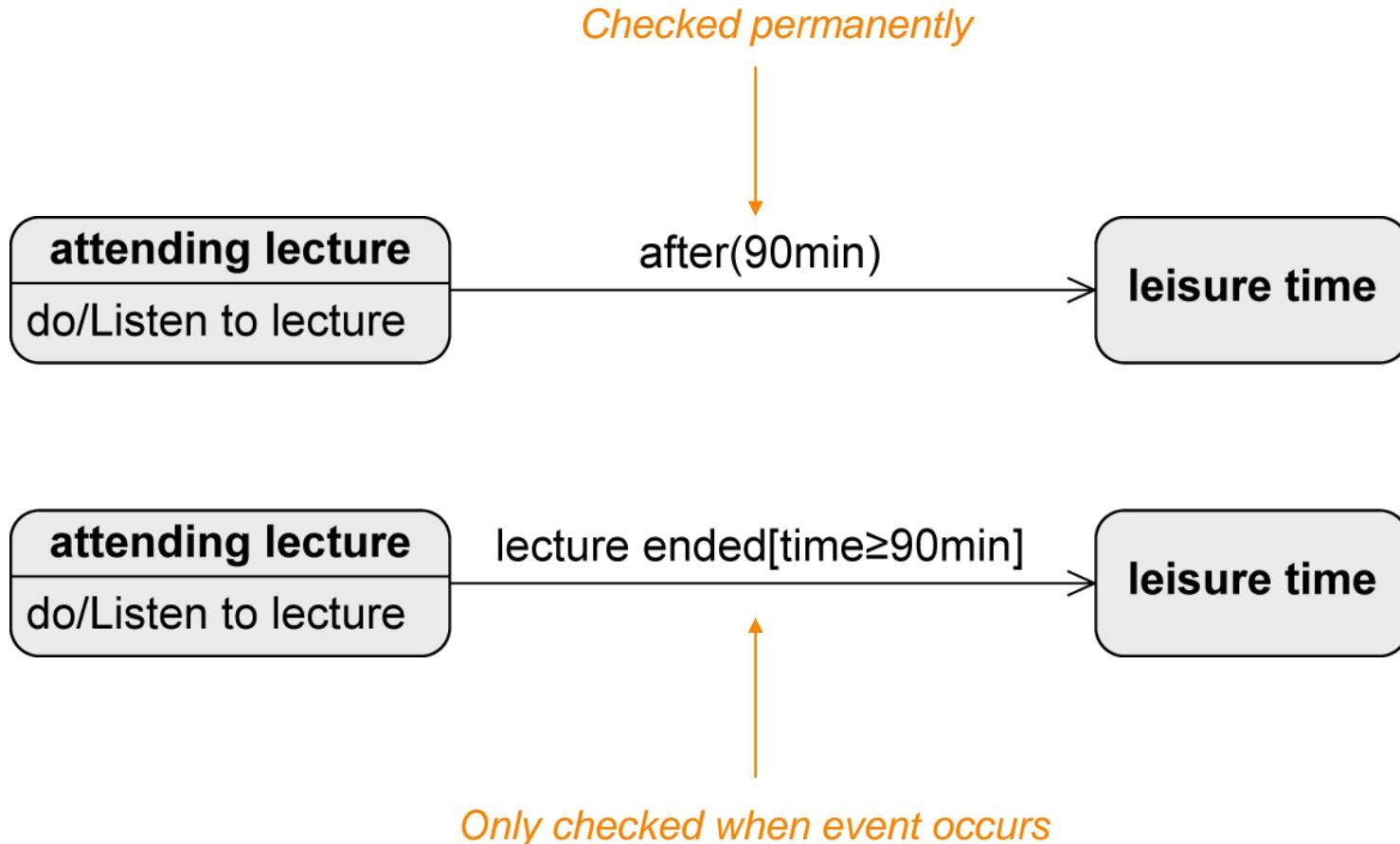
- **Change event**

Permanently checking whether a condition becomes true

- E.g., `when (x > y), after(90min)`



Change Event vs. Guard



Question: What if the lecture is shorter than 90min?

Initial State

- “Start” of a state machine diagram
- **Pseudostate**
 - Transient, i.e., system cannot remain in that state
 - Rather a control structure than a real state
- No incoming edges
- If >1 outgoing edges
 - Guards must be mutually exclusive and cover all possible cases to ensure that exactly one target state is reached
- If initial state becomes active, the object immediately switches to the next state
 - No events allowed on the outgoing edges (exception: `new()`)

Final State and Terminate Node

Final State

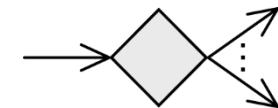


- **Real state**
- Marks the end of the sequence of states
- Object can remain in a final state forever

Terminate Node

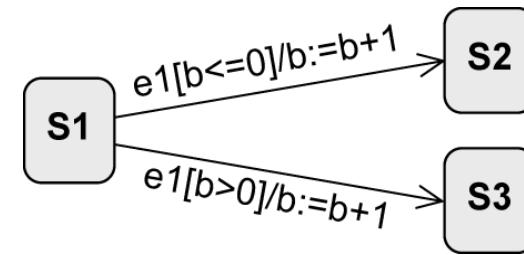
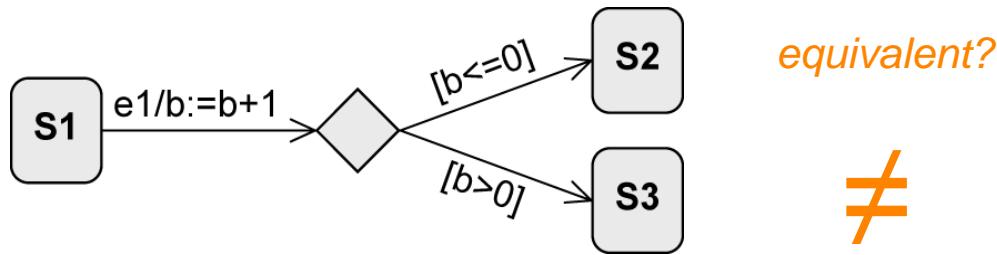
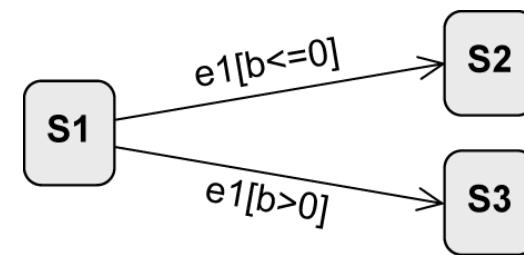
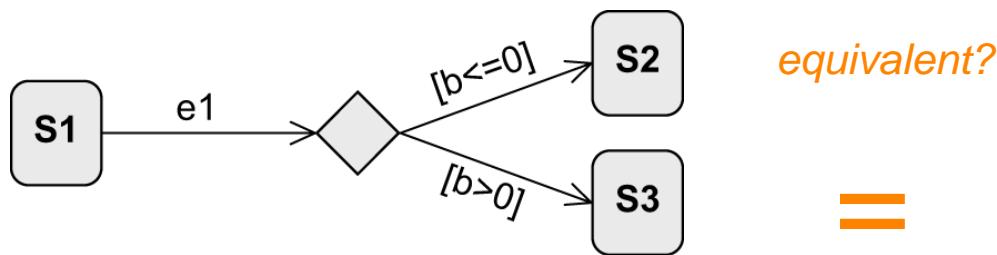


- **Pseudostate**
- Terminates the state machine
- The modeled object ceases to exist (= is deleted)

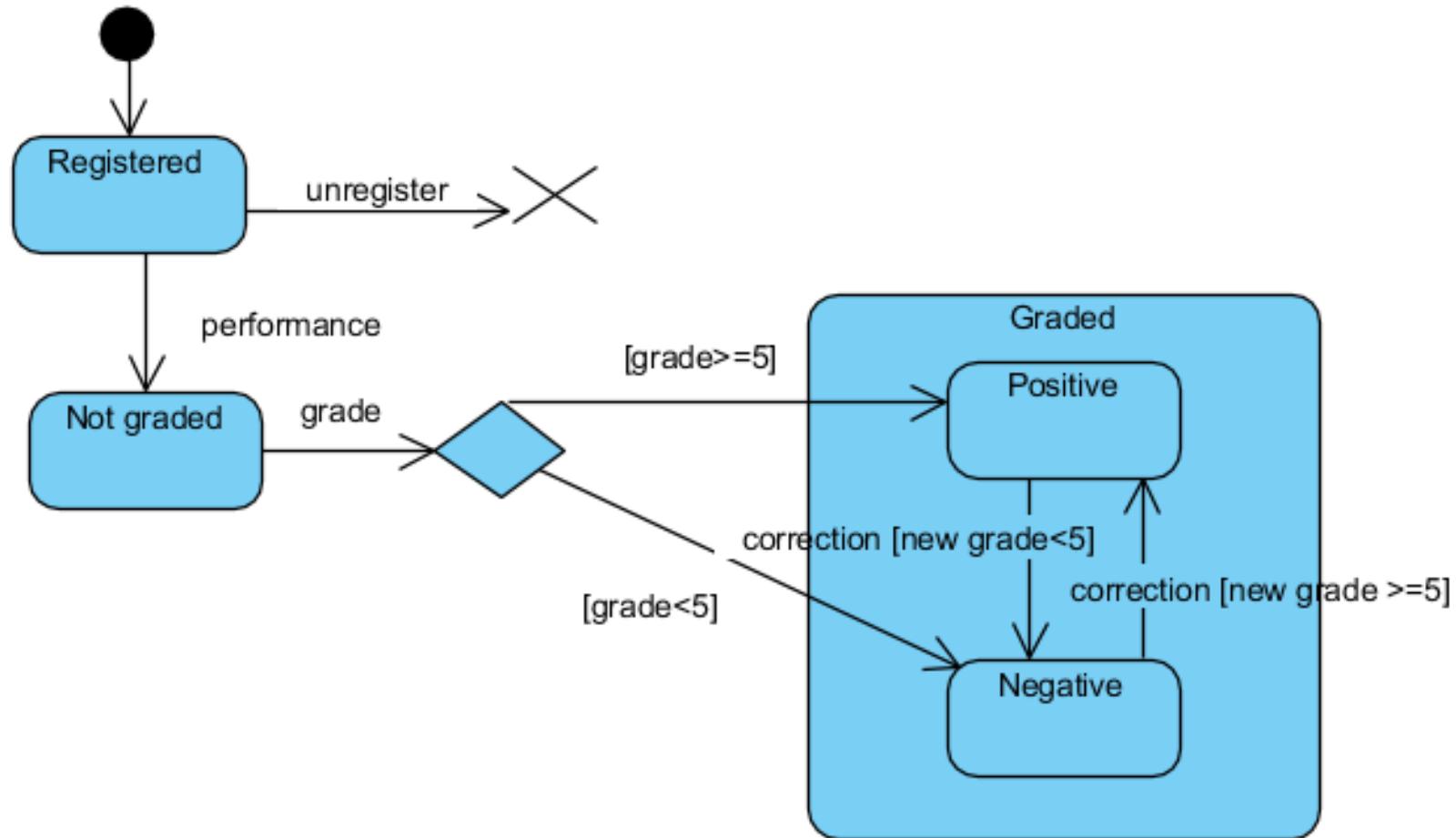


Decision Node

- Pseudostate
- Used to model alternative transitions



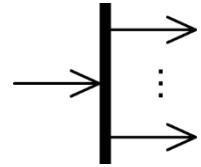
Example: Decision Node



Parallelization and Synchronization Node

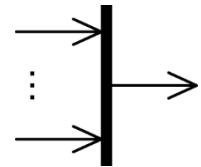
Parallelization node

- Pseudostate
- Splits the control flow into multiple concurrent flows
- 1 incoming edge
- >1 outgoing edges



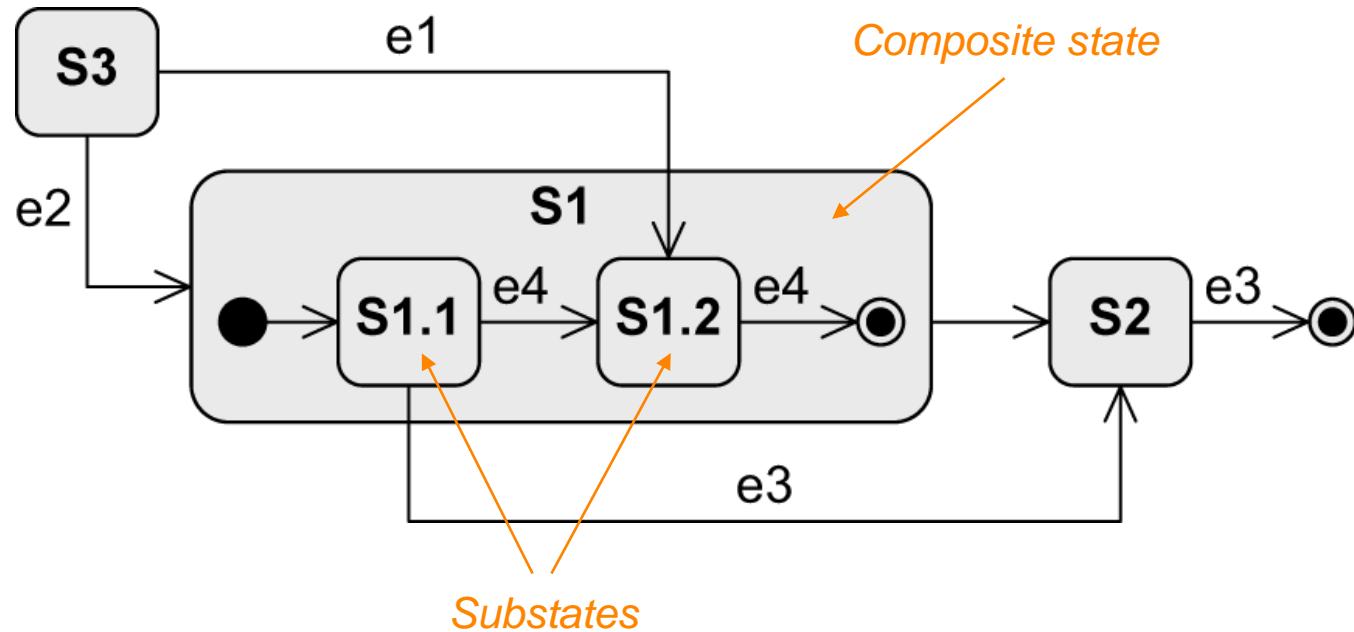
Synchronization node

- Pseudostate
- Merges multiple concurrent flows
- >1 incoming edges
- 1 outgoing edge



Composite State

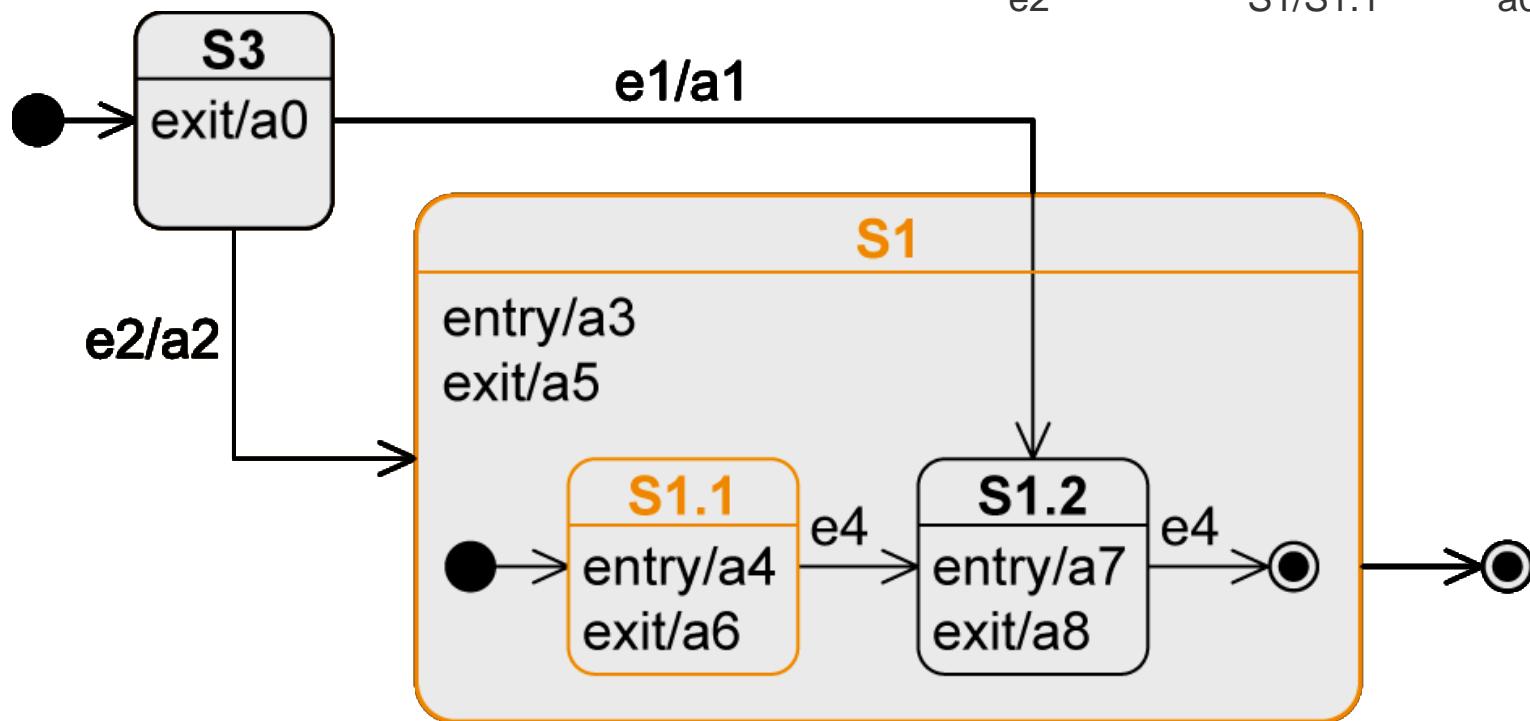
- Synonyms: complex state, nested state
- Contains other states – “substates”
 - Only one of its substates is active at any point in time
- Arbitrary nesting depth of substates



Entering a Composite State (1/2)

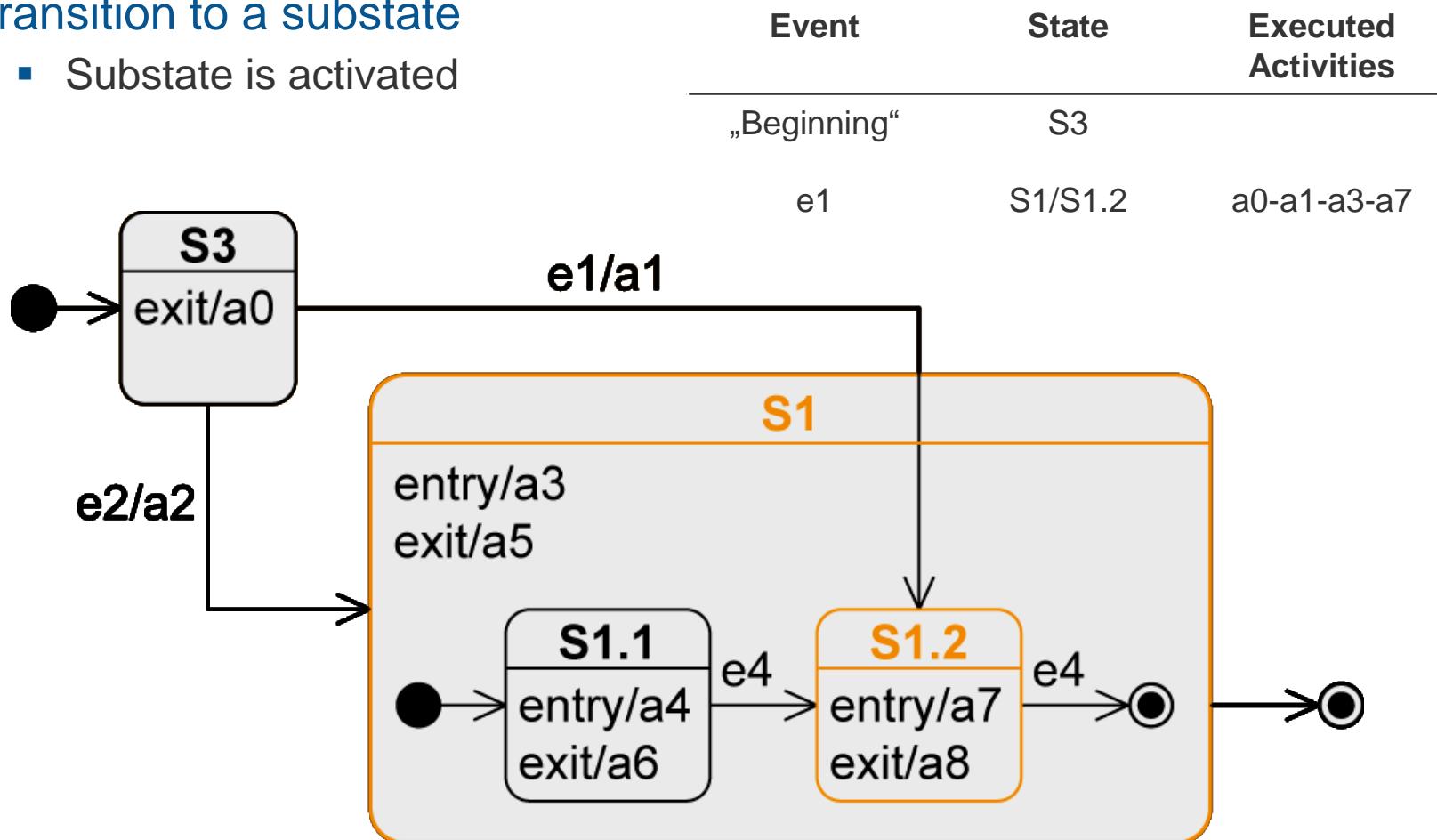
- Transition to the boundary
 - Initial node of composite state is activated

Event	State	Executed Activities
„Beginning“	S3	
e2	S1/S1.1	a0-a2-a3-a4



Entering a Composite State (2/2)

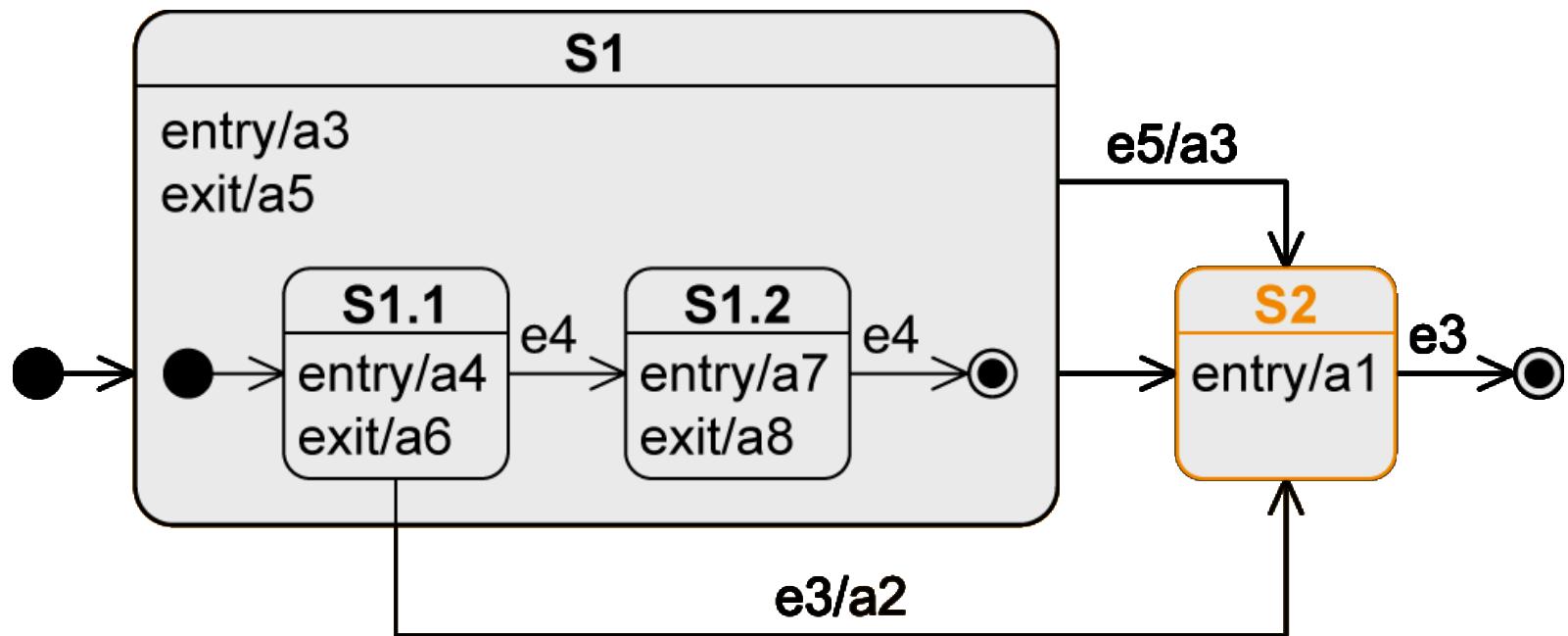
- Transition to a substate
 - Substate is activated



Exiting from a Composite State (1/3)

- Transition from a substate

Event	State	Executed Activities
„Beginning“	S1/S1.1	a3-a4
e3	S2	a6-a5-a2-a1

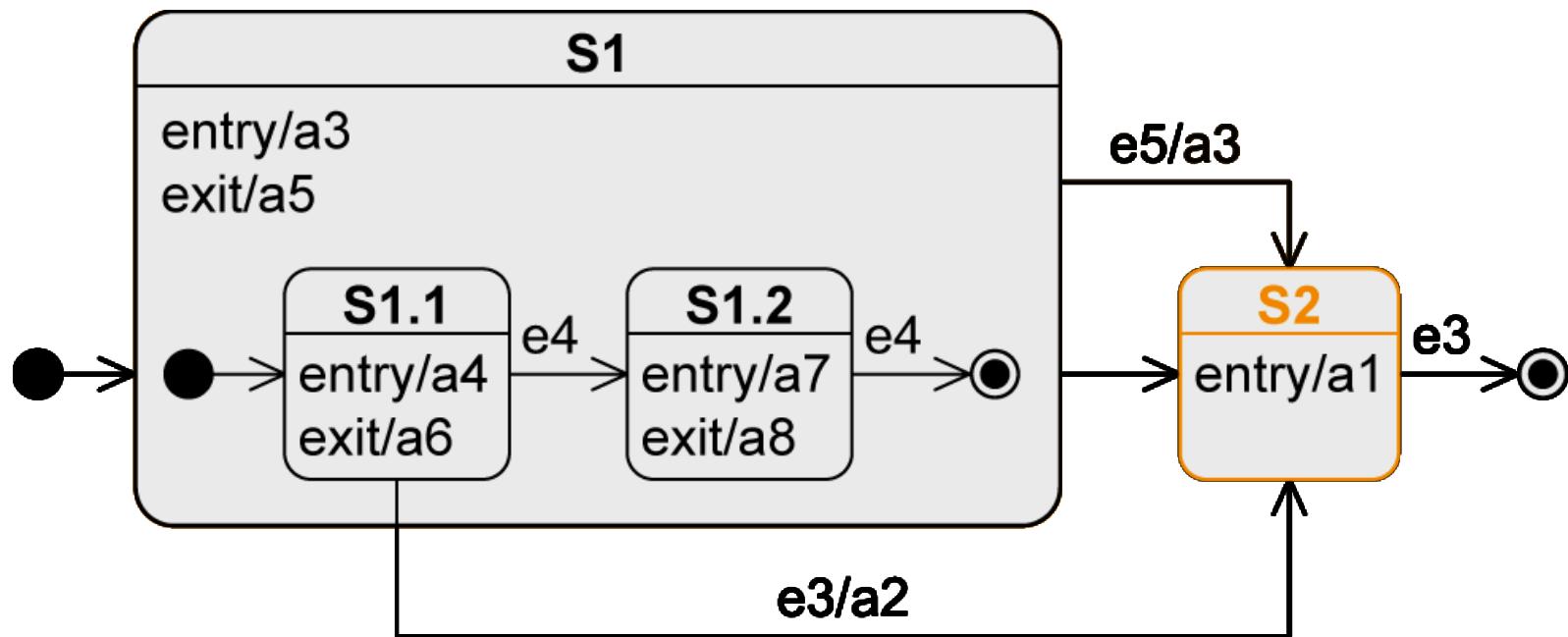


Exiting from a Composite State (2/3)

- Transition from the composite state

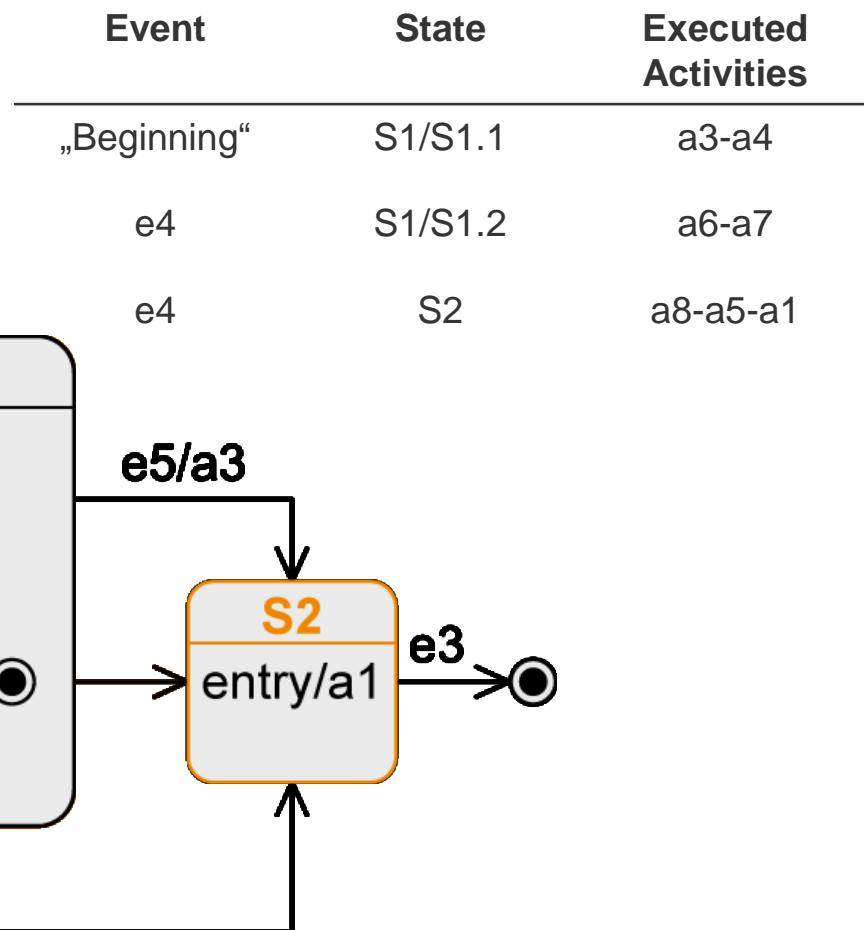
No matter which substate of S1 is active, as soon as e5 occurs, the system changes to S2

Event	State	Executed Activities
„Beginning“	S1/S1.1	a3-a4
e5	S2	a6-a5-a3-a1



Exiting from a Composite State (3/3)

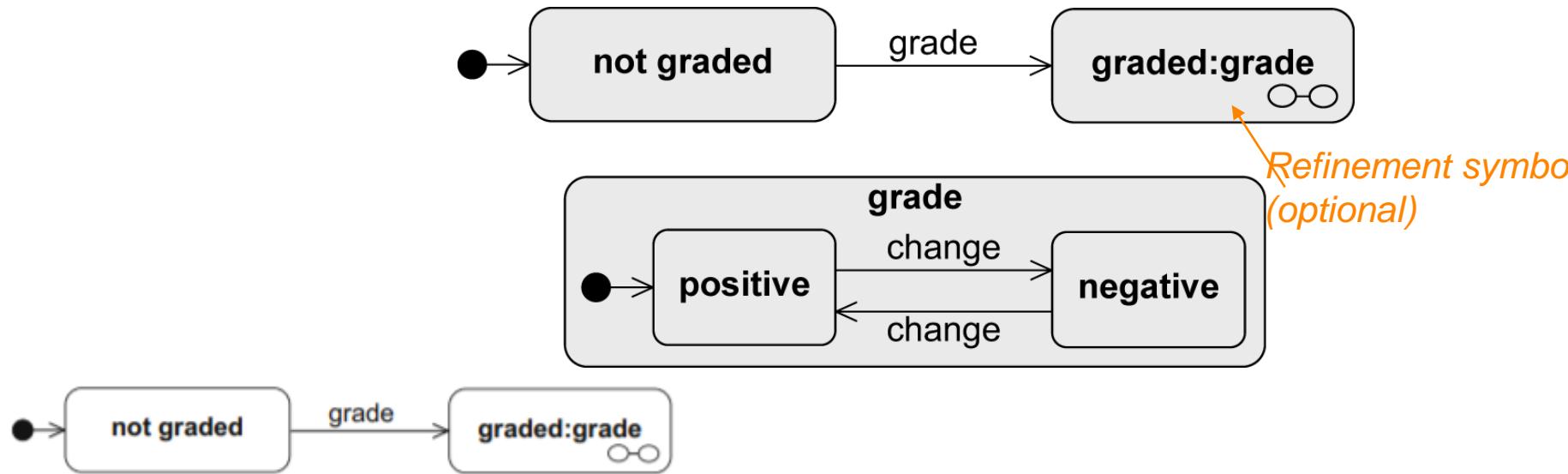
- Completion transition from the composite state





Submachine State (SMS)

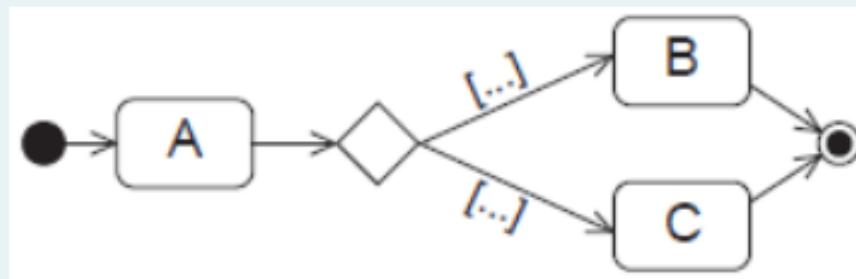
- To reuse parts of state machine diagrams in other state machine diagrams
- Notation: **state : submachineState**
- As soon as the submachine state is activated, the behavior of the submachine is executed
 - Corresponds to calling a subroutine in programming languages



Notation Elements (1/2)

Name	Notation	Description
State		Description of a specific “time span” in which an object finds itself during its “life cycle”. Within a state, activities can be executed by the object.
Transition		State transition e from a source state S to a target state T
Initial state		Start of a state machine diagram
Final state		End of a state machine diagram
Terminate node		Termination of an object’s state machine diagram
Decision node		Node from which multiple alternative transitions can origin

You are given the following activity diagram. Which of the following action sequences are possible during one execution of the activity diagram?

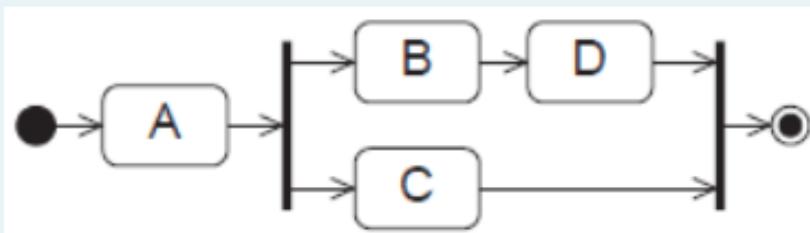


- i. A → B → C
- ii. A → C → B
- iii. A → B
- iv. A → C

Select one:

- a. iii+iv
- b. ii+iv
- c. i+ii
- d. i+iii+iv

You are given the following activity diagram. Which of the following action sequences are possible during one execution of the activity diagram?



- i. A → B → C → D
- ii. A → B → D → C
- iii. A → C → B → D
- iv. A → B → D
- v. A → C

Select one:

- a. i+ii
- b. i+iv+v
- c. i+ii+iii
- d. i+ii+iv

What does the syntax for labeling a transition look like?

Select one:

- a. [effect]event/guard
- b. effect[guard]/event
- c. [guard]effect/event
- d. event[guard]/effect
- e. [effect]guard/event

A fork node of a UML2 activity diagram ...

- i. ... is used for modeling parallel threads.
- ii. ... produces tokens for all outgoing edges.
- iii. ... deletes tokens which are not accepted.
- iv ... is only valid when combined with a join node.
- v. ... is an alternative to the decision node.

Select one:

- a. i+ii
- b. i+iv
- c. i+ii+iii+iv
- d. i+ii+iii

5th Lecture - Dynamic analysis of the computer system

Interaction diagrams:

- Sequence diagrams
- Other types of interaction diagrams

Content

- Introduction
- Interactions and interaction partners
- Messages
- Combined fragments
 - Branches and loops
 - Concurrency and order
 - Filters and assertions
- Further types of interaction diagrams

Introduction

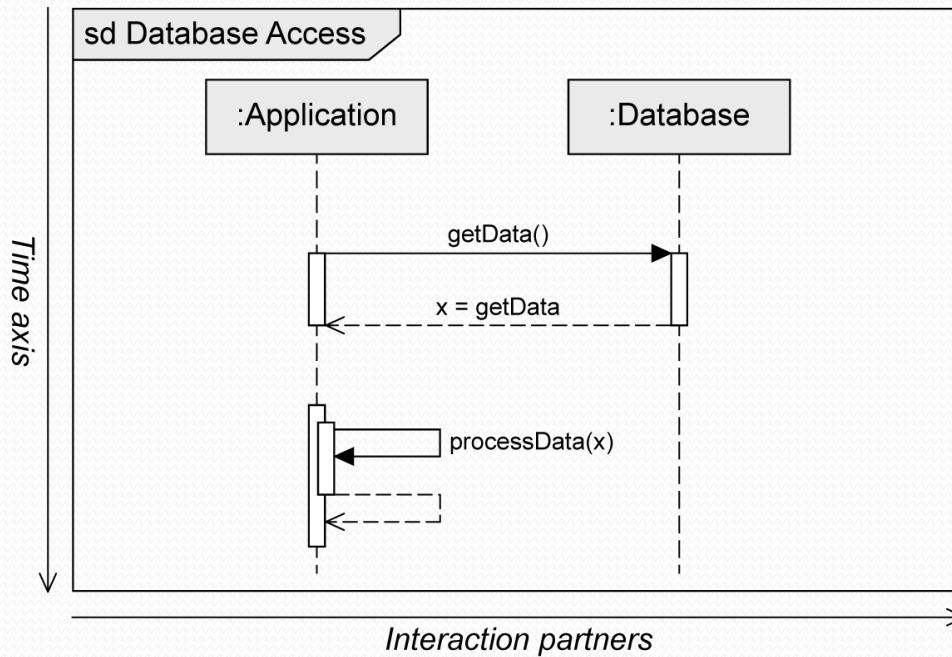
- Modeling inter-object behavior
 - = interactions between objects
- Interaction
 - Specifies how messages and data are exchanged between interaction partners
- Interaction partners
 - Human (lecturer, administrator, ...)
 - Non-human (server, printer, executable software, ...)
- Examples of interactions
 - Conversation between persons
 - Message exchange between humans and a software system
 - Communication protocols
 - Sequence of method calls in a program
 - ...

Interaction Diagrams

- Used to specify interactions
- Modeling concrete scenarios
- Describing communication sequences at different levels of detail
- Interaction Diagrams show the following:
 - Interaction of a system with its environment
 - Interaction between system parts in order to show how a specific use case can be implemented
 - Interprocess communication in which the partners involved must observe certain protocols
 - Communication at class level (operation calls, inter-object behavior)

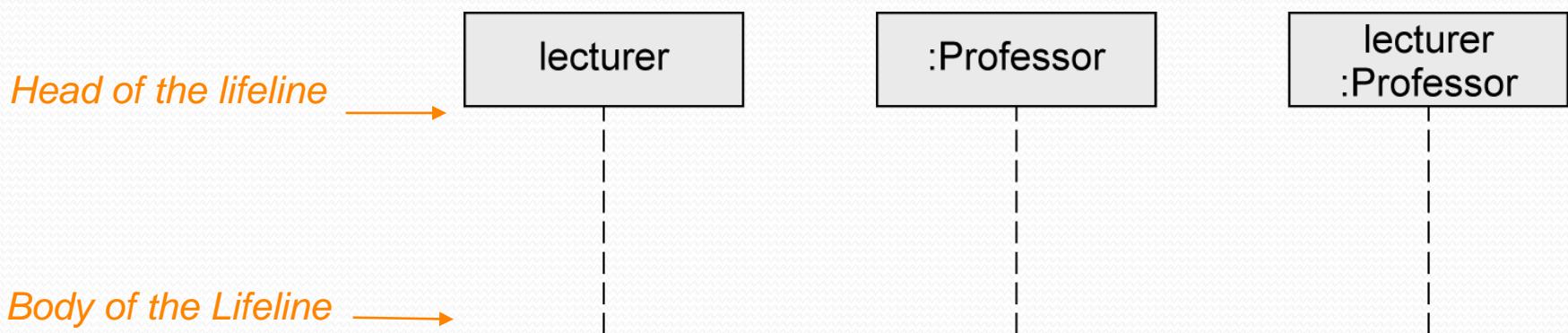
Sequence Diagram

- Two-dimensional diagram
 - Horizontal axis: involved interaction partners
 - Vertical axis: chronological order of the interaction
- Interaction = sequence of event specifications



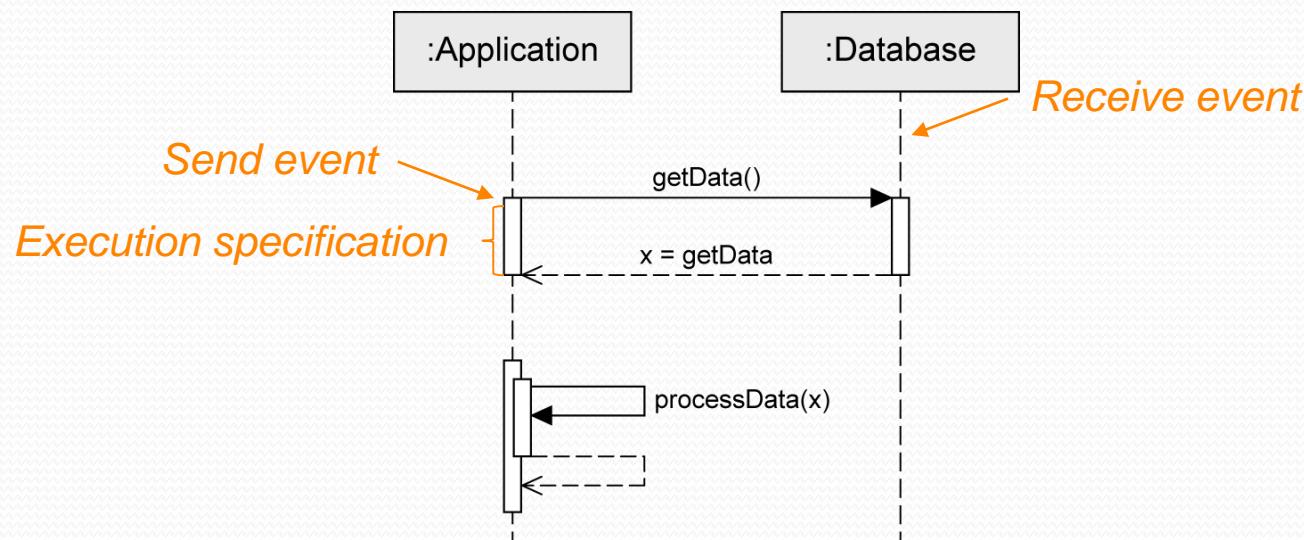
Interaction Partners

- Interaction partners are depicted as lifelines
- Head of the lifeline
 - Rectangle that contains the expression **roleName:Class**
 - Roles are a more general concept than objects
 - Object can take on different roles over its lifetime
- Body of the lifeline
 - Vertical, usually dashed line
 - Represents the lifetime of the object associated with it



Exchanging Messages (1/2)

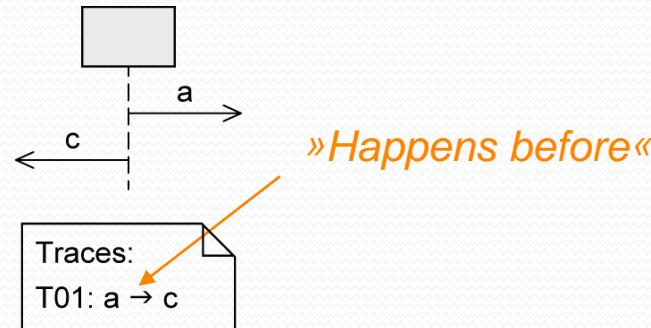
- Interaction: sequence of events
- Message is defined via send event and receive event
- Execution specification
 - Continuous bar
 - Used to visualize when an interaction partner executes some behavior



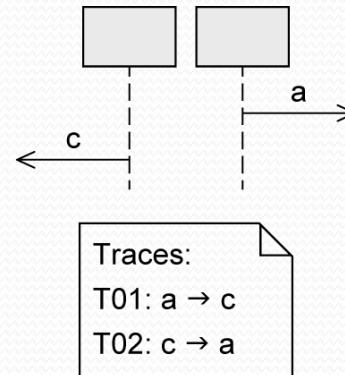
Exchanging Messages (2/2)

Order of messages:

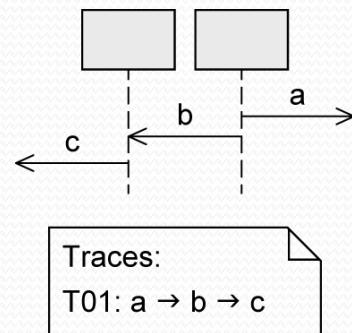
... on one lifeline



... on different lifelines



... on different lifelines which exchange messages

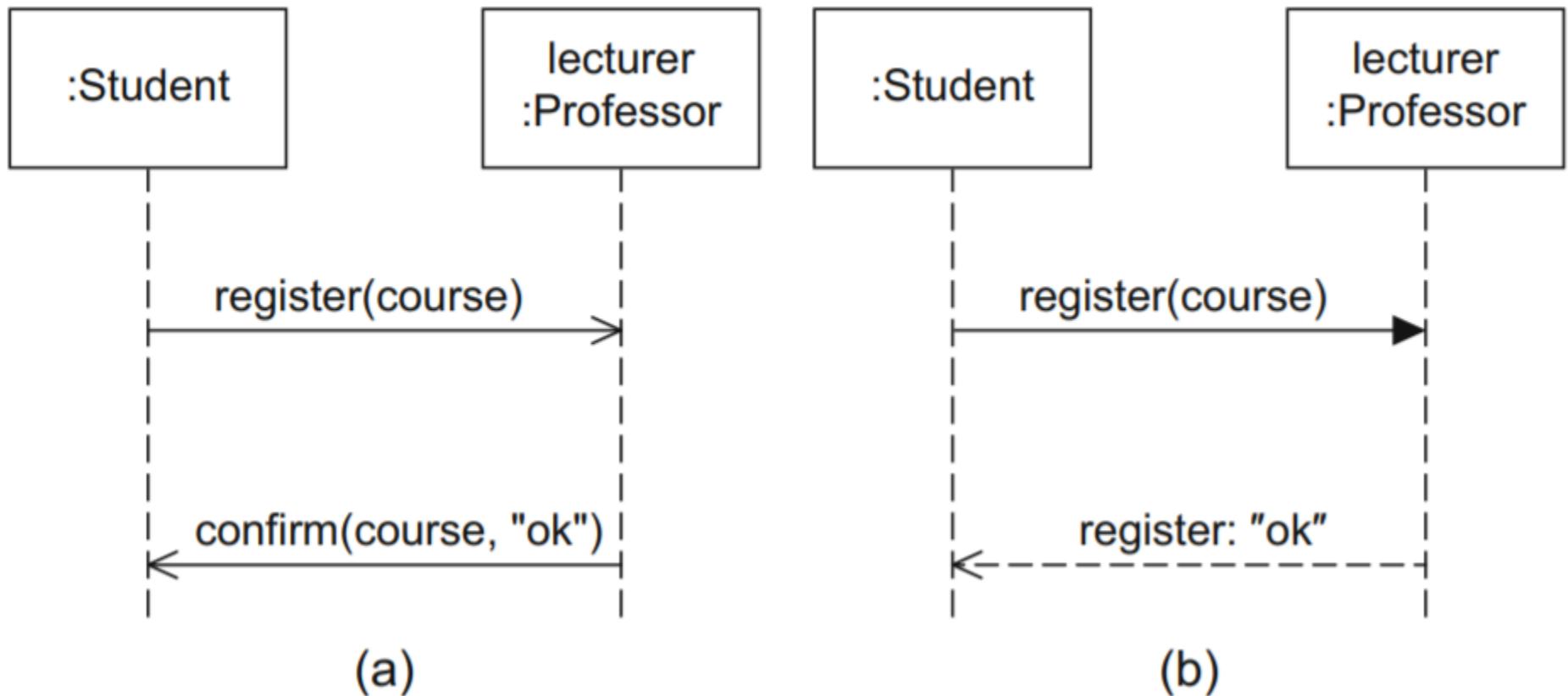


Messages (1/3)

- Synchronous message
 - Sender waits until it has received a response message before continuing
 - Syntax of message name: **msg (par1 , par2)**
 - **msg**: the name of the message
 - **par**: parameters separated by commas
- Asynchronous message
 - Sender continues without waiting for a response message
 - Syntax of message name: **msg (par1 , par2)**
- Response message
 - May be omitted if content and location are obvious
 - Syntax: **att=msg (par1 , par2) :val**
 - **att**: the return value can optionally be assigned to a variable
 - **msg**: the name of the message
 - **par**: parameters separated by commas
 - **val**: return value



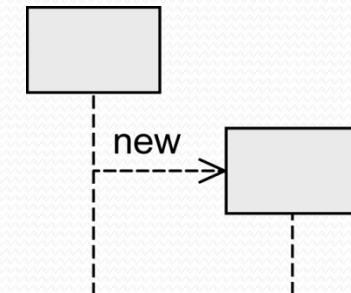
Examples of (a) asynchronous and (b) synchronous communication



Messages (2/3)

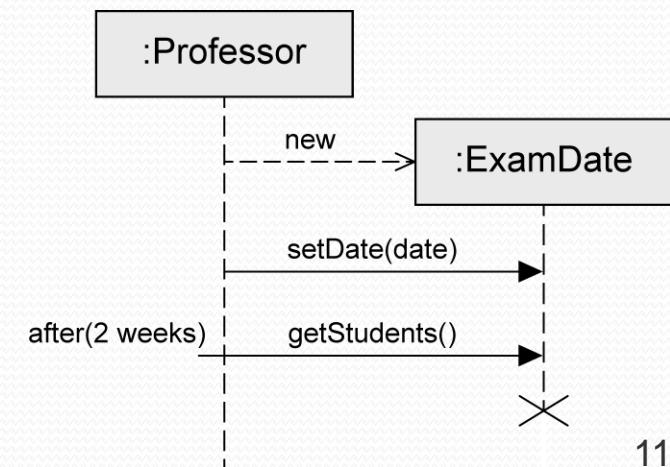
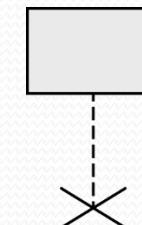
- Object creation

- Dashed arrow
- Arrowhead points to the head of the lifeline of the object to be created
- Keyword **new**



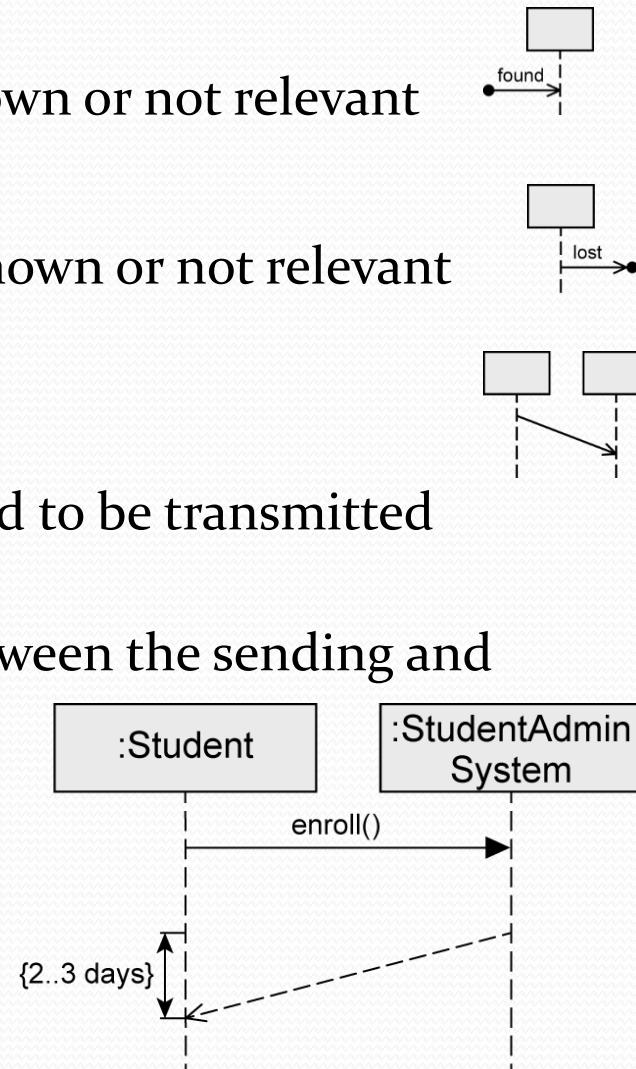
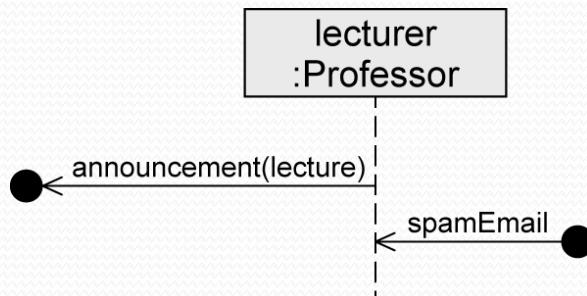
- Object destruction

- Object is deleted
- Large cross (×) at the end of the lifeline



Messages (3/3)

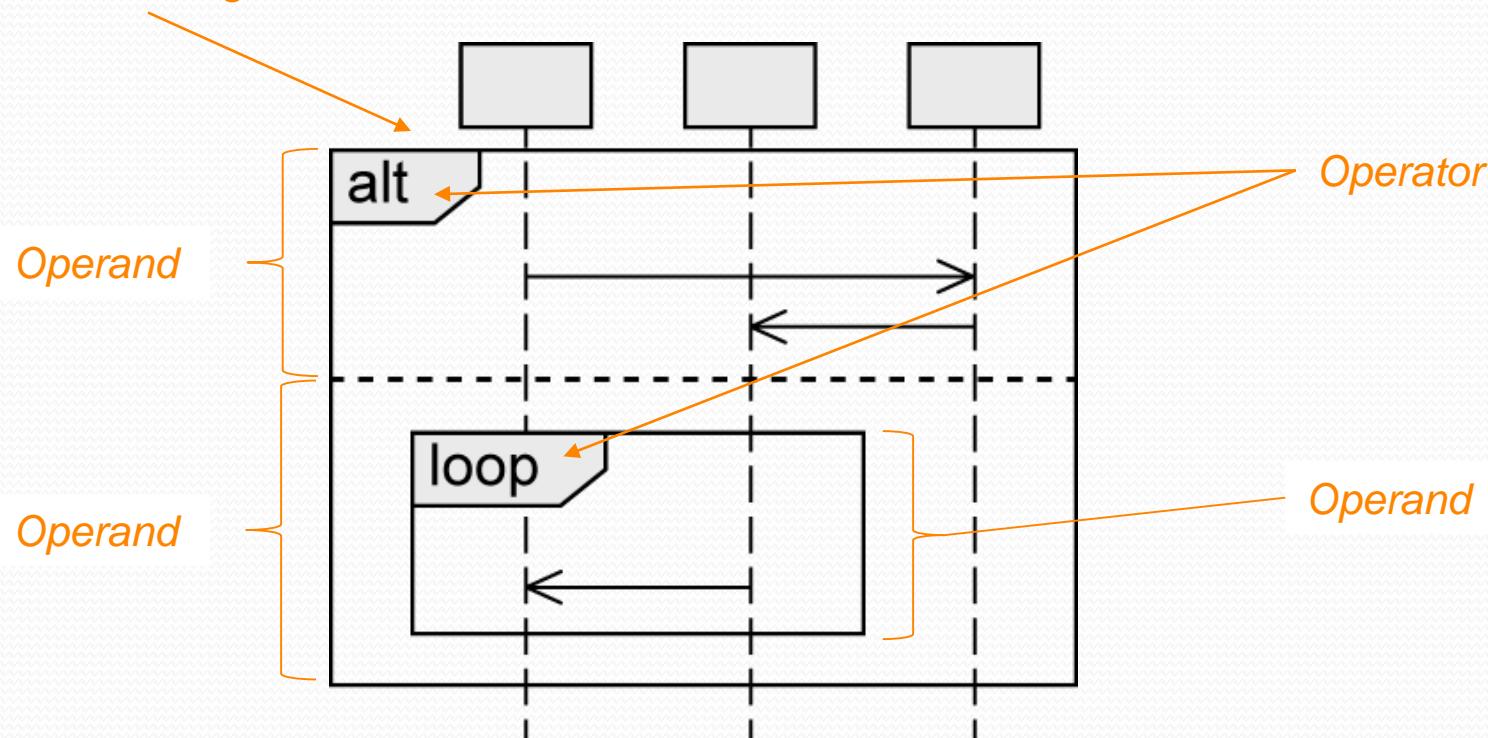
- Found message
 - Sender of a message is unknown or not relevant
- Lost message
 - Receiver of a message is unknown or not relevant
- Time-consuming message
 - "Message with duration"
 - Usually messages are assumed to be transmitted without any loss of time
 - Express that time elapses between the sending and the receipt of a message



Combined Fragments

- Model various control structures
- 12 predefined types of operators

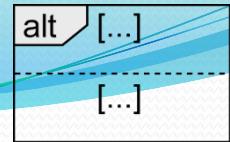
Combined Fragment



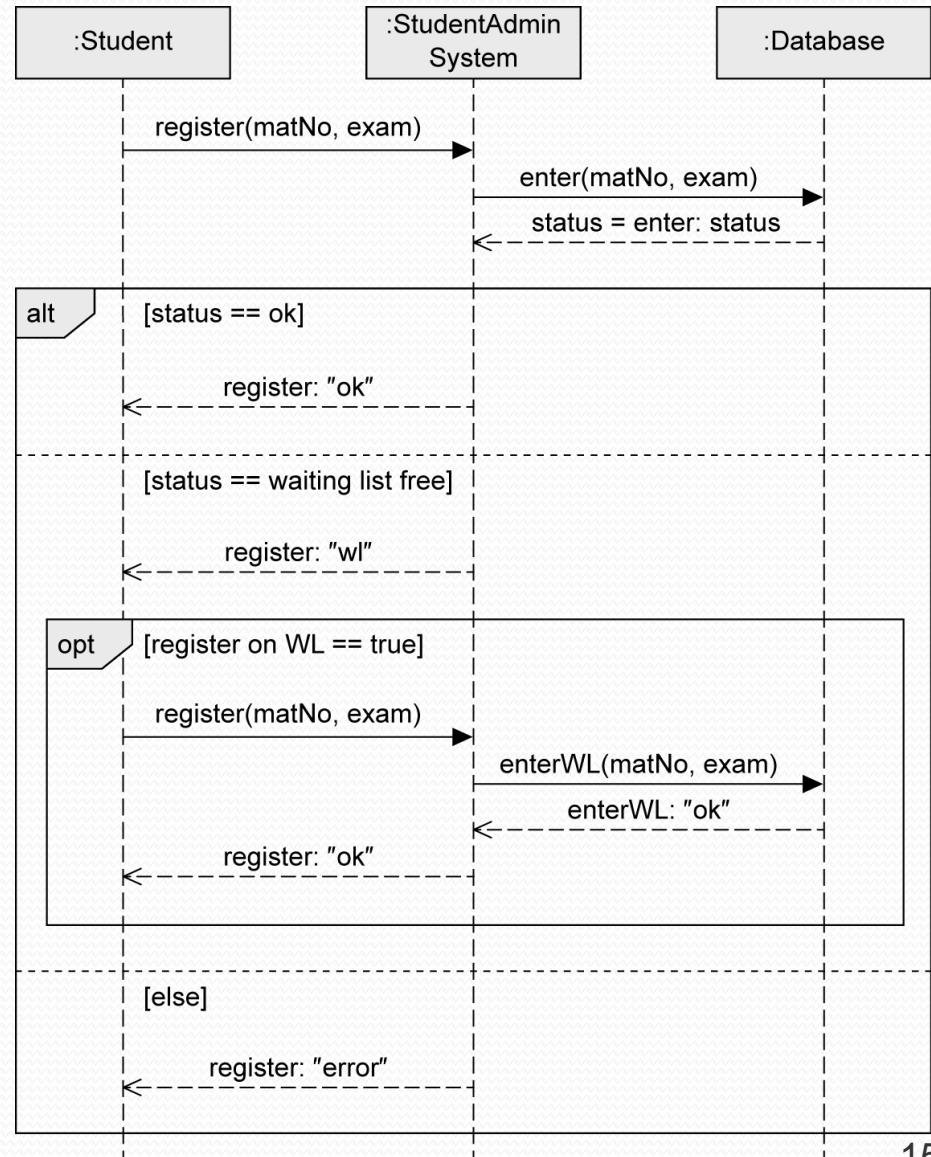
Types of Combined Fragments

	Operator	Purpose
Branches and loops	alt	Alternative interaction
	opt	Optional interaction
	loop	Repeated interaction
	break	Exception interaction
Concurrency and order	seq	Weak order
	strict	Strict order
	par	Concurrent interaction
	critical	Atomic interaction
Filters and assertions	ignore	Irrelevant interaction
	consider	Relevant interaction
	assert	Asserted interaction
	neg	Invalid interaction

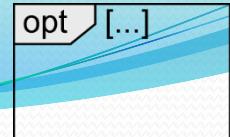
A1.alt Fragment



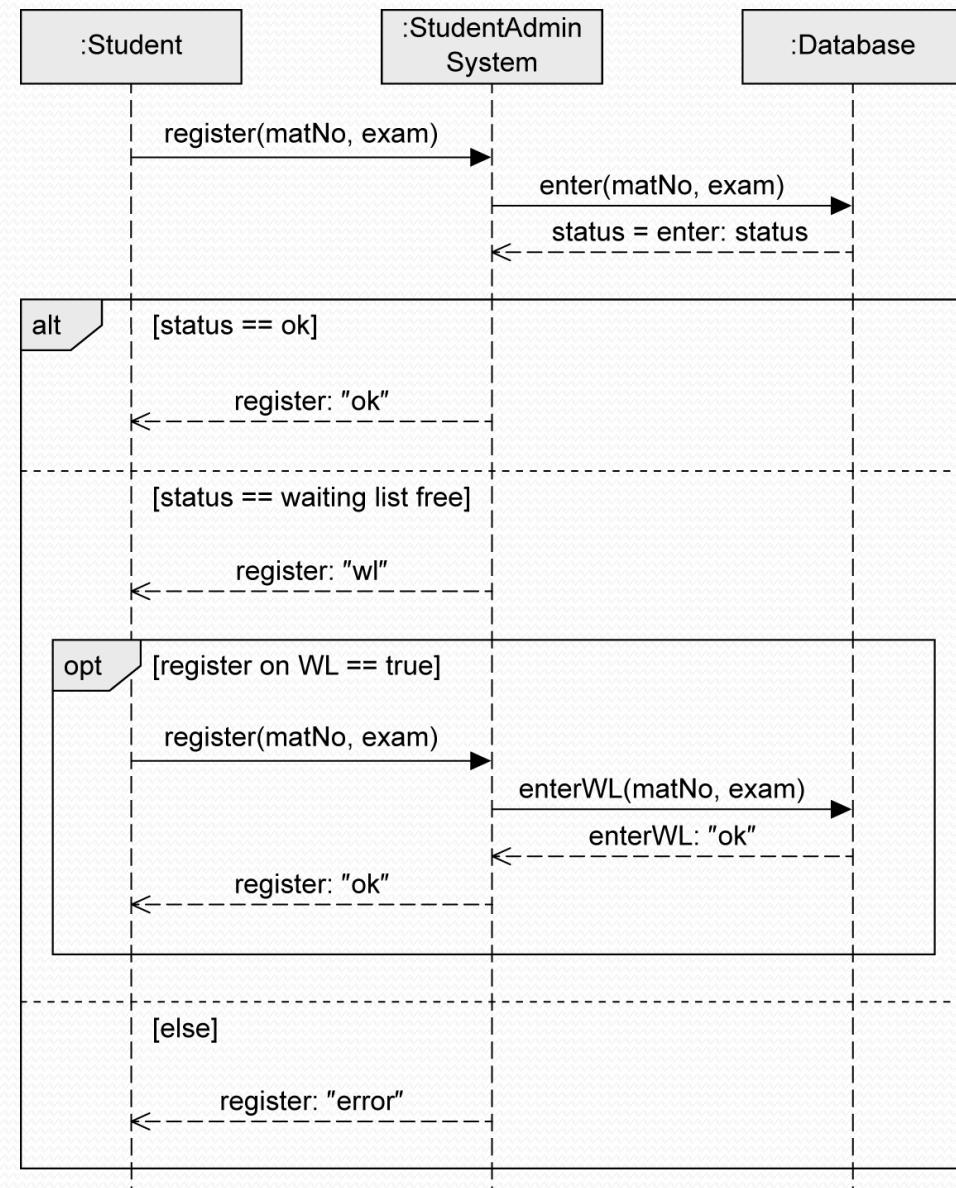
- To model alternative sequences
- Similar to switch statement in Java
- Guards are used to select the one path to be executed
- Guards
 - Modeled in square brackets
 - default: true
 - predefined: [else]
- Multiple operands
- Guards have to be disjoint to avoid indeterministic behavior



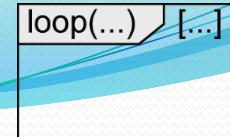
A2 .opt Fragment



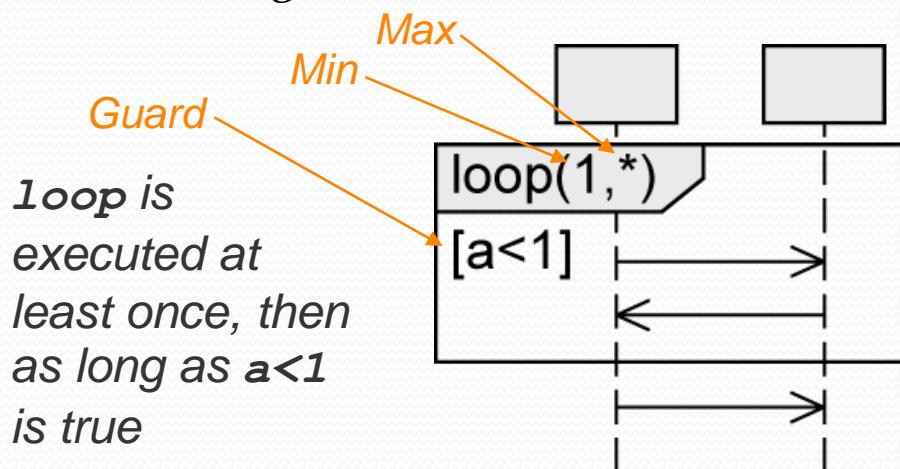
- To model an optional sequence
- Actual execution at runtime is dependent on the guard
- Exactly one operand
- Similar to **if** statement without **else** branch
- equivalent to **alt** fragment with two operands, one of which is empty



A3. loop Fragment



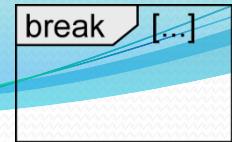
- To express that a sequence is to be executed repeatedly
- Exactly one operand
- Keyword `loop` followed by the minimal/maximal number of iterations (`min..max`) or (`min, max`)
 - default: `(*) ..` no upper limit
- Guard
 - Evaluated as soon as the minimum number of iterations has taken place
 - Checked for each iteration within the `(min, max)` limits
 - If the guard evaluates to false, the execution of the loop is terminated



Notation alternatives:

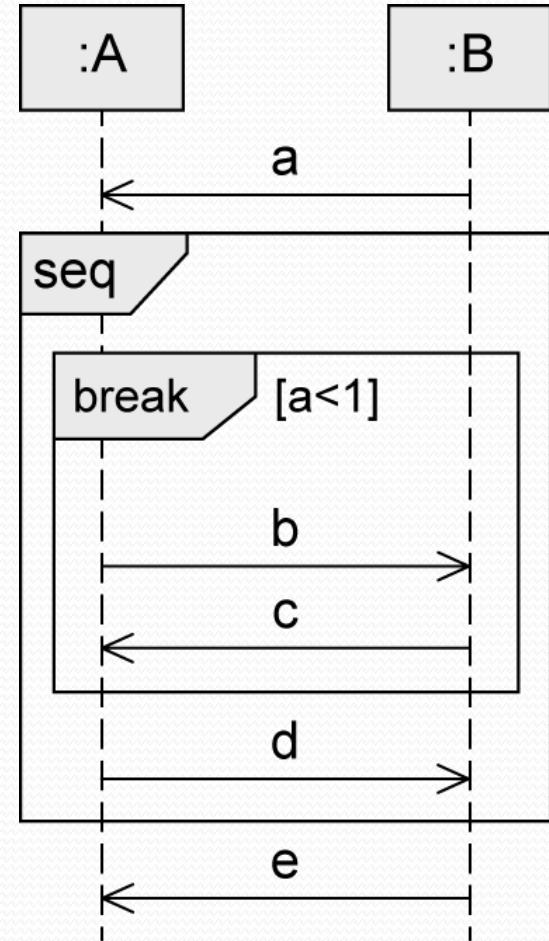
$\text{loop}(3, 8) = \text{loop}(3..8)$
 $\text{loop}(8, 8) = \text{loop}(8)$
 $\text{loop} = \text{loop}(*) = \text{loop}(0, *)$

A4 .break Fragment



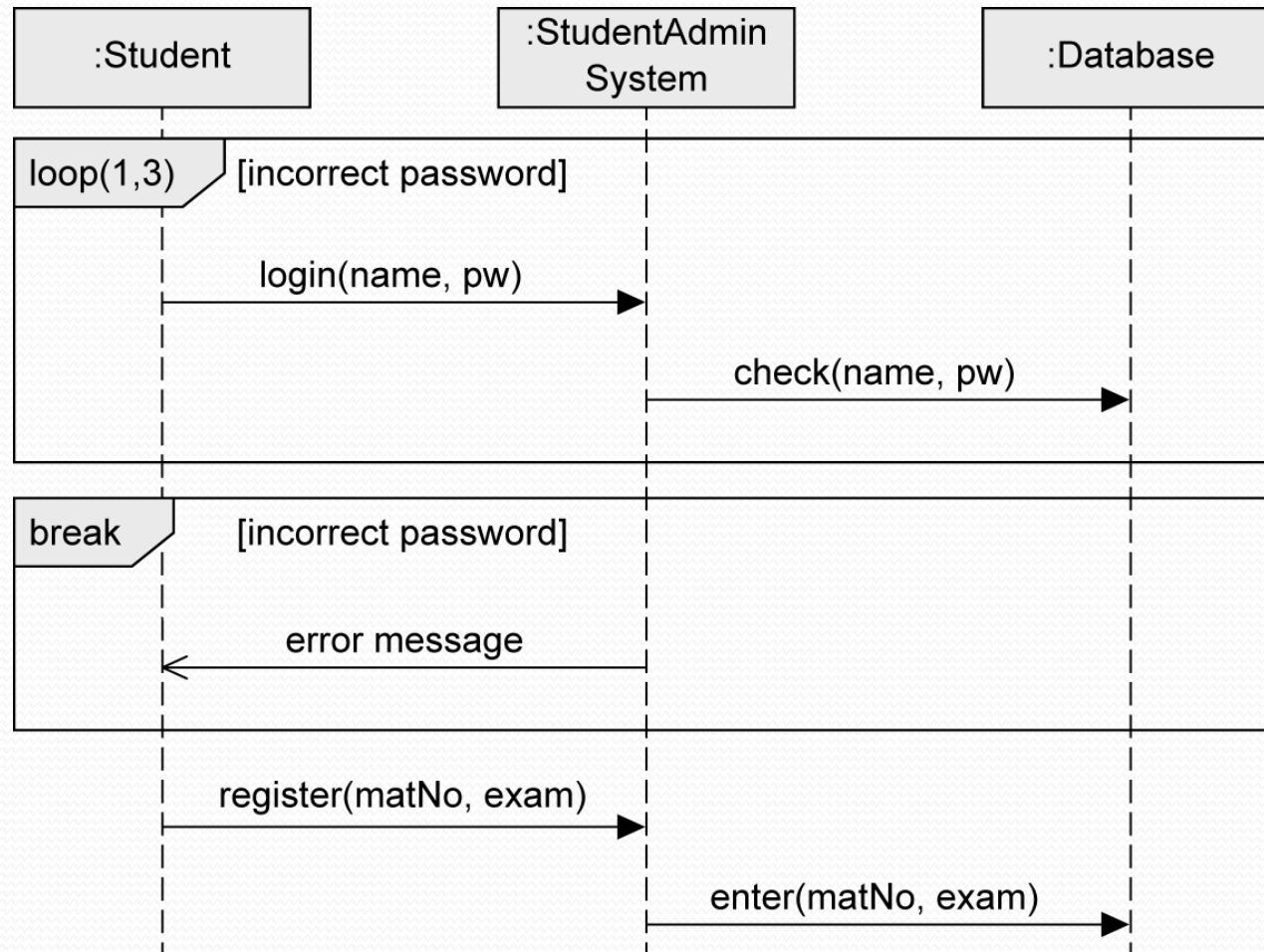
- Simple form of exception handling
- Exactly one operand with a guard
- If the guard is true:
 - Interactions within this operand are executed
 - Remaining operations of the surrounding fragment are omitted
 - Interaction continues in the next higher level fragment
 - Different behavior than `opt` fragment

Not executed if break is executed

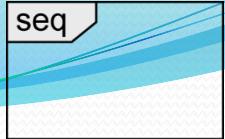


loop and break Fragment - Example

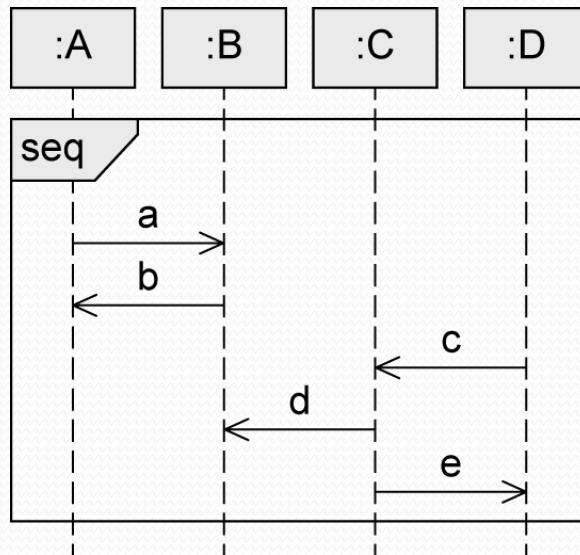
```
loop(...)[...]
```



B1 . seq Fragment



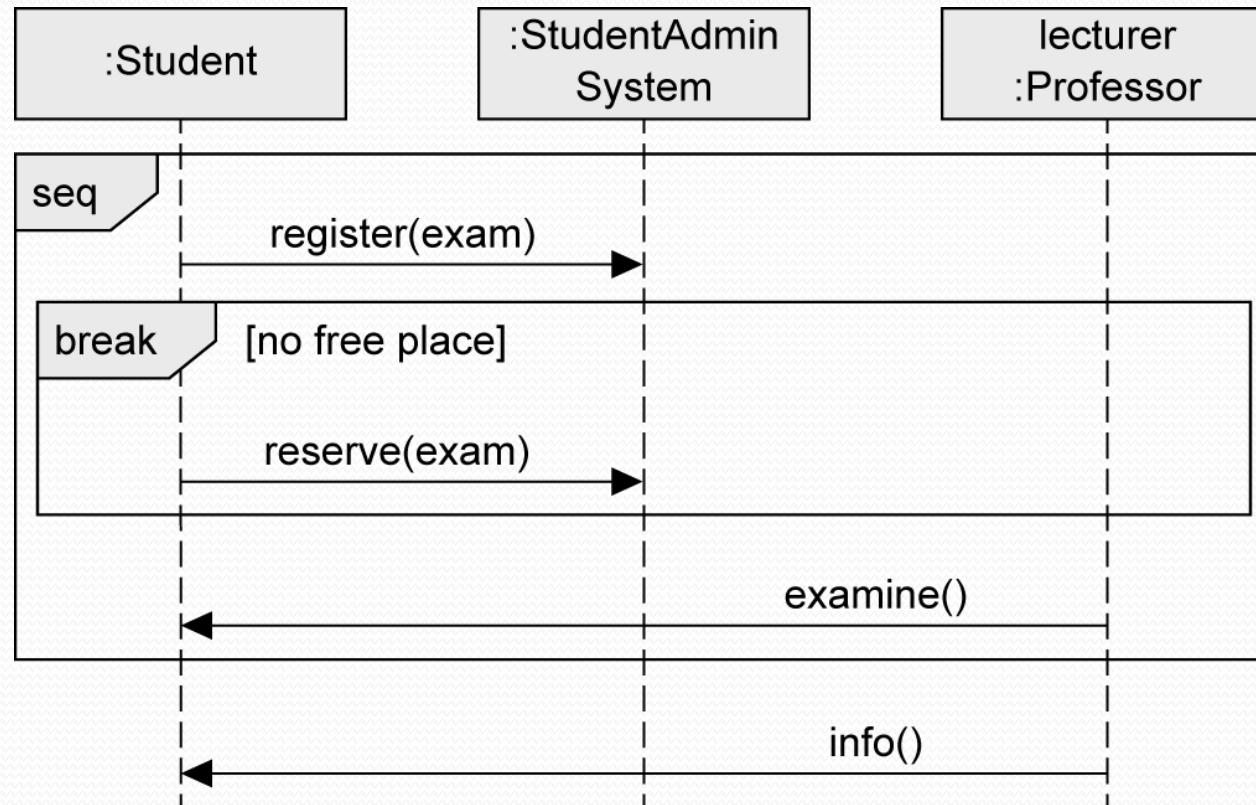
- Default order of events
- Weak sequencing:
 1. The ordering of events within each of the operands is maintained in the result.
 2. Events on different lifelines from different operands may come in any order.
 3. Events on the same lifeline from different operands are ordered such that an event of the first operand comes before that of the second operand.



Traces:

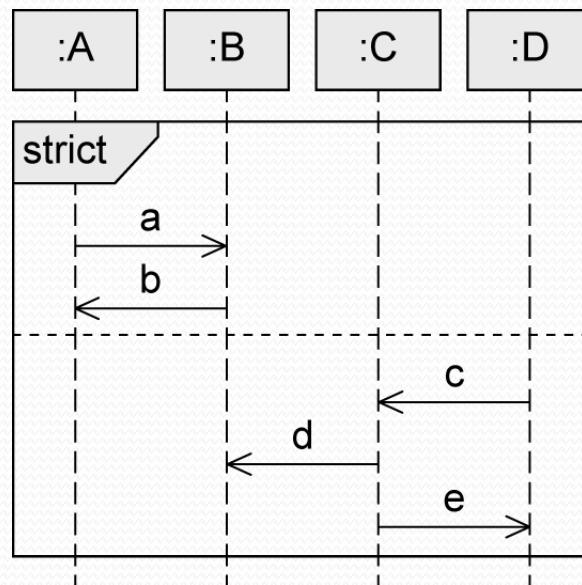
T01: a → b → c → d → e
T02: a → c → b → d → e
T03: c → a → b → d → e

seq Fragment – Example



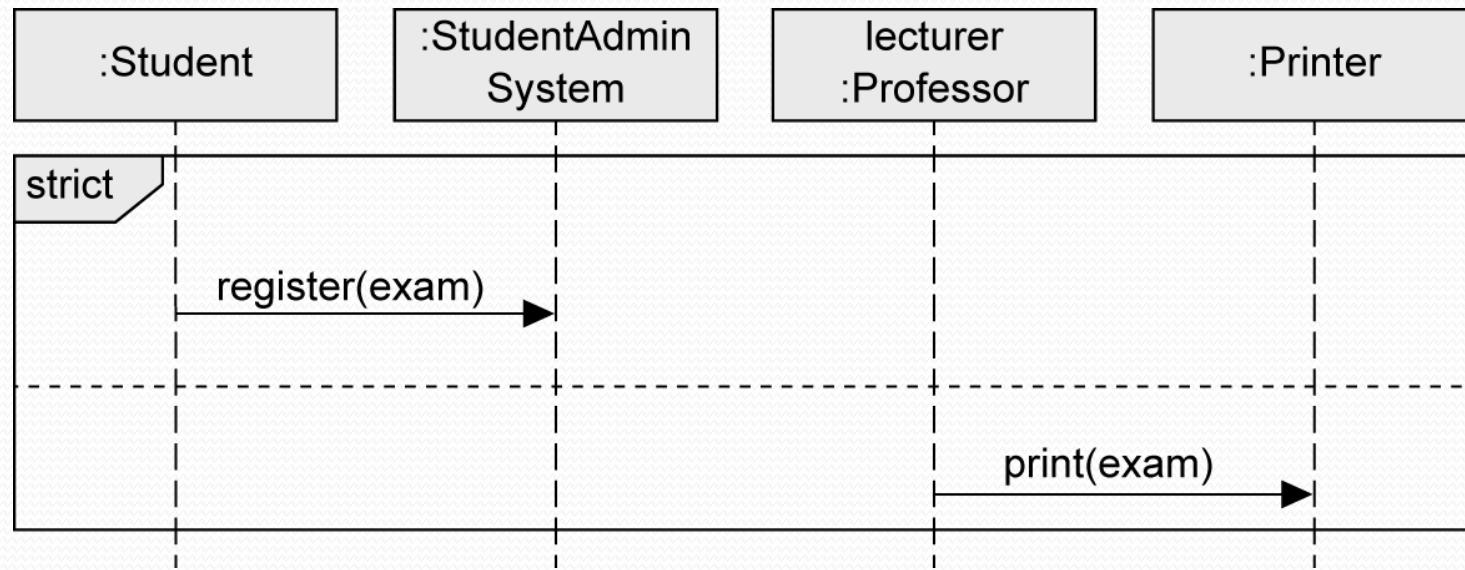
B2. strict Fragment

- Sequential interaction with order
- Order of event occurrences on different lifelines between different operands is **significant**
 - Messages in an operand that is higher up on the vertical axis are always exchanged before the messages in an operand that is lower down on the vertical axis

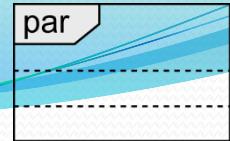


Traces:
T01: a → b → c → d → e

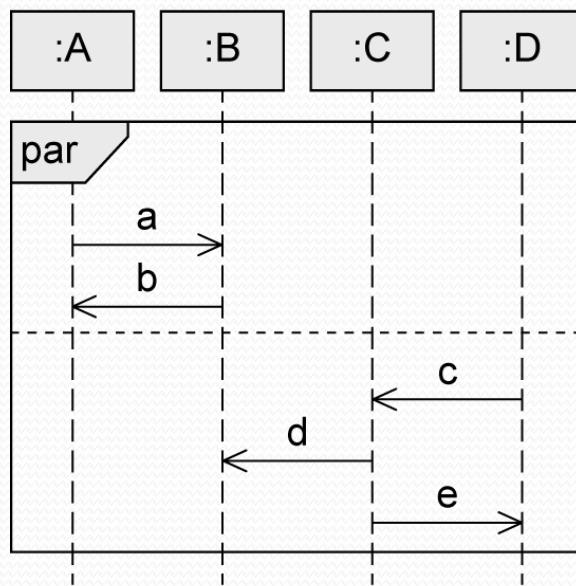
strict Fragment - Example



B3.par Fragment



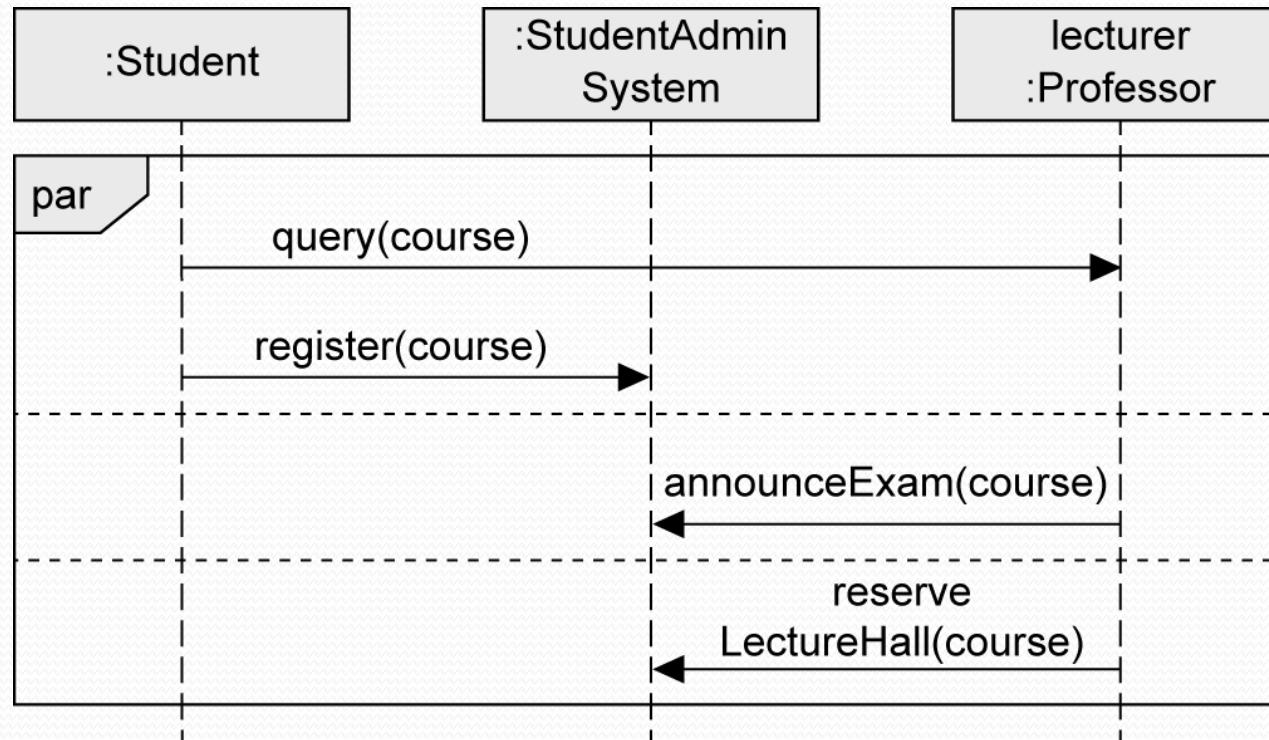
- To set aside chronological order between messages in different operands
- Execution paths of different operands can be interleaved
- Restrictions of each operand need to be respected
- Order of the different operands is irrelevant
- Concurrency, no true parallelism



Traces:

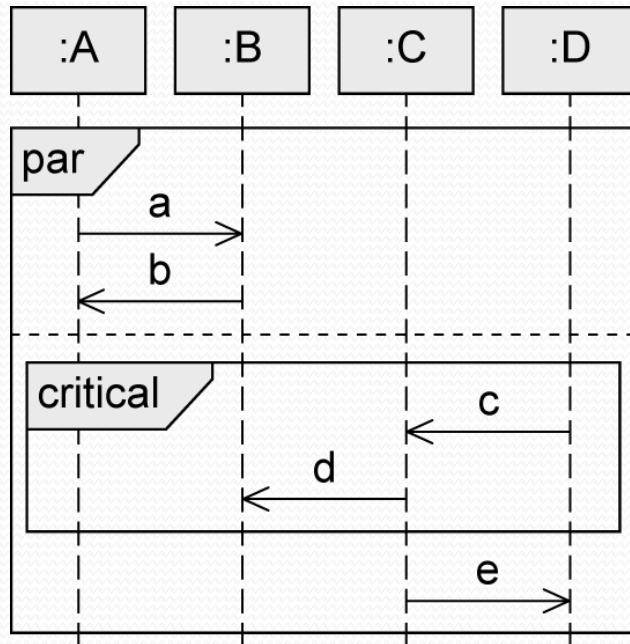
T01: a → b → c → d → e
T02: a → c → b → d → e
T03: a → c → d → b → e
T04: a → c → d → e → b
T05: c → a → b → d → e
T06: c → a → d → b → e
T07: c → a → d → e → b
T08: c → d → a → b → e
T09: c → d → a → e → b
T10: c → d → e → a → b

par Fragment - Example



B4. critical Fragment

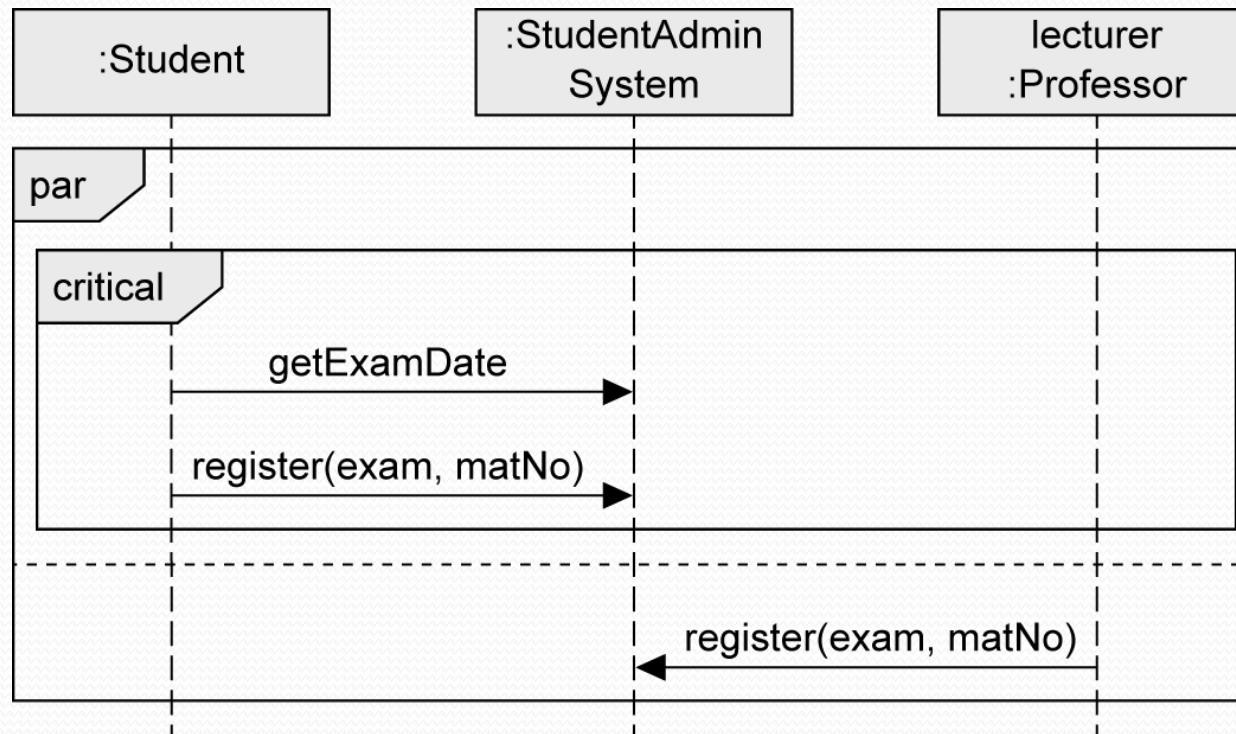
- Atomic area in the interaction (one operand)
- To make sure that certain parts of an interaction are not interrupted by unexpected events
- Order within **critical**: default order **seq**



Traces:

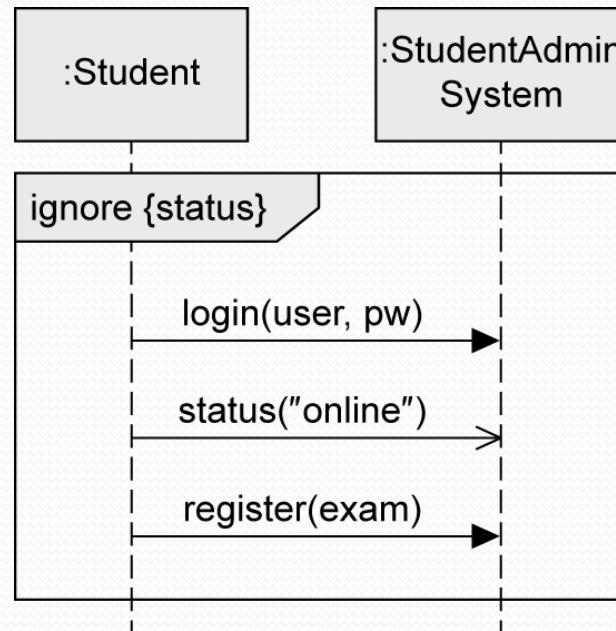
- T01: a → b → c → d → e
- T02: a → c → d → b → e
- T03: a → c → d → e → b
- T04: c → d → a → b → e
- T05: c → d → a → e → b
- T06: c → d → e → a → b

critical Fragment - Example



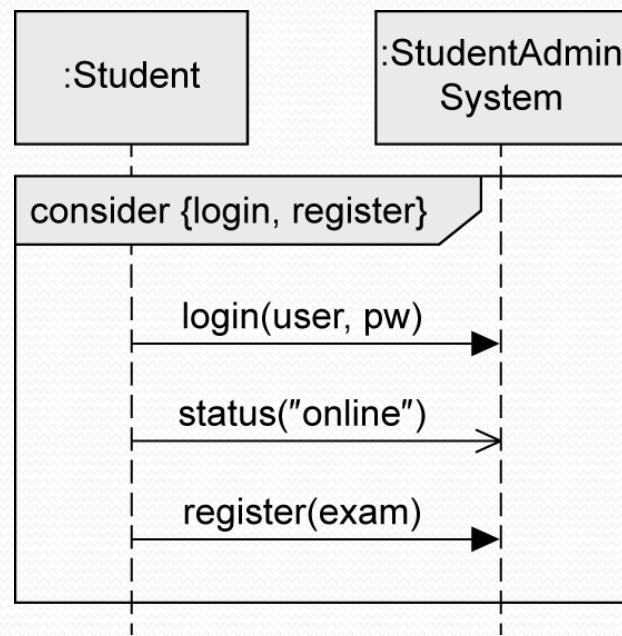
C1. ignore Fragment

- To indicate irrelevant messages
- Messages may occur at runtime but have no further significance
- Exactly one operand
- Irrelevant messages in curly brackets after the keyword **ignore**

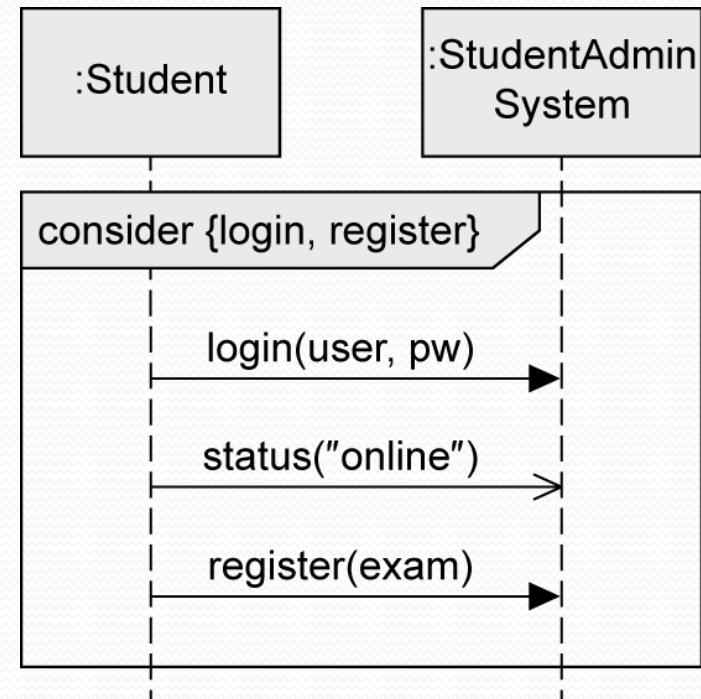
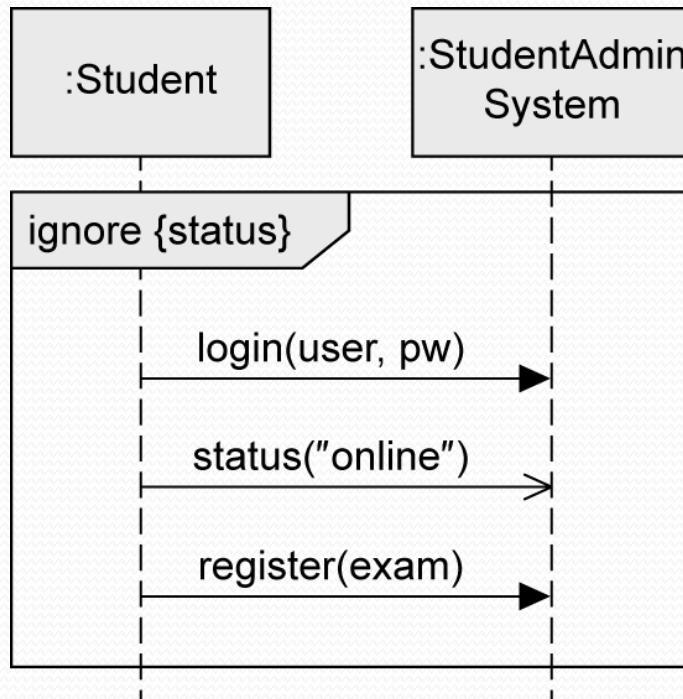


C2. consider Fragment

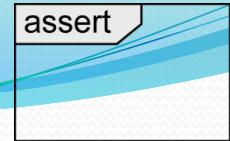
- To specify those messages that are of particular importance for the interaction under consideration
- Exactly one operand, dual to ignore fragment
- Considered messages in curly brackets after the keyword **consider**



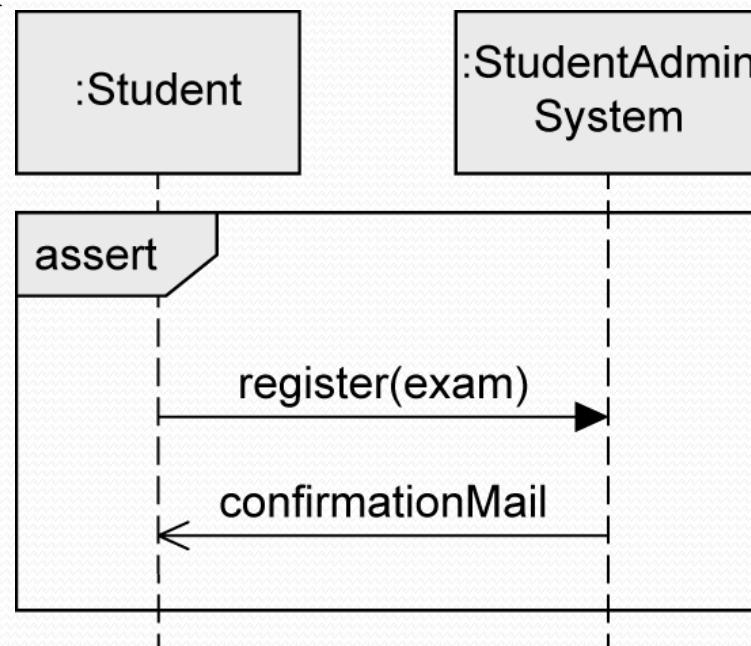
ignore vs. consider



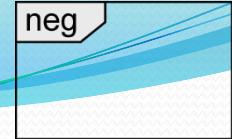
C3.assert Fragment



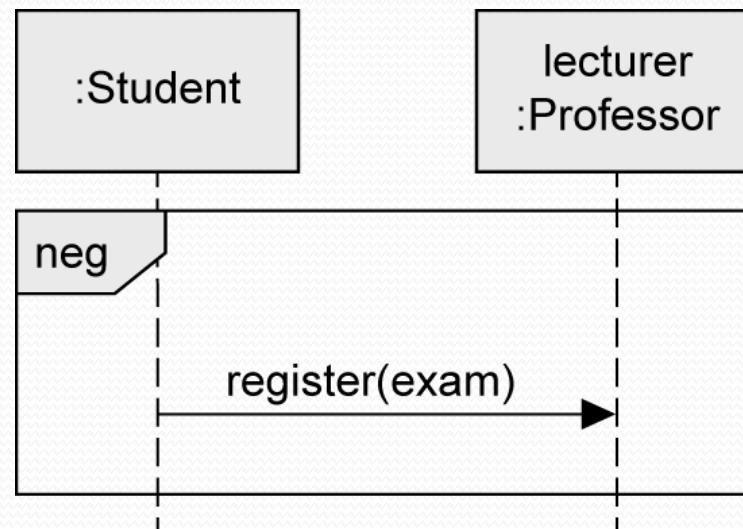
- To identify certain modeled traces as mandatory
- Deviations that occur in reality but that are not included in the diagram are not permitted
- Exactly one operand



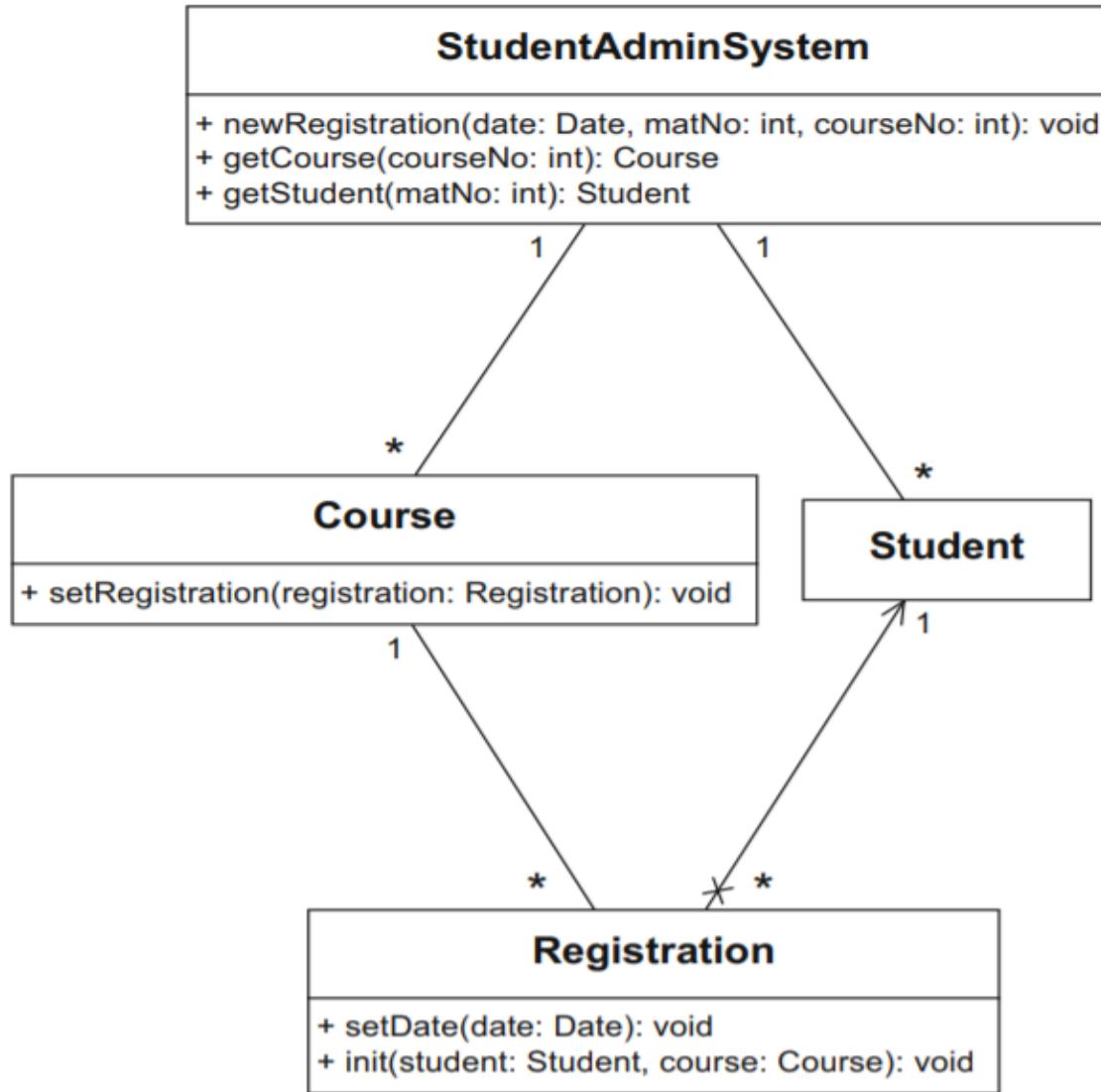
C4 .neg Fragment



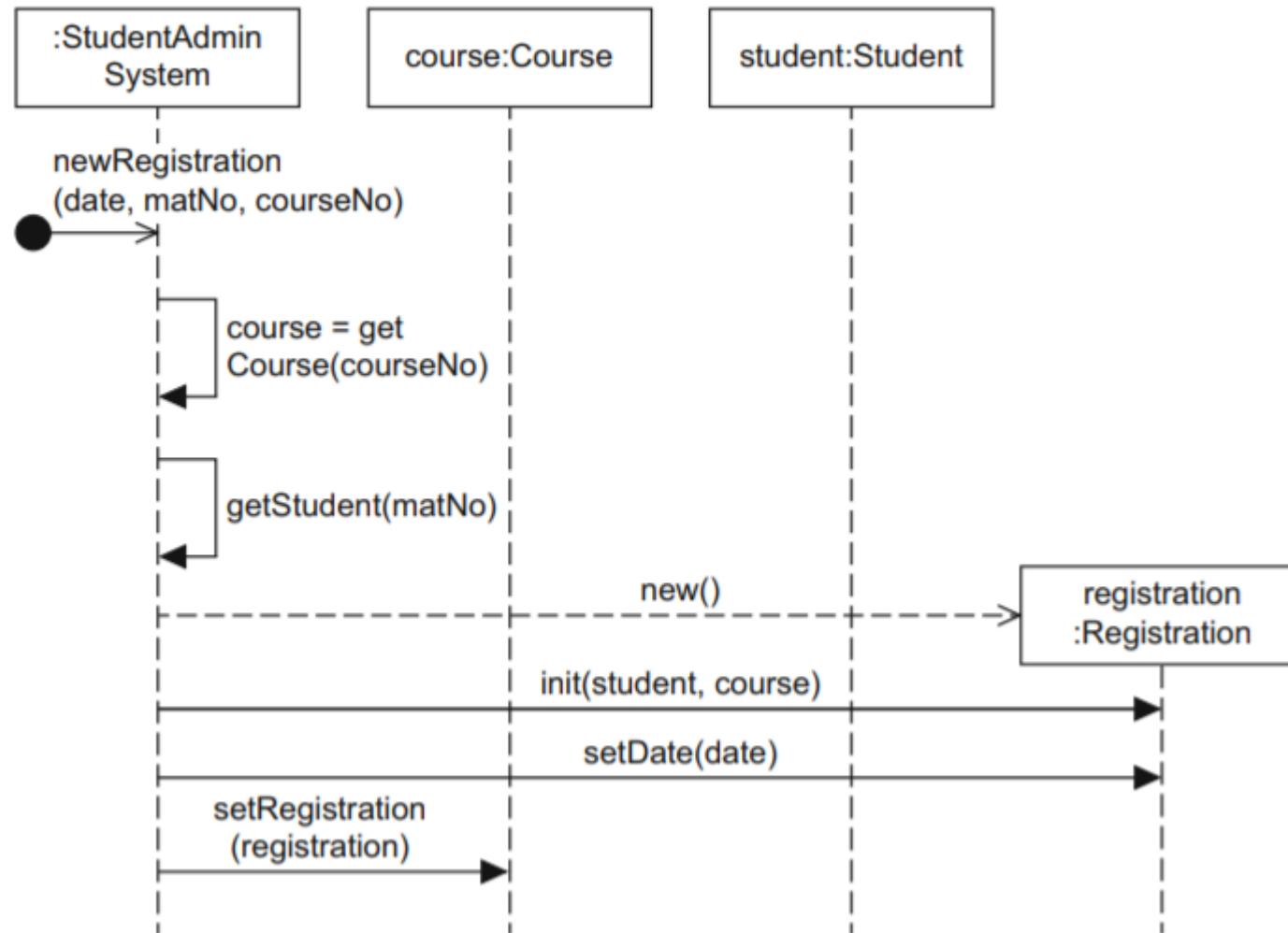
- To model invalid interactions
- Describing situations that must not occur
- Exactly one operand
- Purpose
 - Explicitly highlighting frequently occurring errors
 - Depicting relevant, incorrect sequences



The connection between a class diagram and a sequence diagram

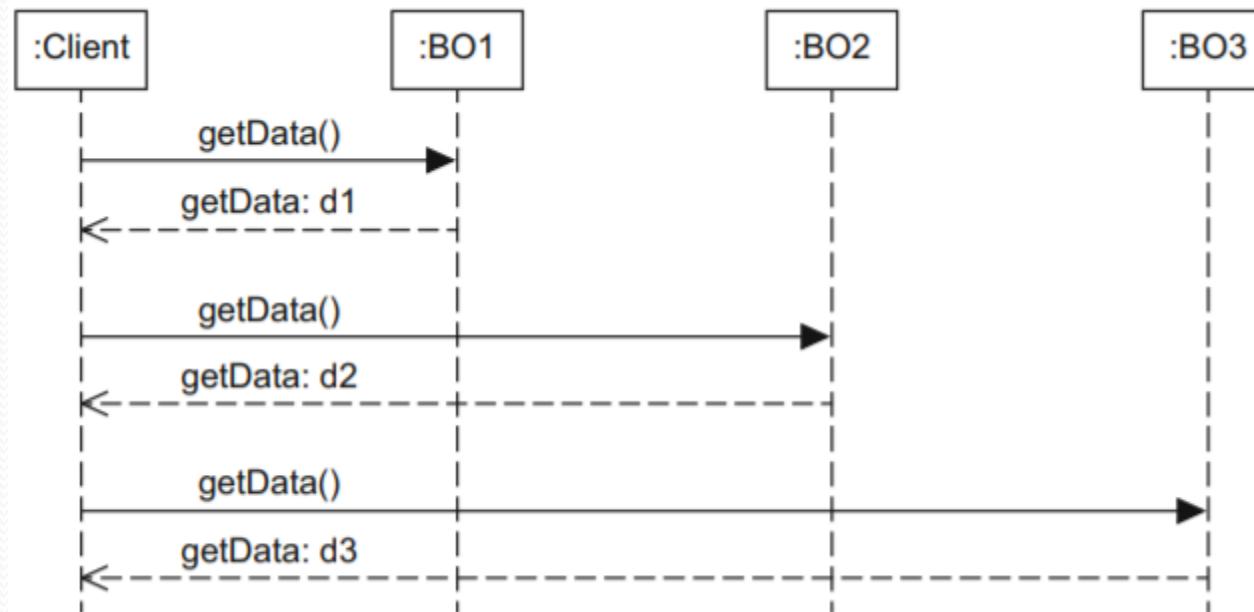


The communication that is required to create a new registration

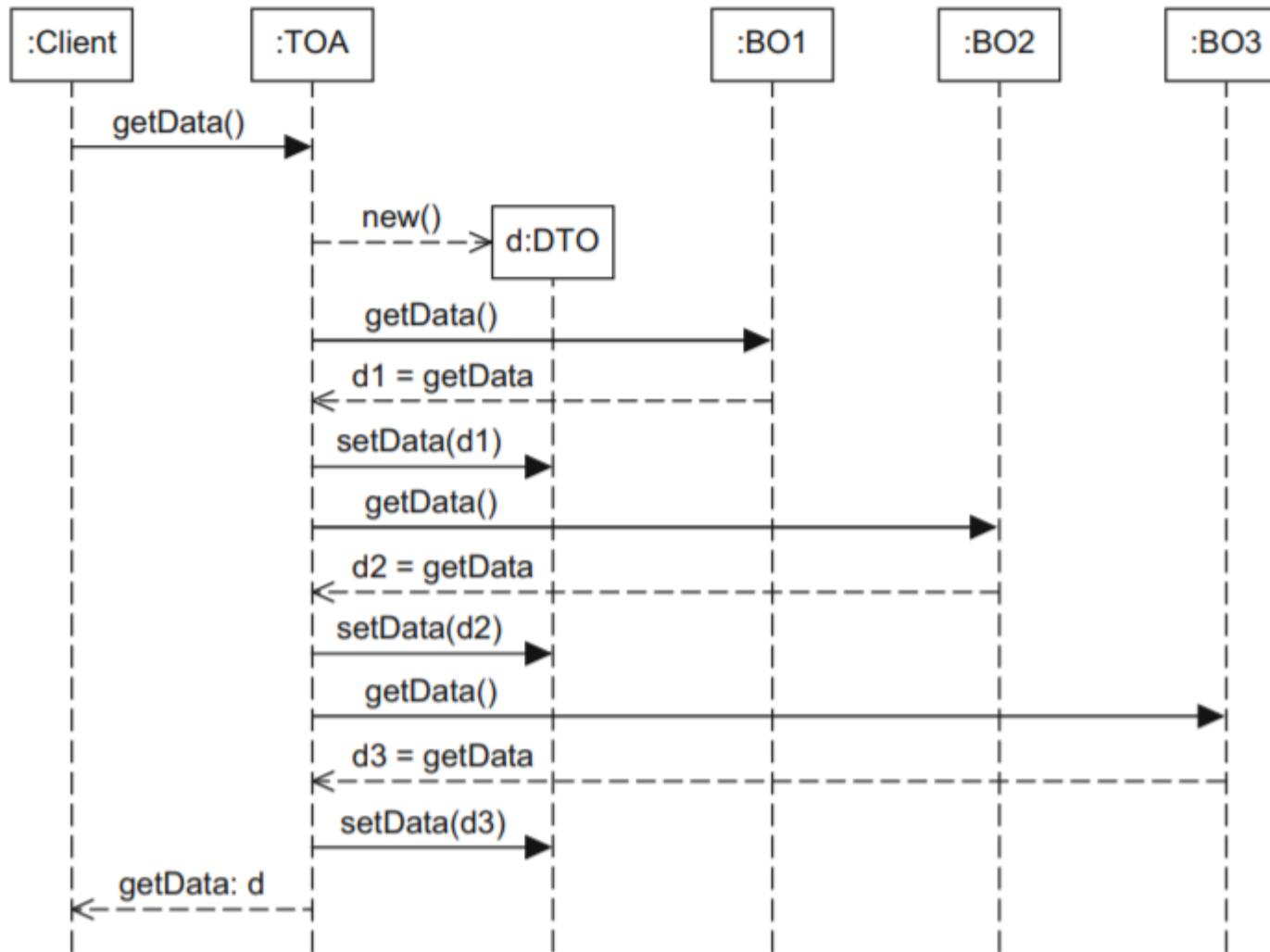


Modeling design patterns

- Sequence diagrams are often used to describe design patterns. Design patterns offer solutions for describing recurring problems
- A client in a distributed environment requires information from 3 business objects

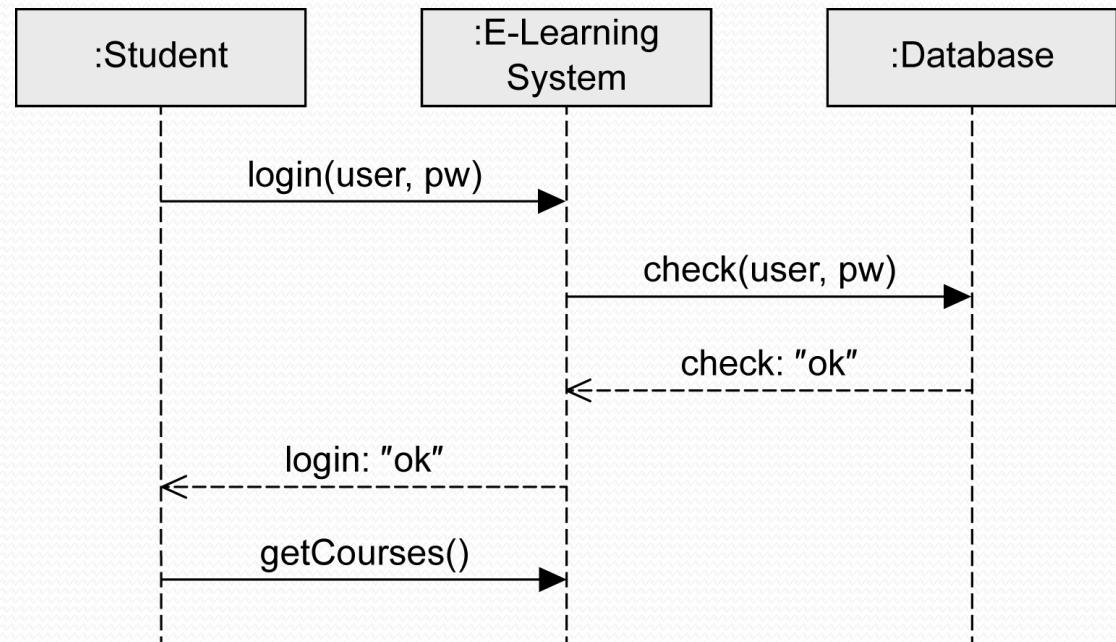


Transfer Object Assembler pattern



Four Types of Interaction Diagrams (1/4)

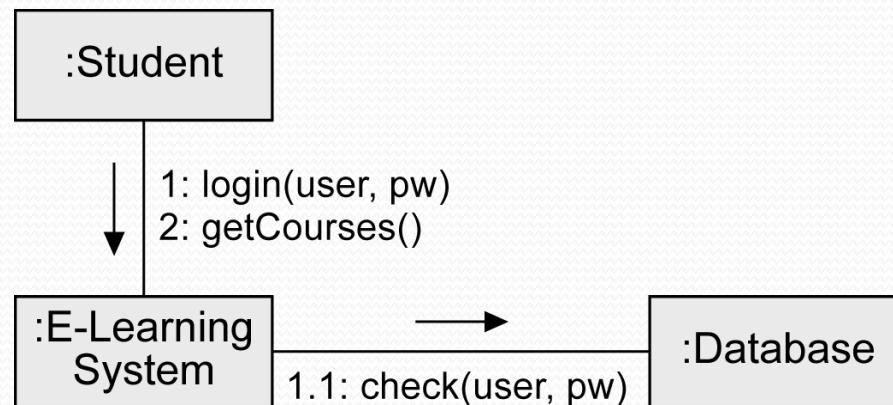
- Based on the same concepts
- Generally equivalent for simple interactions, but different focus
- **Sequence diagram**
 - Vertical axis:
chronological order
 - Horizontal axis:
interaction partners



Four Types of Interaction Diagrams (2/4)

- **Communication diagram**

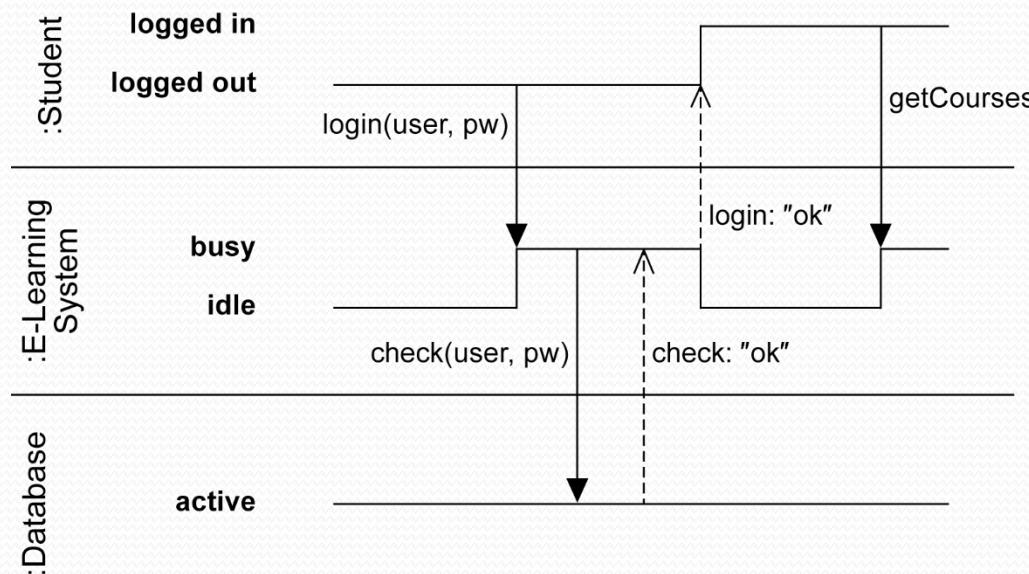
- Models the relationships between communication partners
- Focus: Who communicates with whom
- Time is not a separate dimension
- Message order via decimal classification



Four Types of Interaction Diagrams (3/4)

- **Timing diagram**

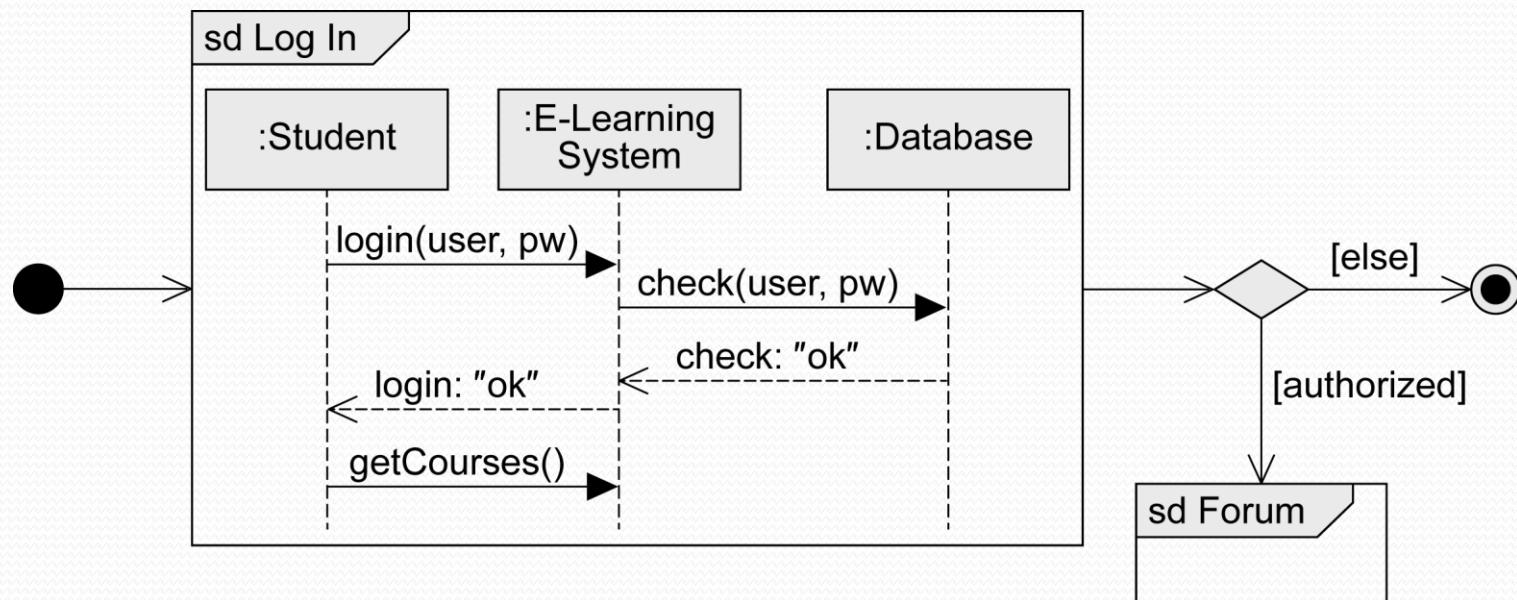
- Shows state changes of the interaction partners that result from the occurrence of events
- Vertical axis: interaction partners
- Horizontal axis: chronological order



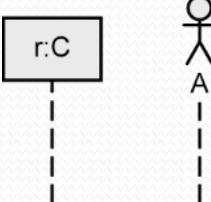
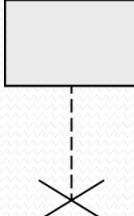
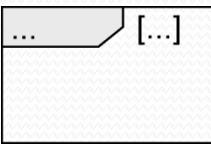
Four Types of Interaction Diagrams (4/4)

- **Interaction overview diagram**

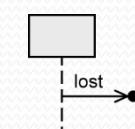
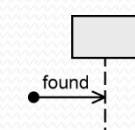
- Visualizes order of different interactions
- Allows to place various interaction diagrams in a logical order
- Basic notation concepts of activity diagram



Notation Elements (1/2)

Name	Notation	Description
Lifeline		Interaction partners involved in the communication
Destruction event		Time at which an interaction partner ceases to exist
Combined fragment		Control constructs

Notation Elements (2/2)

Name	Notation	Description
Synchronous message		Sender waits for a response message
Response message		Response to a synchronous message
Asynchronous communication		Sender continues its own work after sending the asynchronous message
Lost message		Message to an unknown receiver
Found message		Message from an unknown sender

BUSINESS PROCESS MODELING - BPMN

Lecture 9

Prof. Bologa Ana Ramona

AGENDA

- Business Process Modeling
- BPMN Language
- Elements of BPMN
- Flow objects
- Connecting objects
- Partitioning objects
- Data
- Artefacts
- Diagram types



BUSINESS PROCESSES

- A **business process** can be seen as *a set of interrelated activities, executed by different organizational units, that work together to accomplish one of the company objectives.*
- Activities that are included into a business process can be executed manually by a human factor or by a computer system
- Information technology, in general, and computer systems, in particular, influence economic process management.

BUSINESS PROCESS CHARACTERISTICS

- A BP is a collection of activities performed in a **coordinated** manner that are logically correlated.
- They work in operational and technical environments;
- They produce **results** in line with organization objectives;
- They are performed by an organization;
- **They can interact with business processes performed by other organizations.**

BUSINESS PROCESS MANAGEMENT

- **Business process management** includes concepts, methods, and techniques needed to support the design, administration, configuration, implementation, and analysis of business processes. There are different levels of process modeling:
 - i. **Business process mapping**— simple workflows of activities;
 - ii. **Business process description** – workflows that are extended with additional information, but not enough to define the process in detail;
 - iii. **Business process models**— activity flows that are extended with enough information to analyze, simulate and/or execute the process.
- BPMN supports all three levels
- BPMN = Business Process Model and Notation

BPMN - BUSINESS PROCESS MODELING NOTATIONS

- BPMN offers support for BP modeling by providing **intuitive notations** that are able to capture complex business rules and to establish a connection between design and implementation stages.
- It is a graphical representation based on **activity flows** for defining business processes.
- It represents a **consensus** between manufacturers of modeling tools that used proprietary notations.
- BPMN **offer a mechanism for executable business process generation (BPEL)** from this graphical representation.
- The business process modeled in BPMN can then be passed to a **BPEL engine for execution**, instead of being interpreted by a person or translated in other programming languages

HISTORY OF BPMN

- **BPMI (Business Process Management Institute)** – it is now part of OMG (Object Management Group); it developed BPML (a **XML-based language** for business process execution)
- BPML was later replaced by **BPEL**
- BPMN adoption:
 - May **2004** - BPMN 1.0
 - February **2006** BPMN 1.0 adopted as OMG standard
 - March **2010** BPMN 2.0 – an improved version of standard
 - The latest **stable** version
 - It brings significant improvements by extending the language with new symbols
 - It adds two new diagram types (Choreography and Conversation diagrams)
 - The new version also provides the first formal representation of BPML as a **metamodel**

BUSINESS PROCESS MODELING TOOLS



Visual Paradigm



ProcessMaker



bizagi

<https://www.youtube.com/watch?v=NdWvYqYoCB0>

BPMN– BASIC ELEMENTS(1)

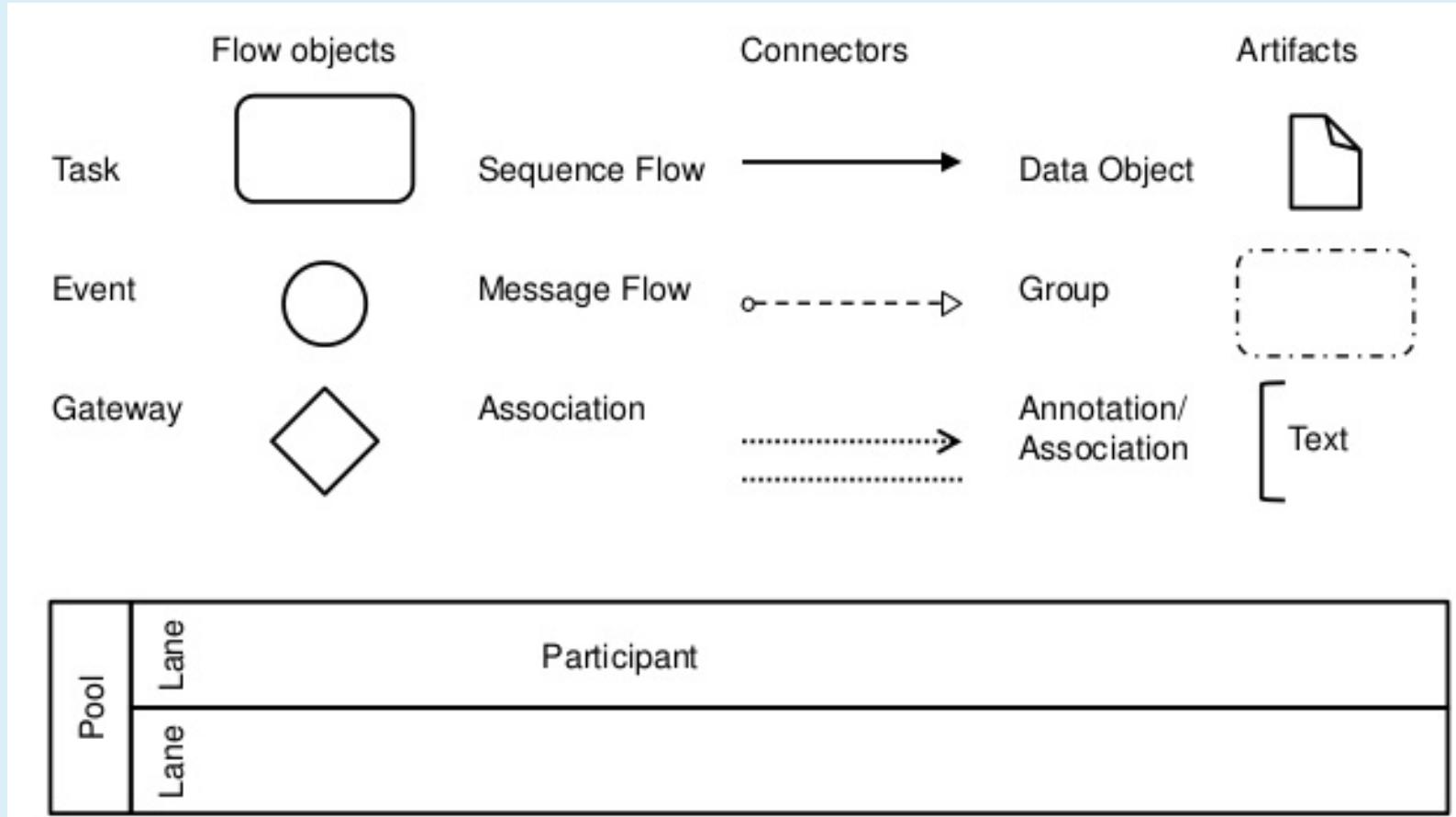
1. **Flow objects** represent the main elements of process diagrams. They fall into one of the following categories:
 - a) **Event**,
 - b) **Activity**,
 - c) **Gateway**.
2. **Connecting objects** have the role of connecting object flows with each other or with other object types. There are three types of connecting objects:
 - a) **Sequence flow**,
 - b) **Message flow** and
 - c) **Association**.
3. **Swimlanes** – They set subgraphs in the process flow in order to logically separate some parts of it, depending on the **entities involved** in carrying out the process. There are two types:
 - a) **Pool** and
 - b) **Lane**.

BPMN– BASIC ELEMENTS(2)

4. **Data** are necessary to show data needed in activities or resulted from activities. They fall into four categories:
 - i. **Data object**,
 - ii. **Data input**,
 - iii. **Data output** and
 - iv. **Data store**.
5. **Artifacts** - they provide additional information about how documents, data, and other objects are used and updated within a Process. There are two types of standard artifacts:
 - i. **Group** and
 - ii. **Annotation**,

but both language and modeling tools provide an option of adding any other custom user artifacts necessary to understand the model.

BPMN – BASIC ELEMENTS(3)



FLOW OBJECTS

- Flow objects are the main describing elements within BPMN, and consist of three core elements:
 - **Activity** - An activity is a generic term for work that a company performs. It can be **atomic (task)** or **compound (sub-process)**. It is represented with a rounded-corner rectangle.
 - **Event**: it denotes something that **happens** (compared with an activity, which is something that is **done**). Icons within the circle denote the type of event. They impact the model flow and usually have a cause (trigger) and an effect (result). There are three event types, considering the moment they impact the process flow:



Start



Intermediate



End

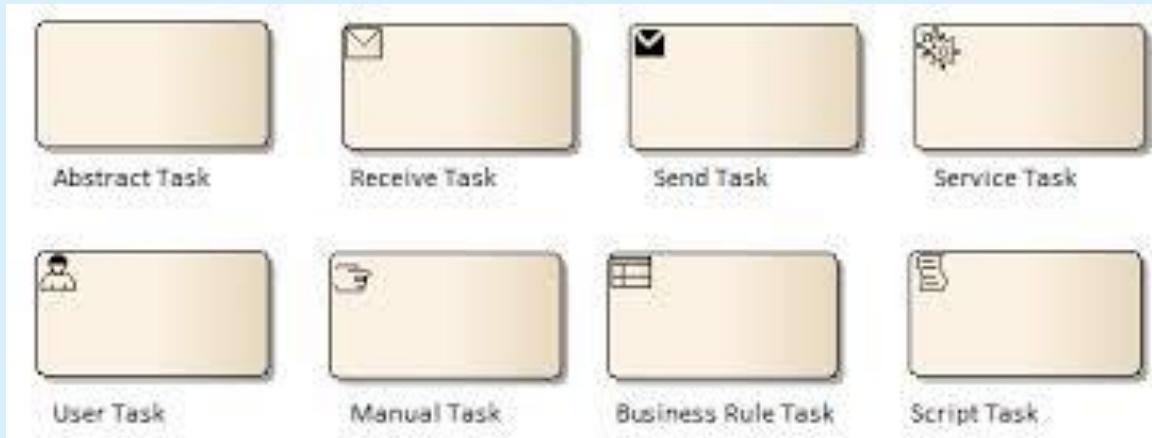
Task

Sub-Process
+

- **Gateways**: they are modeling elements that determine forking and merging of paths, depending on the conditions expressed. They are considered **decision** elements and are represented with a diamond shape.



A **task** represents a single unit of work that is not or cannot be broken down to a further level of business process detail.



Abstract task – task without any specialization

Receive task – task that waits for a message to arrive from an external participant. After the message is received, that task is completed.

Send task – task for sending a message to an external participant. After the message is send, that task is completed.

Service task – task that uses some sort of service, like a web service.

User task – task executed by a human with the assistance of a software application

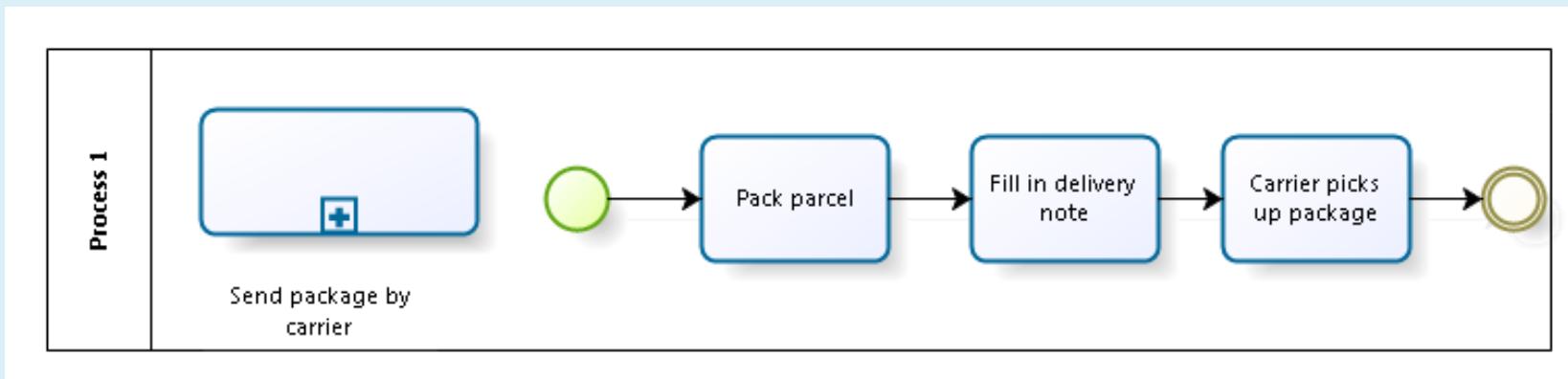
Manual task – task executed by a human without the assistance of any process execution engine or application.

Business rule task – Provides input to a business rule engine. Receives output from a business rule engine.

Script task – It is executed by a business process execution engine. When the script is completed, the task is finished.

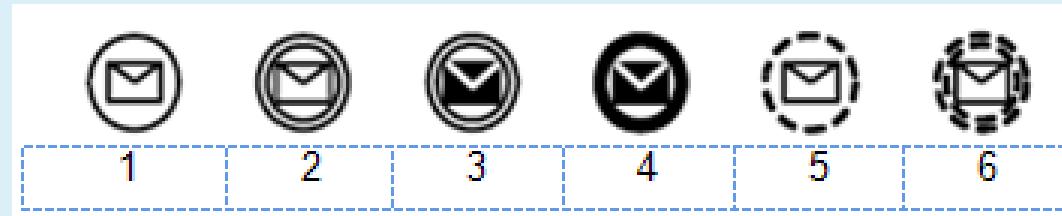
Sub-processes

- They are compound activities included in a process.
- They can be hierarchically nested to any level of detail that is necessary to fully describe a process.
- A sub-process can be collapsed or expanded to show or hide its details
- Each expended description of a sub-process must include start and end events which do not specify a particular behavior.



EVENT CATEGORIES

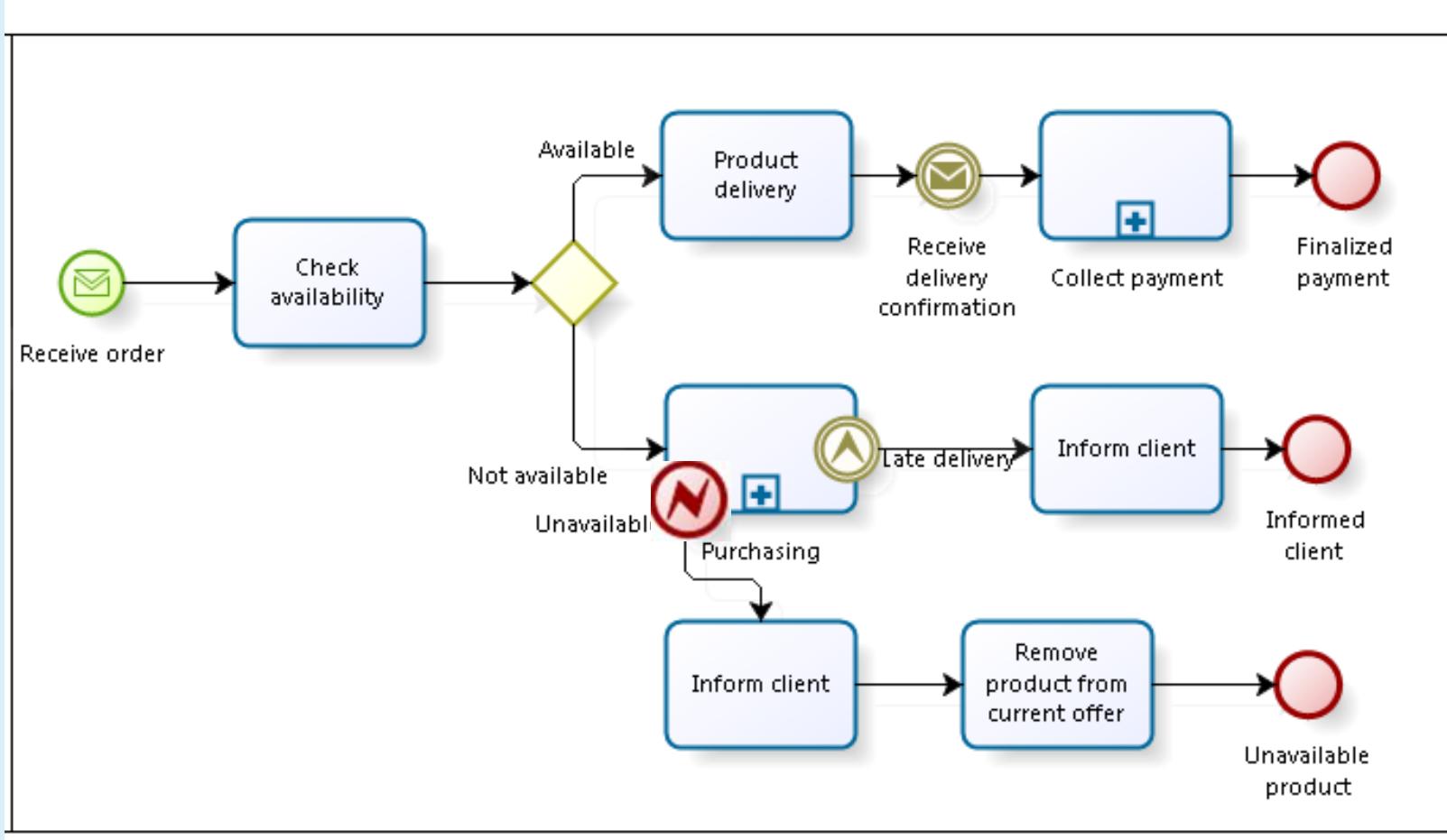
1. Start event that receives a message.
2. Intermediate event that receives a message.
3. Intermediate event that sends a message.
4. End event that sends a message.
5. Start event that receives a message without interrupting another activity.
6. Intermediate event that receives a message without interrupting another activity.



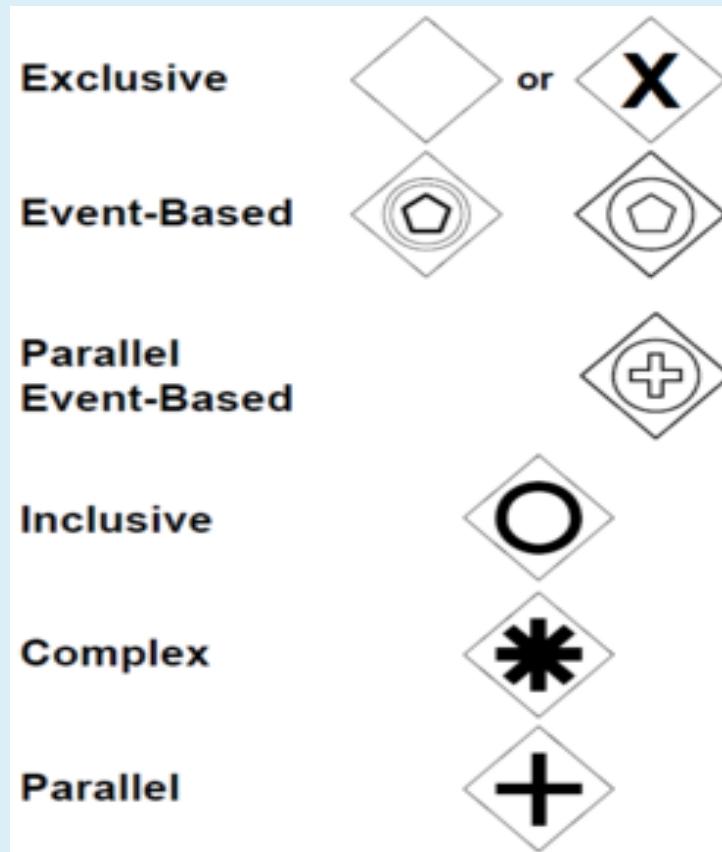
EVENT QUALIFIERS

Symbol	Type	Description
	Message	It receives or sends messages. Exemples: Phone call, send form, service request, etc.
	Timer	Expresses a time gap in processing or a wait for a period of time. Intermediate timer is a specific time-date or a specific cycle. Exemples: wait for 5 minutes, everyday at 8 o'clock.
	Signal	It is used to receive or transmit signals to one or more participants. Examples: approved expense budget, planned production order.
	Error	It is used to catch exceptions and may take place at the end of a process. Example: Insufficient information.
	Conditional	The process is triggered when a particular data condition or business rule is met. Examples: stock level below minimal, check amount for account exceeds account balance
	Escalation	It shows an alternative flow for solving an issue. Its main advantage ist hat it doesn't interrupt activity execution. It is mainly used with or within sub-processes. Examples: late delivery.

EVENTS - EXEMPLE

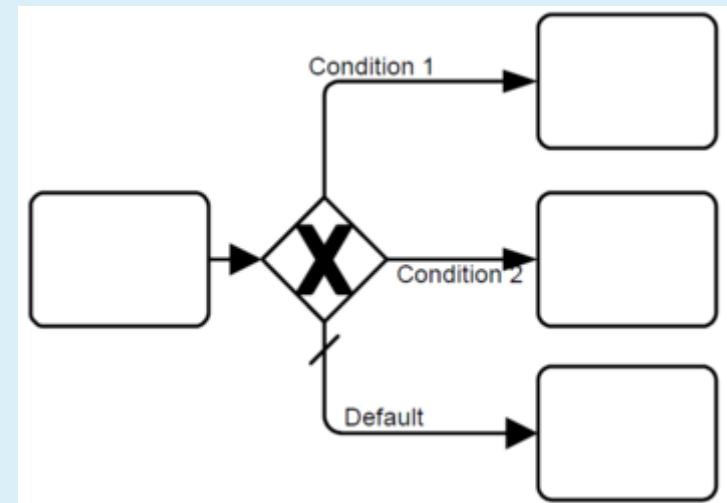
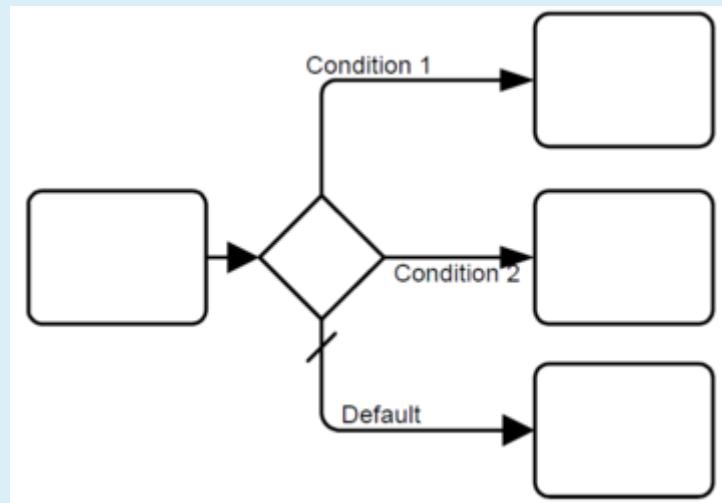


GATEWAY TYPES

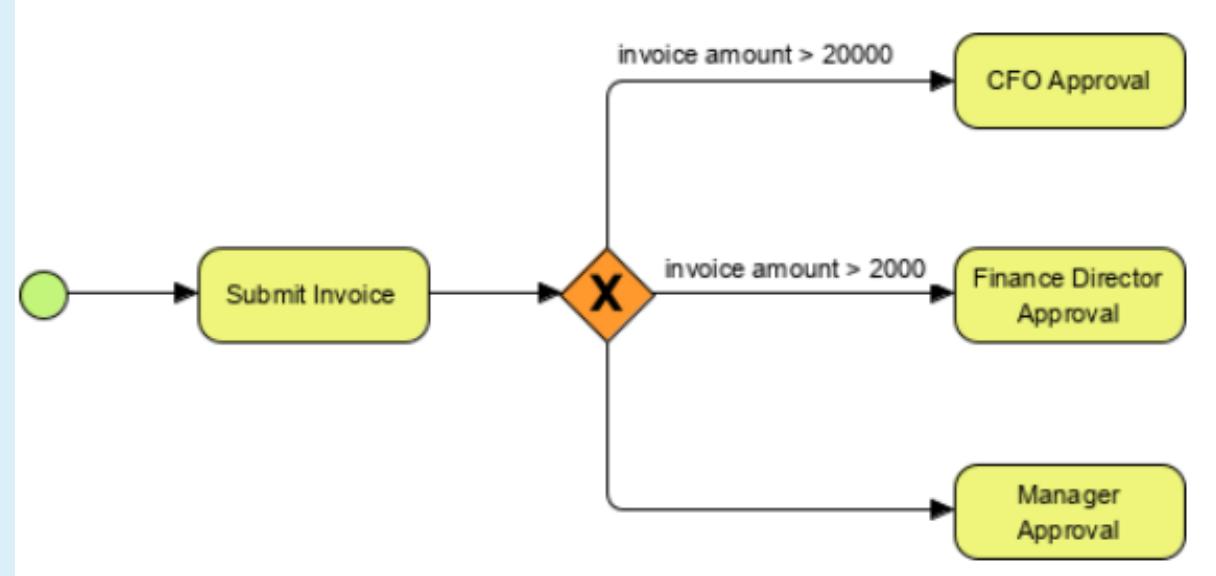
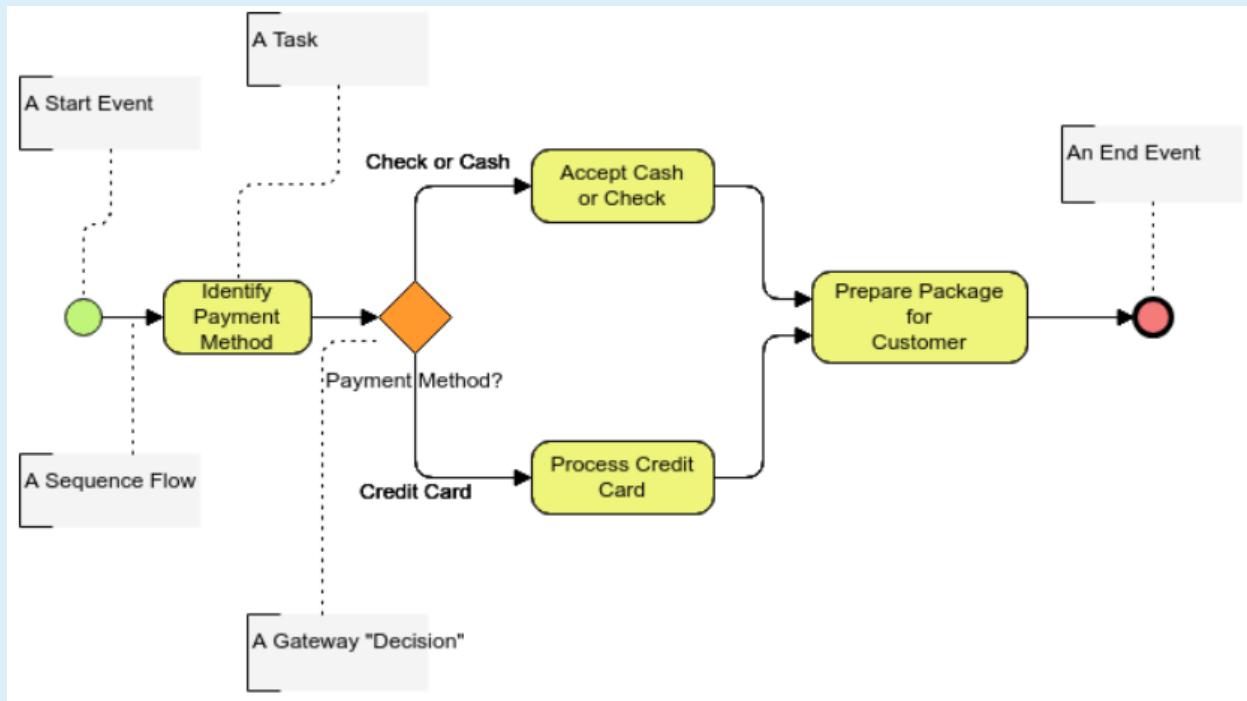


A. EXCLUSIVE GATEWAYS

- They are also called **Decisions** and are used to create alternative paths within a Process flow
- Only one of the paths can be taken, this means the gateway is exclusive
- A Decision can be seen as a question that is asked at a particular point in the Process
- The question has a defined set of alternative answers (condition Expression)

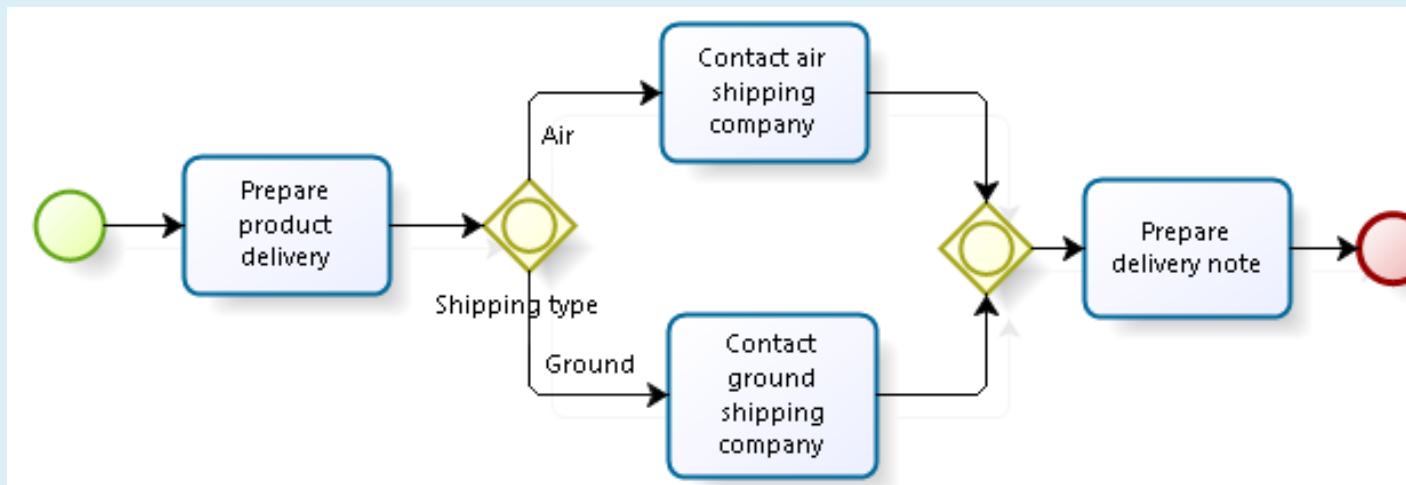


EXCLUSIVE GATEWAYS - EXAMPLE

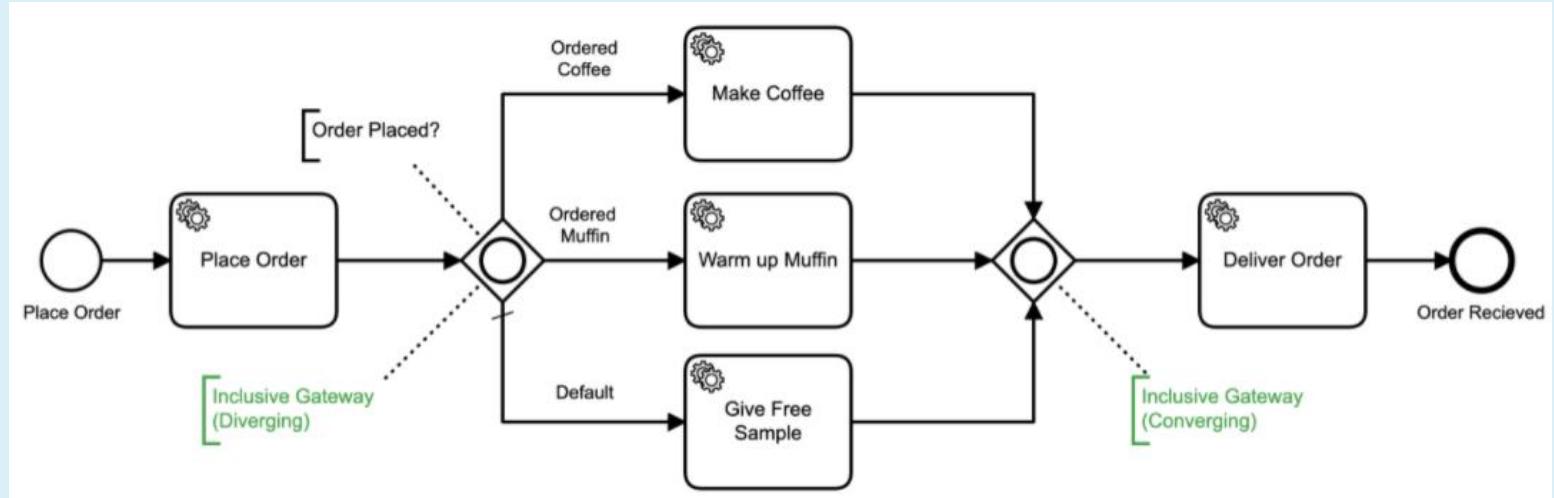
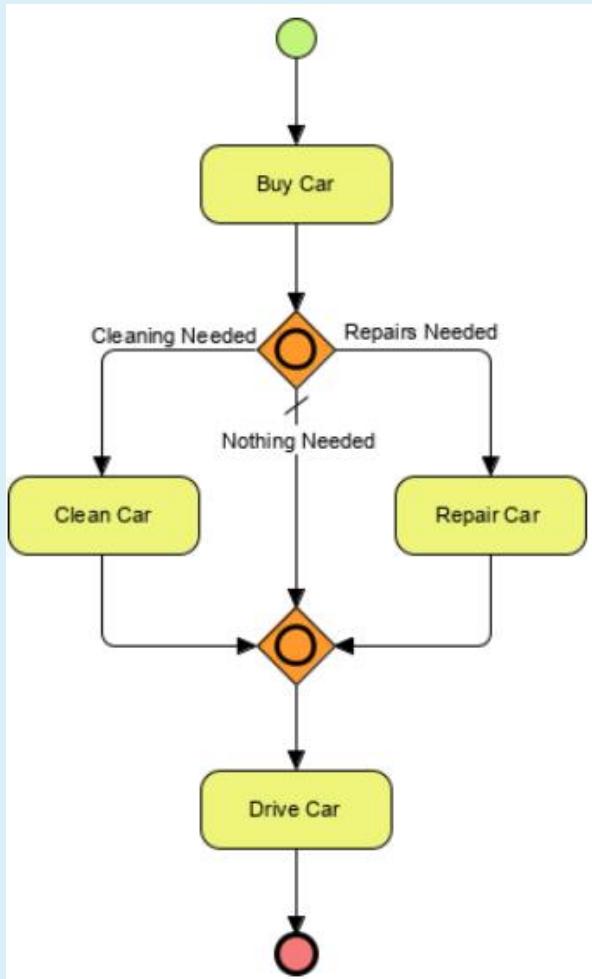


B. INCLUSIVE GATEWAYS

- It can be used to create **alternative**, but **also parallel paths** within a Process flow
- All condition Expressions are evaluated; the true evaluation of one condition Expression **does not exclude** the evaluation of other condition Expressions
- Since each path is considered to be independent, all combinations of the paths **MAY** be taken, from zero to all
- However, it should be designed so that at least one path is taken.

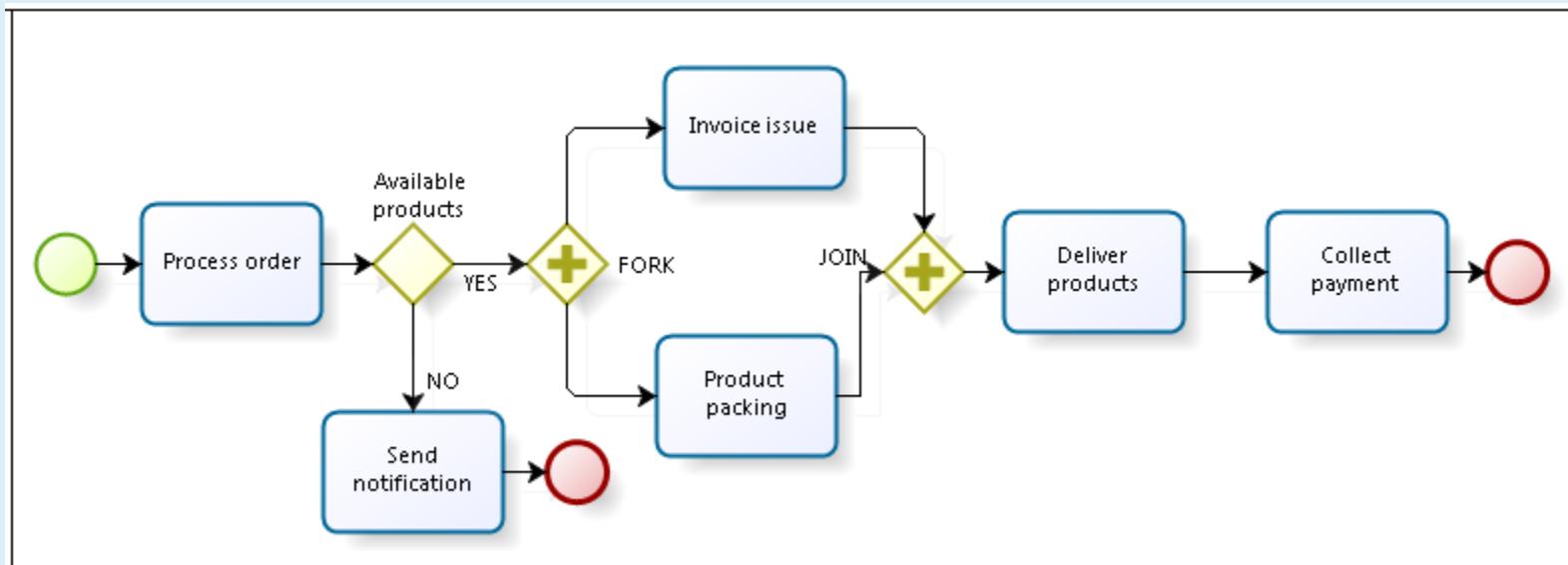


INCLUSIVE GATEWAYS - EXAMPLE

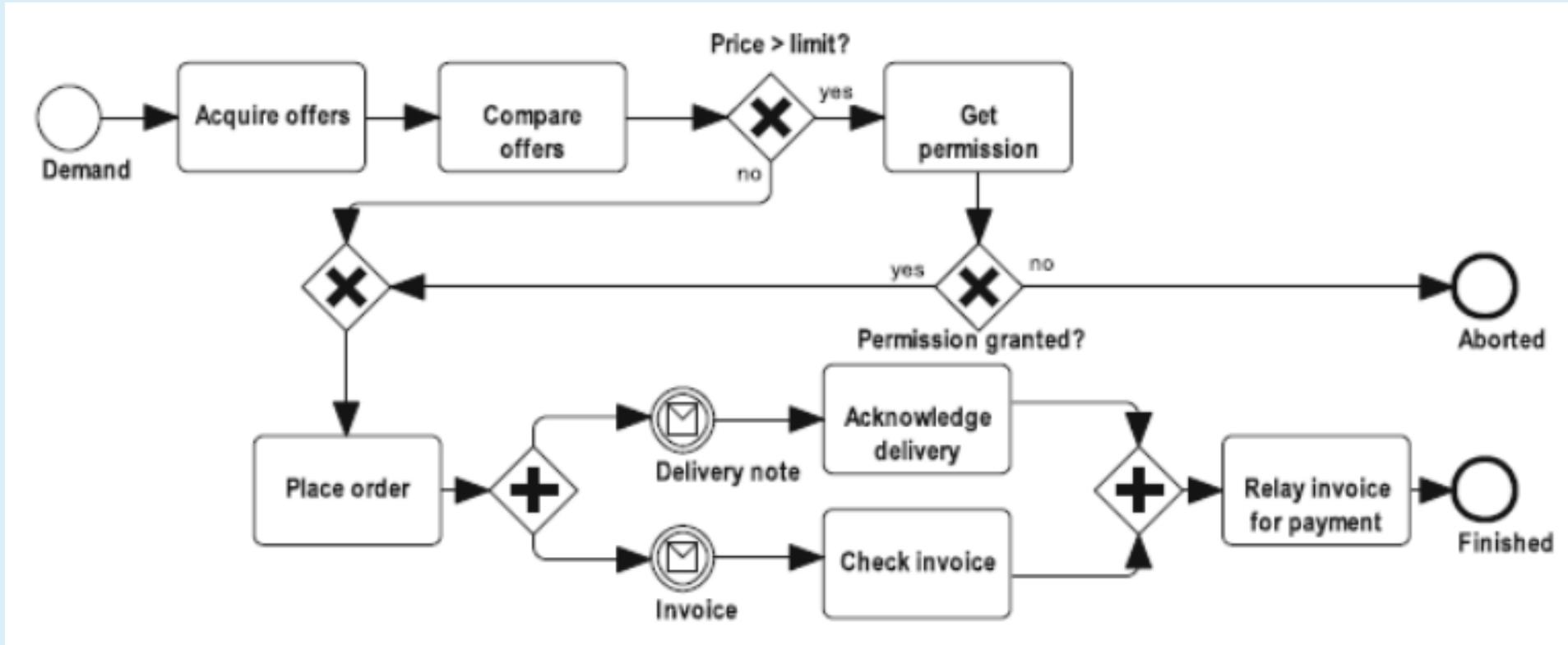


C. PARALLEL GATEWAYS

- A Parallel Gateway is used to synchronize (combine) parallel flows and to create parallel flows
- A Parallel Gateway creates parallel paths without checking any conditions
- For incoming flows, the Parallel Gateway will wait for all incoming flows before triggering the flow through its outgoing Sequence Flows

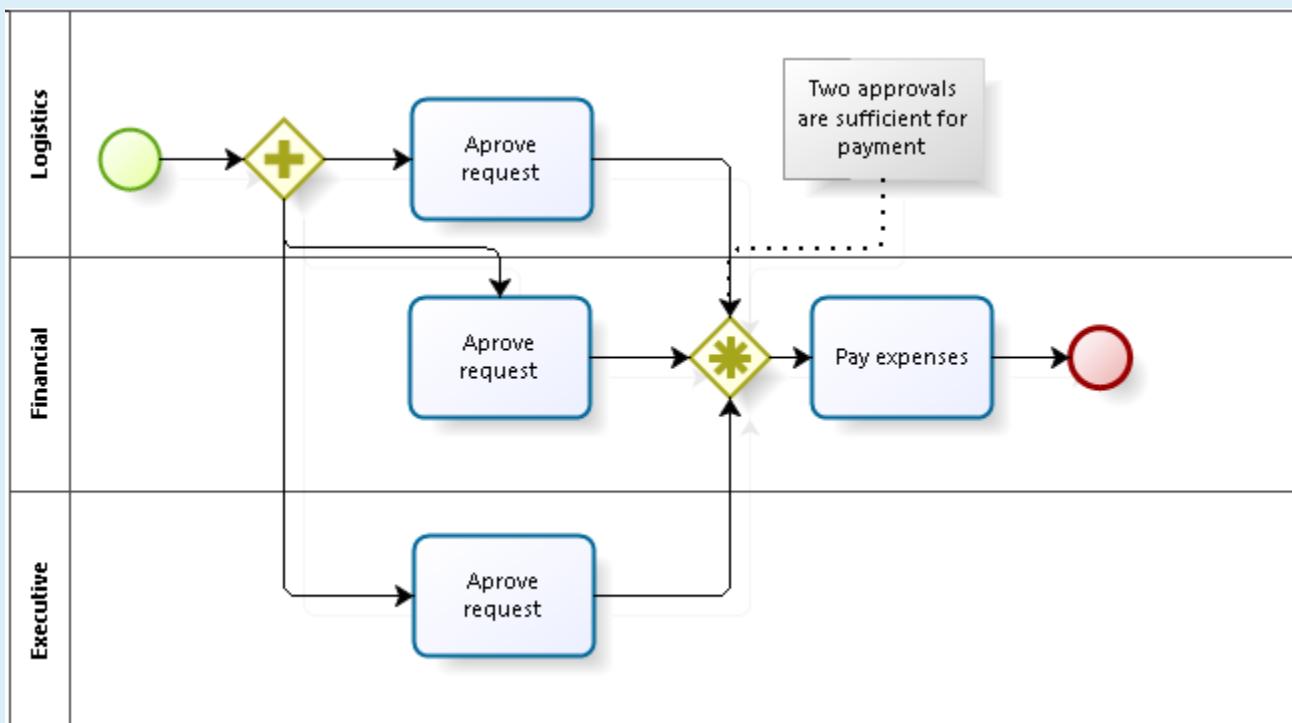


PARALLEL GATEWAYS - EXAMPLE



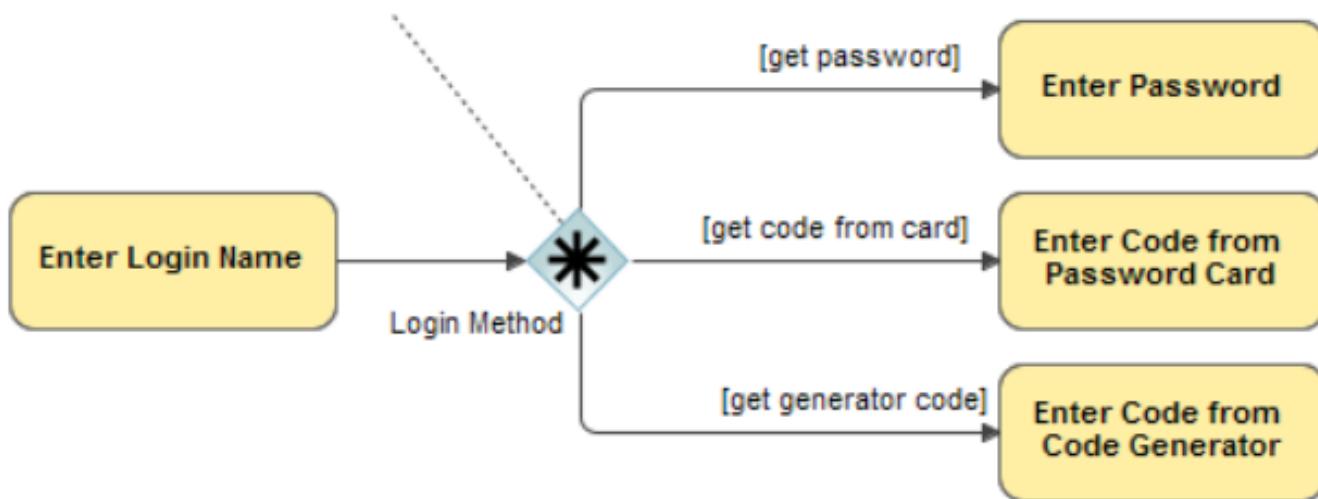
D. COMPLEX GATEWAYS

- The Complex Gateway can be used to model complex synchronization behavior. An Expression is used to describe the precise behavior.
- For example, this Expression could specify that tokens on three out of five incoming Sequence Flows are needed to activate the Gateway
- What tokens are produced by the Gateway is determined by conditions on the outgoing Sequence Flows as in the split behavior of the Inclusive Gateway



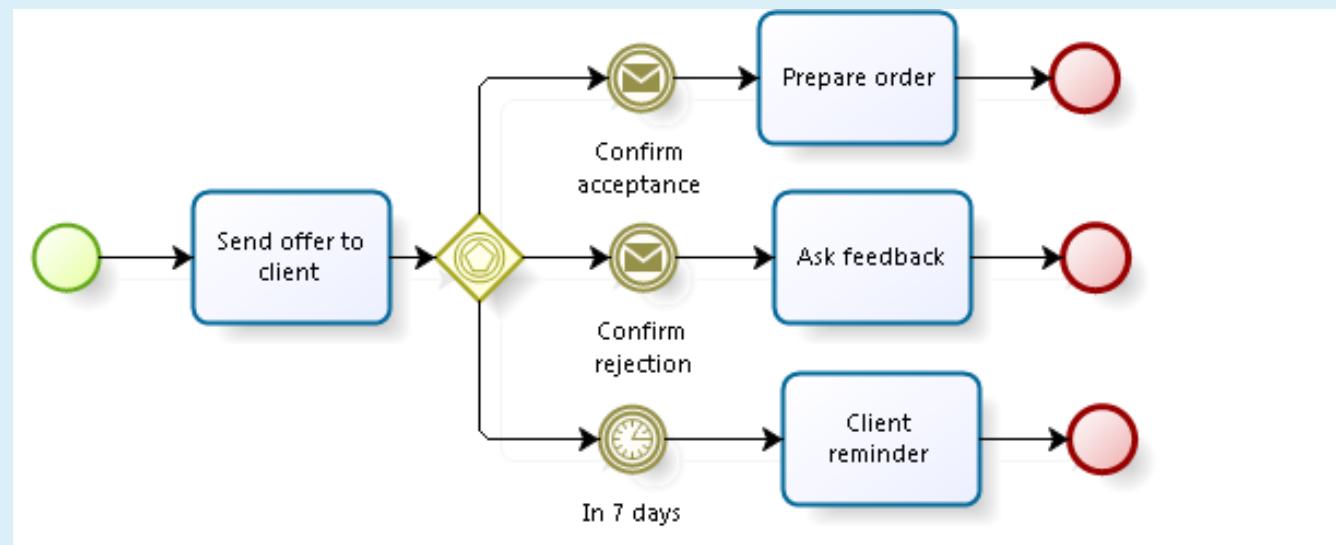
COMPLEX GATEWAYS

According to the Login properties, users are required to enter either password and code from the password card or enter code from code generator

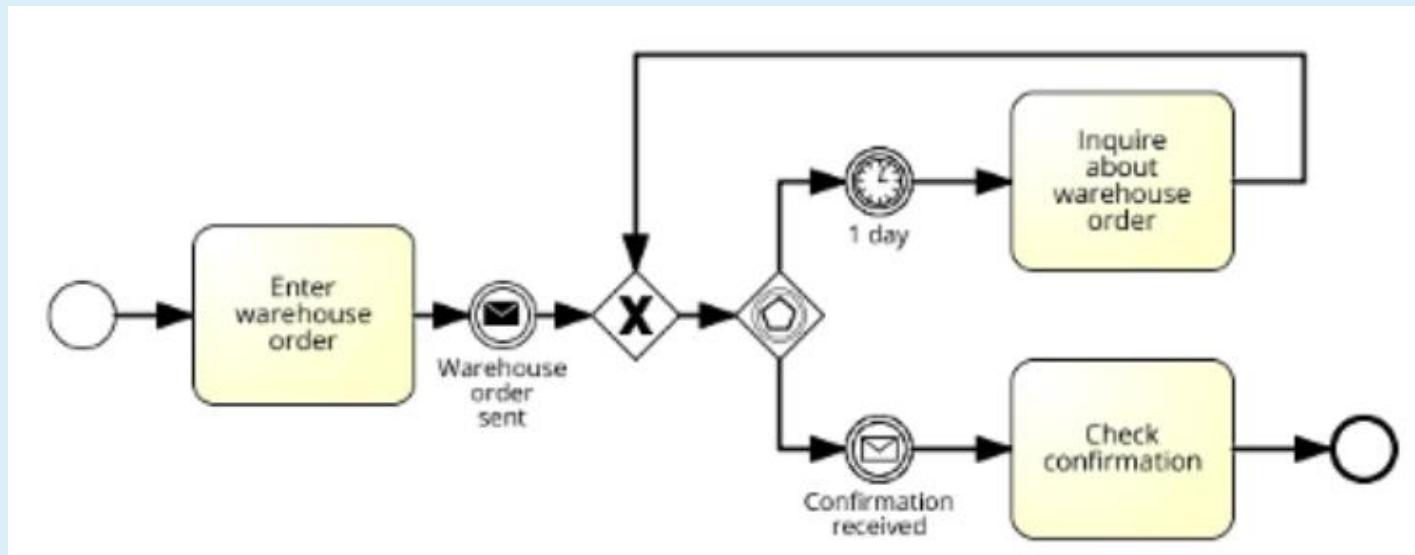
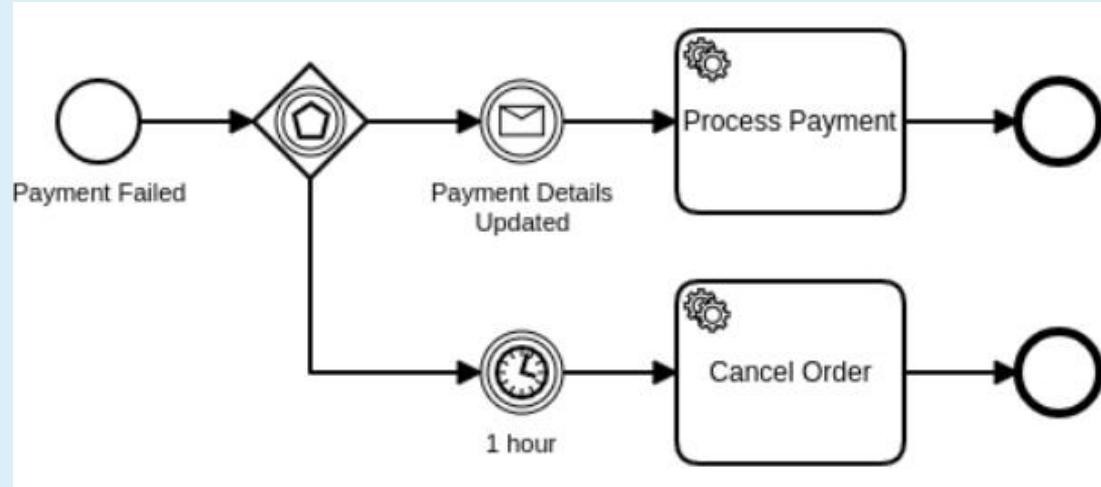


E. EVENT-BASED GATEWAYS

- The Event-Based Gateway represents a branching point in the Process where the alternative paths that follow the Gateway are based on Events that occur
- A specific **Event**, usually the receipt of a **Message**, determines the path that will be taken
- Basically, the decision is made by another Participant, based on data that is not visible to Process, thus, requiring the use of the Event-Based Gateway.

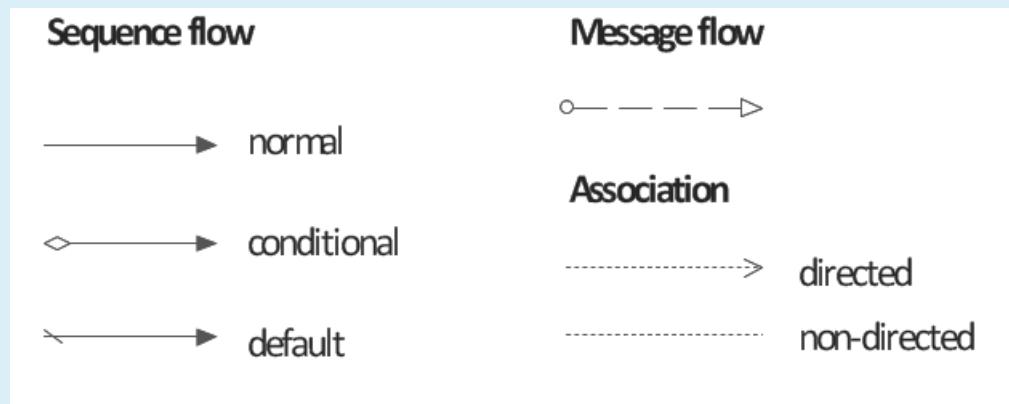


EVENT-BASED GATEWAYS

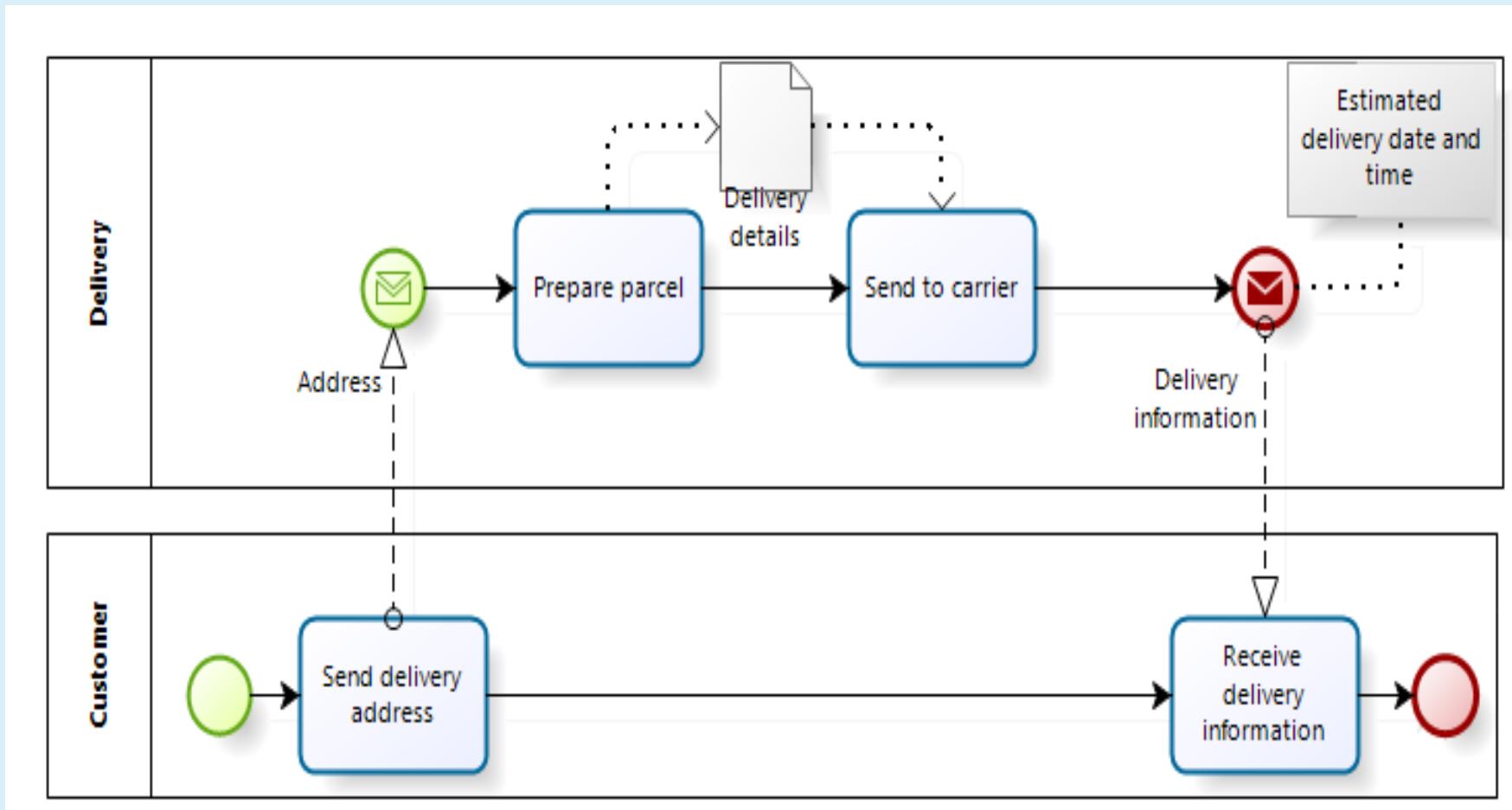


2. CONNECTING OBJECTS

- **Sequence flow** is used to connect flow elements. It shows the order of flow elements.
- In BPMN, the communication between pools is achieved by the use of message. **Message flow** is used to show the flow of messages between pools or flow elements between pools.
- A **data association** shows the information flow between activities of a business process.
- **An association** links artefacts with other BPMN graphical elements



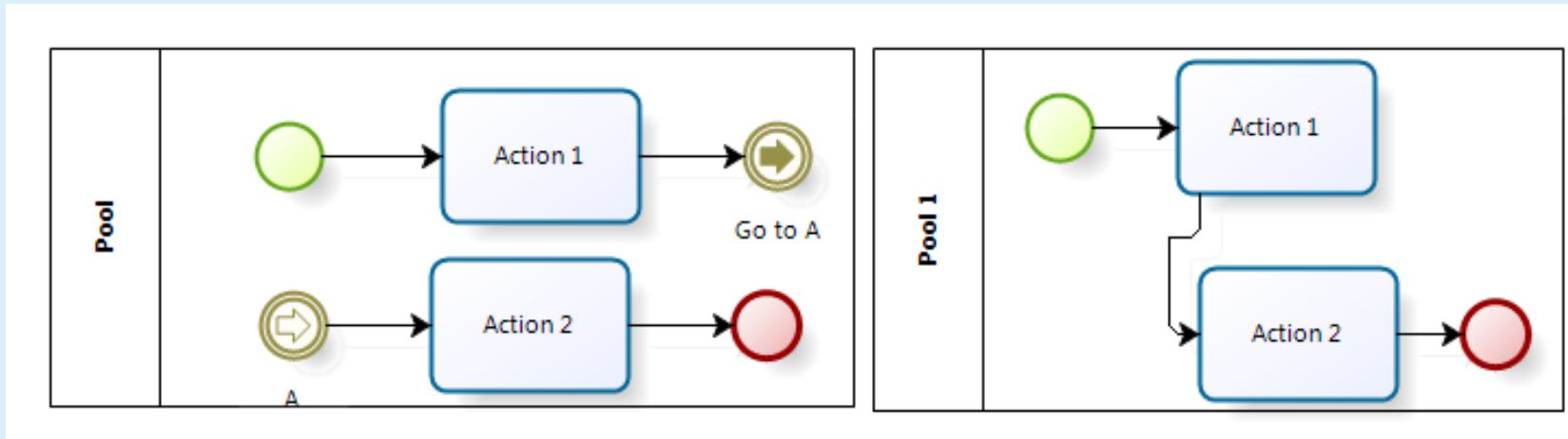
Exemple of connecting objects



SEQUENCE FLOWS

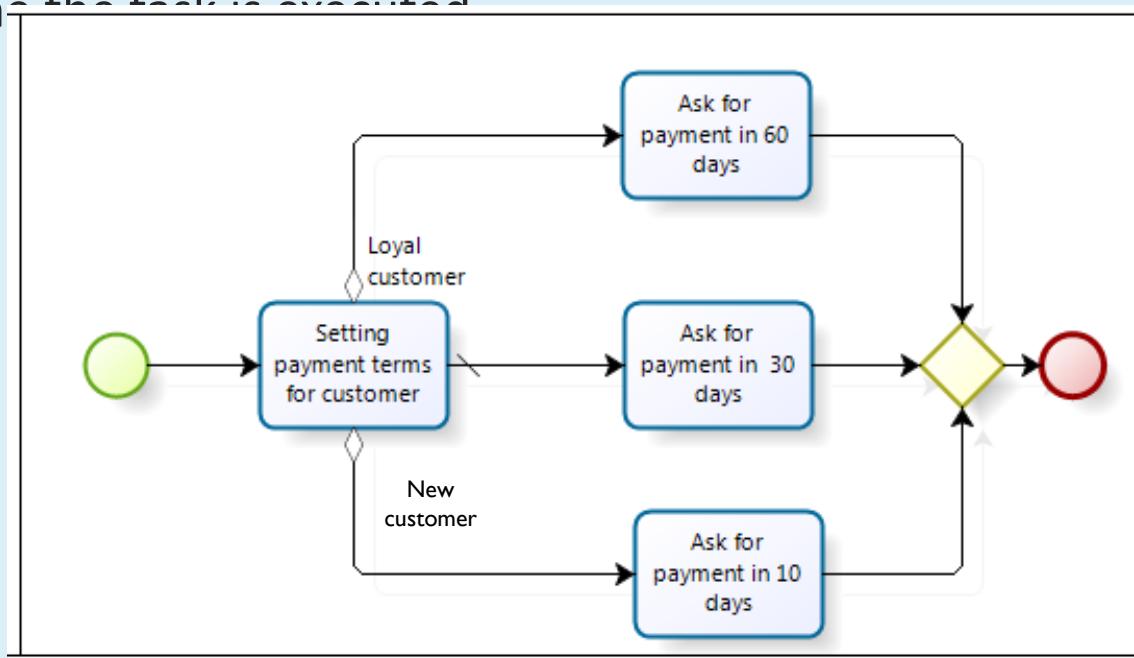
- They can connect the following elements: events (start, intermediate and end events), actions, sub-processes and gateways.
- Limitations of a sequence flow:
 - It cannot represent an entry for a start event;
 - It cannot represent an exit for an end event;
 - It cannot directly connect an action of a process with an action of a sub-process; the link must be correctly established between action and sub-process.
 - It is allowed only within a pool; for interactions between pools, message flows must be used.
 - It cannot be used to connect artefacts to other modeling elements; association must be used.
 - It can be replaced by intermediate link events, but both intermediate link events must belong to the same pool.

LINK EVENTS- EXAMPLE



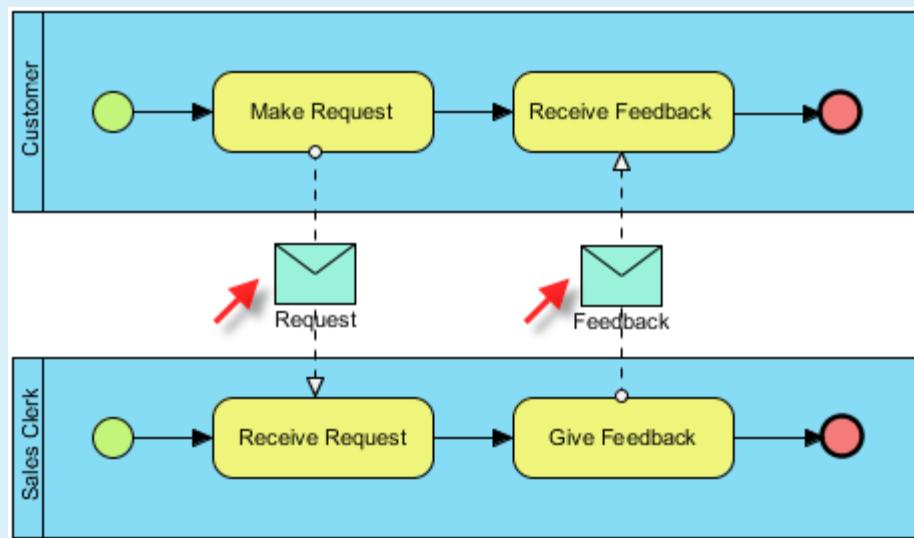
CONDITIONAL SEQUENCE FLOWS

- A sequence flow can define a condition when connecting an inclusive or an exclusive gateway or a task; it will be called **conditional sequence flow**.
- It is **an alternative way to represent branching and merging without the usage of gateways**
- When using conditional flows we must pay attention to the set of conditions represented by the outgoing flows: they have to lead to a valid result for each time the task is executed



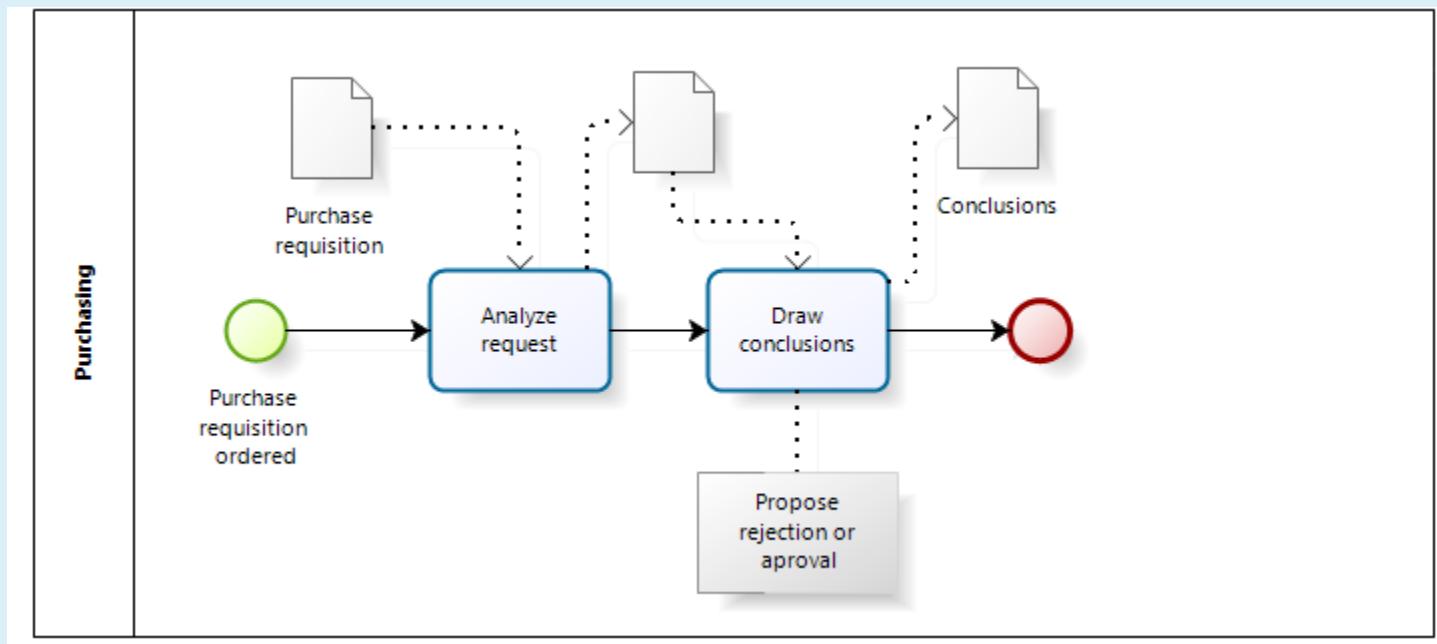
MESSAGE FLOWS

- A message flow is used to show message exchange between two participants that are prepared to send and receive messages. In BPMN, this two participants will be represented by two separate pools within a collaboration diagram.
- Optionally, message flows can be extended with a **message object**, that will be associated with the message flow or overlapping the message flow. Message objects explicitly describe the communication content exchanged between the two participants.



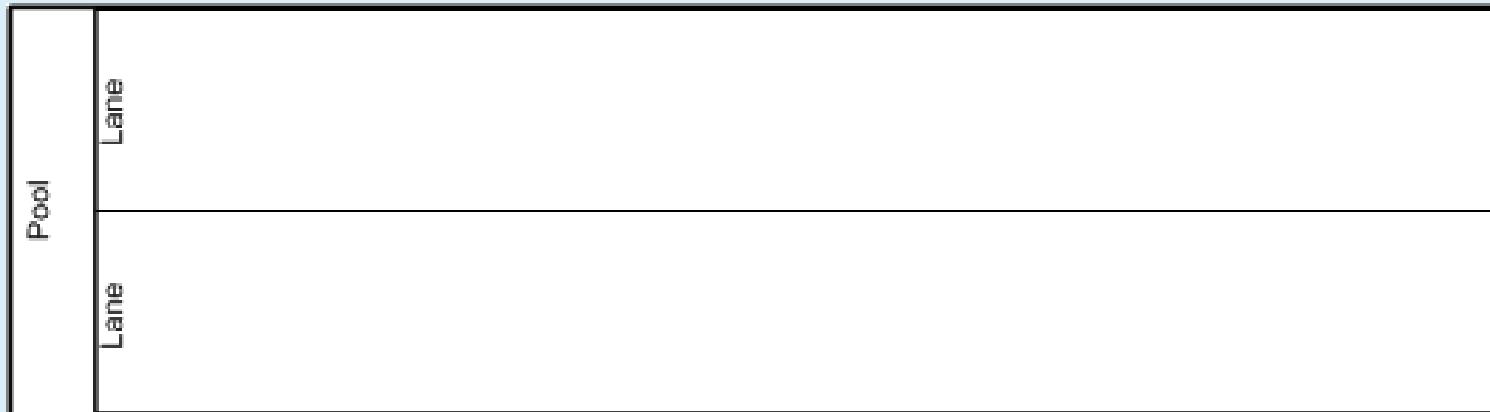
DATA ASSOCIATIONS

- To represent process data flows, BPMN uses **data association** = a bidirectional association. Data associations are used for transferring data between processes or tasks.
- They have no effect on the process task flow, but to show what data a process or task needs and/or produce.



3. PARTITIONING OBJECTS

- Represent a mechanism to organize activities into separate visual categories in order to highlight different functional capabilities or responsibilities.
 - **Pool:** represents a participant in the process. It involves organizational units or physically separated participants.
 - **Lane:** used to organize and share activities. They are placed inside a container and may be nested.

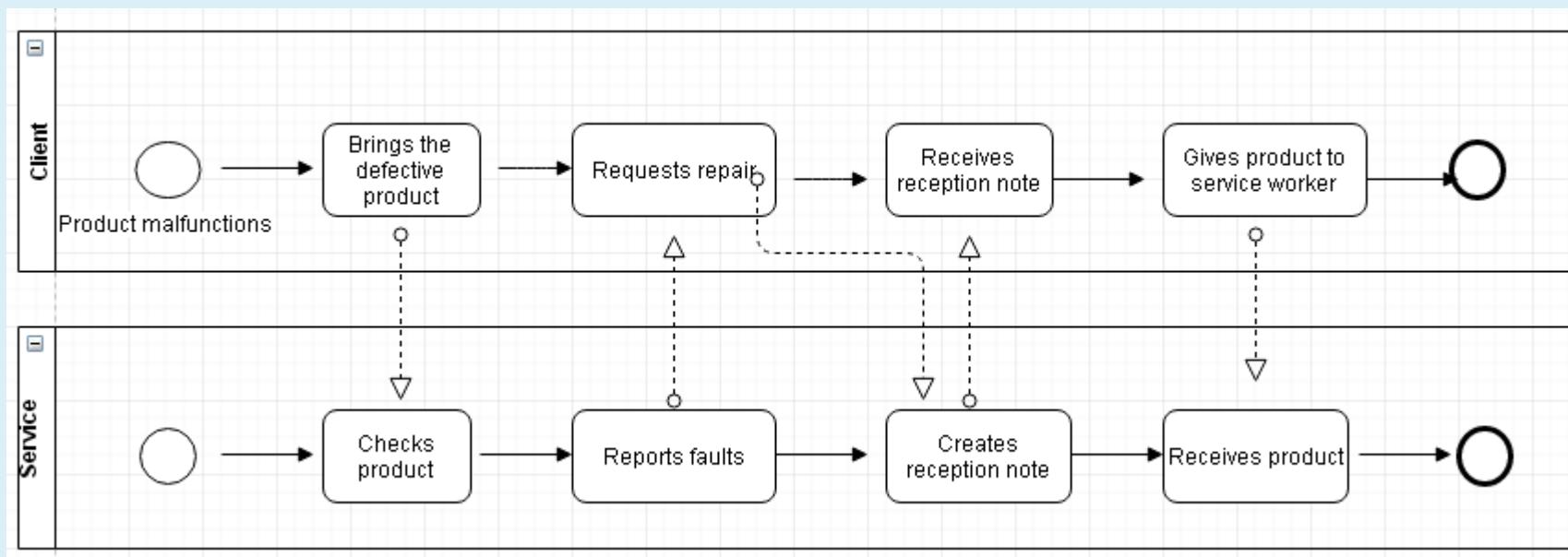


PARTICIPANTS

- A **participant** is an entity identified in the business model, who performs or has certain responsibilities in executing activities in a process and acts as a participant in a collaboration.
- From the perspective of BPMN language, a participant is visually represented in the form of a **container (pool)** BPMN specification distinguishing between two levels of participation :
 - organizational unit, representing an internal or external interest group of the organization, such as the company or a department;
 - role associated to the execution of an activity such as client, supplier, producer etc.

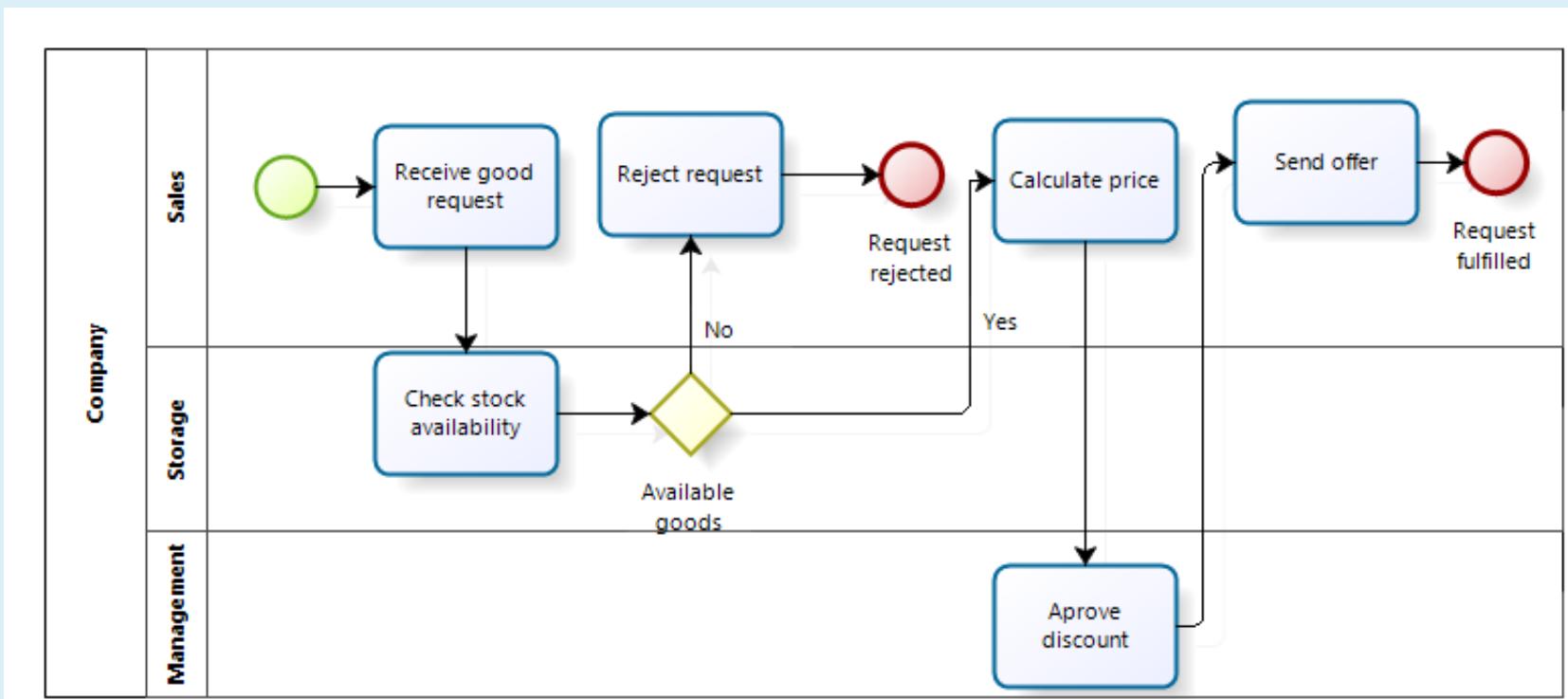
SEQUENCE AND MESSAGE FLOWS

- A container encapsulates a sequence of activities from a process, which means that the sequence flows **can not cross the boundaries** of a container.
- The container name does not necessarily mean **an organizational unit**, it can specify the **name of the process** itself, such as "Product Reception" or "Request repair".



POOL PARTITIONING WITH LANES

- Lanes enable responsibility identification **within a business process**.
- A sequence flow can cross over lanes to accomplish process specific tasks.

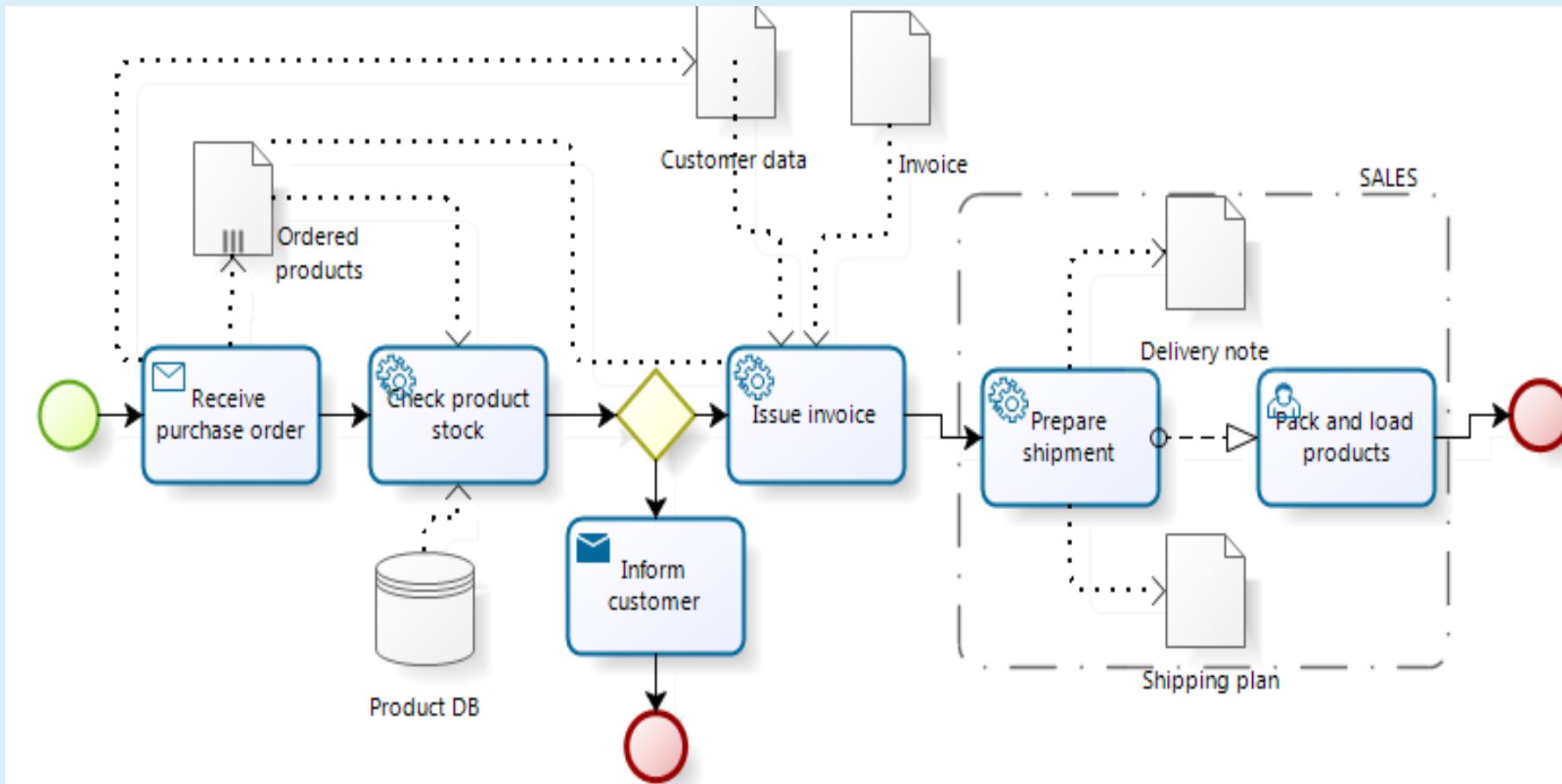


4. DATA OBJECTS

- They are BPMN mechanism used to show what data are necessary to a task and/or produced by a task. They are connected to other elements by **data associations**.
- **Types:**
 - Data objects
 - Input data
 - Output data
 - Data store
 - Collection – can be applied to any of the above



DATA OBJECT TYPES



5. ARTEFACTS

- **Annotation:** mechanism used to specify additional information.
- **Group:** grouping element used for documentation and analysis. It does not impact the sequence flow.

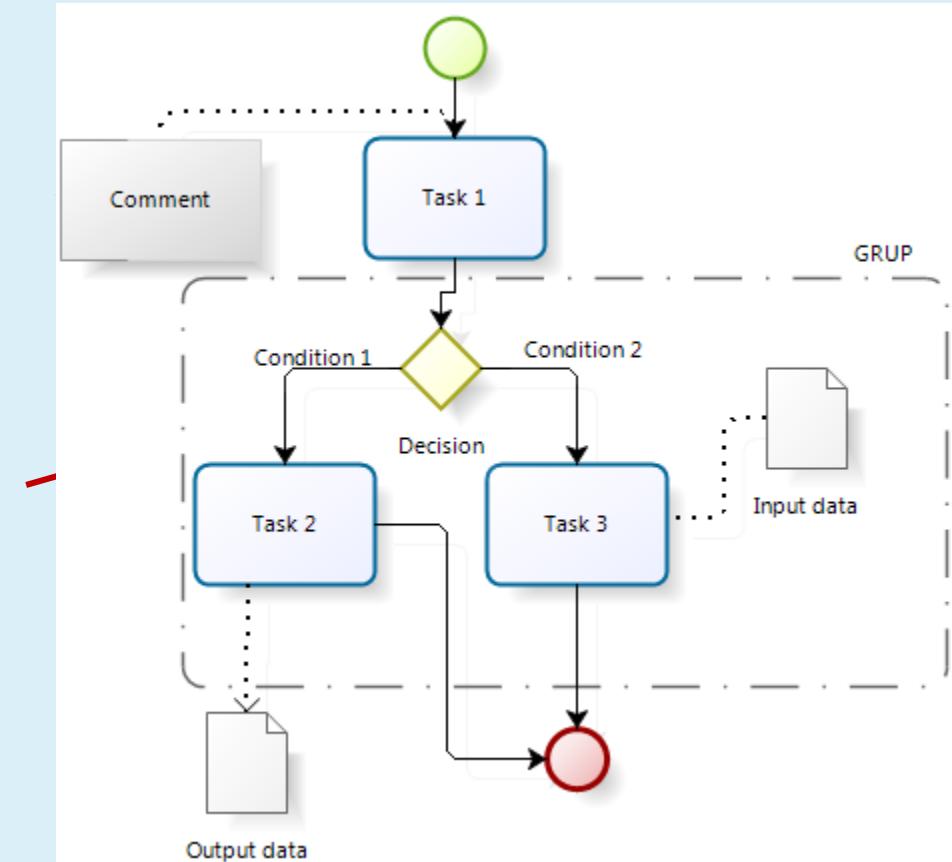


DIAGRAM TYPES

- BPMN 2.0 specification contains four types of models:
 1. Business process diagram— has only one container
 2. Collaboration diagram— **has more containers**
 3. Choreography diagram
 4. Conversation diagram
- Business process diagram, the most detailed one, is the most used in real life. The other three diagrams can be considered as a synthetic representation of business processes knowledge.

WHAT IS BUSINESS PROCESS OPTIMIZATION?

- **Business process optimization** is the practice of increasing organizational efficiency by improving processes
- Examples of optimization include:
 - Eliminating redundancies
 - Streamlining workflows
 - Improving communication
 - Forecasting changes

WHAT IS RPA?

- Robotic process automation (RPA) is the use of software with artificial intelligence (AI) and machine learning capabilities to handle high-volume, repeatable tasks that previously required humans to perform.
- RPA takes productivity optimization to the next level by redefining work and reassigning employees to execute higher-value activities.
- Process bots are capable of independently performing simple human-like functions such as interpreting, deciding, acting, and learning.

8th lecture – Design of computer system architecture

Agenda

- ✓ General considerations on computer system design
- ✓ Database design
- ✓ User interface design
- ✓ Design of computer system architecture
 - A. Client-server architecture
 - B. Service oriented architecture



General considerations on computer system design

- Computer system design consists of:
 - Establishing the **logical design solutions** and
 - **Physical specifications** for the new system components;
- The design process is mainly based on the results of the previous activities:
 - Defining the **implementation solution** for the new system
 - **Modelling of the new system**

Design process objectives

Analysis activities

- **Objectives:** to understand
- Events and processes of the company
- System activities and processing requirements
- Information requirements and storage

Analysis models

Documents

Design activities

Objectives:

- To define, organize and structure the components of the solution system

The goal of systems design

- to build a system that is effective, reliable, and maintainable:
 - **Effective:** it supports business requirements and meets user needs.
 - **Reliable:** it handles input errors, processing errors, hardware failures, or human mistakes. A good design will anticipate errors, detect them as early as possible, make it easy to correct them, and prevent them from damaging the system itself. A reliable system should be available nearly all of the time and proper backups maintained in case of system failure.
 - **Maintainable:** it is flexible, scalable, and easily modified. Changes might be needed to correct problems, adapt to user requirements, or take advantage of new technology.

General considerations on computer system design

- Some methodologies divide system design in two parts:
 - A. **Architectural design**/preliminary design
 - B. **Detailed design** includes:
 - i. Use case design
 - ii. Database design
 - iii. User interface and external system interface design
 - iv. System control and security design
- Within these stages, the computer system is **logically** and **physically** designed, separately or not.

Key questions for design activities

Design activity	Question
i.Execution environment design	Have we specified in detail the environment and all the various options in which the software will execute?
ii. Application and software architecture design	Have we specified in detail all the elements of software and how each use case is executed?
iii.Design system interfaces with external systems	Have we specified in detail how the system will communicate with all other systems inside and outside the organization?
iv.Database design	Have we specified in detail all the information storage requirements, including all the schema elements?
v.Design of system controls and security	Have we specified in detail all the elements to ensure the system and the data are secure and protected?

i. System environment design

Technical architecture

i. Design the environment

- **THE ENVIRONMENT** represents all of the technology required to support the software application
 - Servers, desktop computers
 - Mobile devices, operating systems
 - Communication capabilities, input and output capabilities
- It is also called the **Technology architecture**
- Designing the environment: identifying and defining all the types of computing devices that will be required.
 - identifying all the locations and communication protocols necessary to integrate computing hardware.

i. Design the Environment (TECHNOLOGY ARCHITECTURE)

1. Design for Internal deployment

- a. **Stand alone** software systems- run on one device without networking
- b. **Internal network-based** systems:
 - Local area network, client-server architecture
 - Desktop applications and browser-based applications
- c. Three layer client-server architectures:
 - Data layer, domain layer and presentation/view layer
 - Desktop applications and browser-based applications

i. Design the Environment (TECHNOLOGY ARCHITECTURE)

2. Design for External deployment

- Configuration for **Internet deployment**– with advantages and risks
- Hosting alternatives for **Internet deployment**
 - Colocation
 - Managed services
 - Virtual servers
 - Cloud computing
- Diversity of client devices with Internet deployment
 - Laptop, tablets and notebooks, smart phones etc

Attributes of hosting options

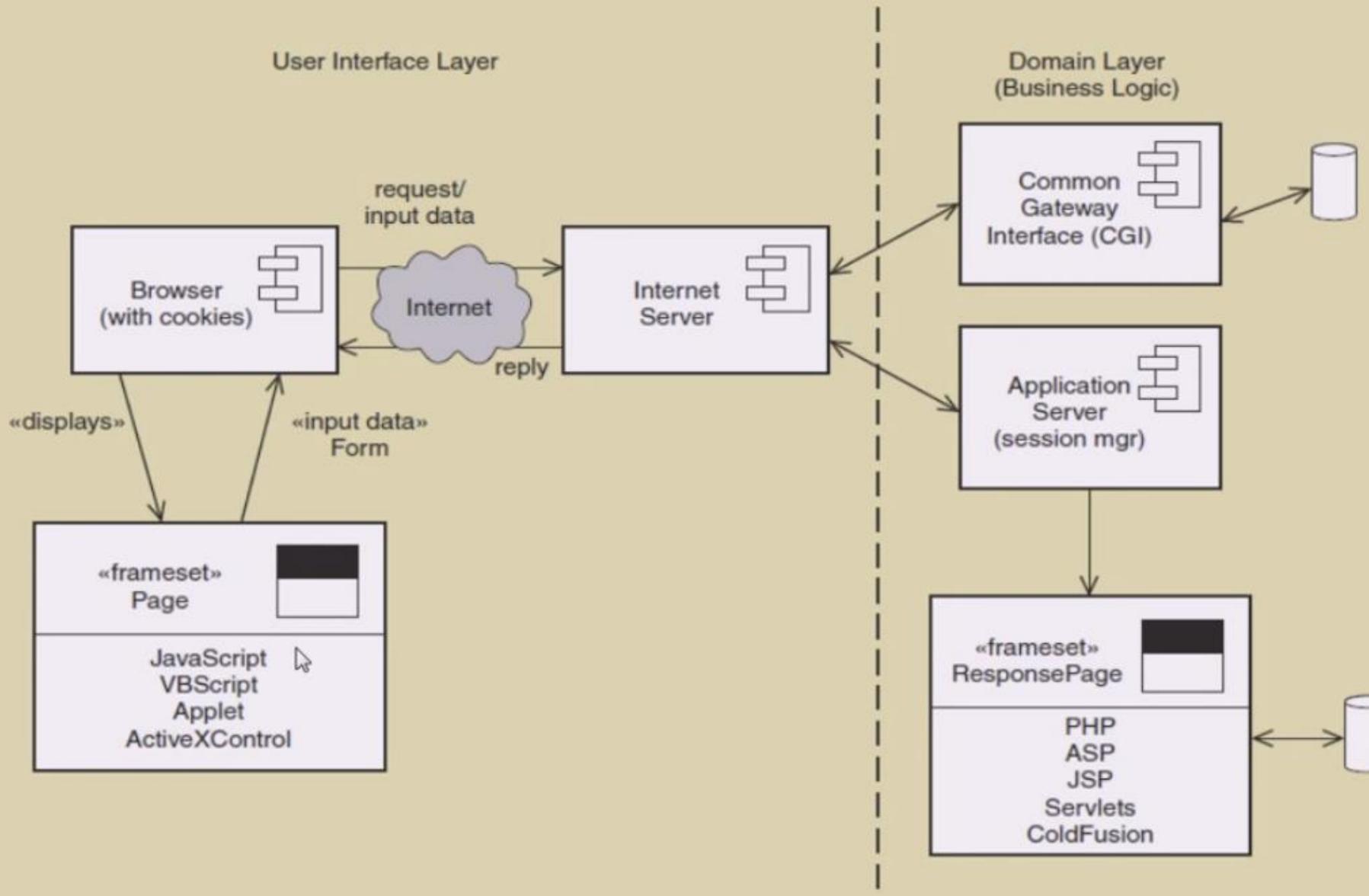
HOSTING OPTIONS				
Service options	Colocation	Managed services	Virtual servers	Cloud computing
Hosting service provides building and infrastructure	Yes	Yes	Yes	Yes
Client owns computer	Yes	Perhaps	No	No
Client manages computer configuration	Yes	No	Possible	No
Scalability	Client adds more computers	Client adds more computers	Client buys larger or more virtual servers	Client adds small increments of computing power
Maintenance	Client provides	Host provides	Host provides	Host provides
Backup and recovery	Client provides	Host provides	Available	Available

Configuration for Internet deployment

- **Advantages**
 - **Accessability** – Web-based applications are accessible to a large number of potential users (including customers, suppliers, off-site employees)
 - **Low cost communication**
 - **Widely implemented standards**—Web standards are well known
- **Potential problems :**
 - **Security**– Web servers are target for security breaches because Web standards are open and widely known, accessible to hackers
 - https – Hypertext Transfer Protocol Secure
 - TLS – Transport Layer Security – Advanced ver. of SSL protocol
 - **Troughput** – When high load occur, troughput and response time can suffer significantly.
 - **Changing standards**—Web standards change rapidly (functionality vs compatibility)

Design for remote, distributed environment

- applications for a remote, distributed environment are often internal systems used by employees of a business
- **Virtual private network (VPN)**
 - Closed network with security and closed access, built on top of a public network (Internet)
- An alternative way to implement remote access is to construct an application that uses a Web-browser interface.
->accessible from any computer with an Internet connection.
- Two interfaces to same Web applications for internal / external access
 - Back end, front end user interface to same Web app
 - Not as secure



Design of system architecture: networks

- The most popular **types** of networks are :
 - Bus/ point-to-point network;
 - Ring network;
 - Star network;
 - Hierarchy network.
- After selecting the network type, the analysis team must identify the **communication protocol**. The most common communication protocols are:
 - **TCP / IP**. It is a protocol indicated when using an Ethernet network or when the network computers have different architectures;
 - **SNA**. It is generally used to connect IBM mainframes.

ii.Design the application architecture and software

Software architecture

ii. Design the application architecture and software

- Partition system into **subsystems**
- Define **software architecture**
 - Three level or
 - Model-view-controller
- Detailed design of each **use case**
 - Design class diagrams
 - Sequence diagrams
 - State machine diagrams

Architectural styles/ patterns

- 1. Pipes and filters**
- 2. Event driven**
- 3. Client-server**
- 4. Model view controller**
- 5. Layered**
- 6. Database centric**
- 7. Three tier**

1. Architectural patterns: Pipes and filters

- a style widely used for Unix scripts and for signal processing applications
- a series of processes connected by pipes; the output of a process serves as input for the next
- the processing can start as soon as some of the inputs is available
- used also for many audio and video processing applications.

2. Architectural patterns: Event driven

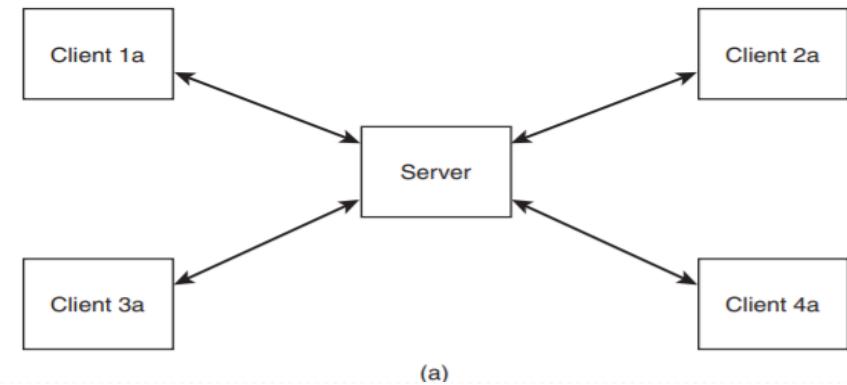
- **components react to externally generated events and communicate with other components through events**
- **Used by:**
 - **modern graphical user interface libraries**
 - **Many distributed systems**
- **it allows for decoupling of the components**

3. Architectural patterns: Client/server

- The client / server architecture is a set of three main components: a server, a client, and a network that connects client computers to servers to collaborate on performing tasks.

- The types of client / server applications are:

- Database systems
- E-mail systems
- Groupware systems
- Legacy systems



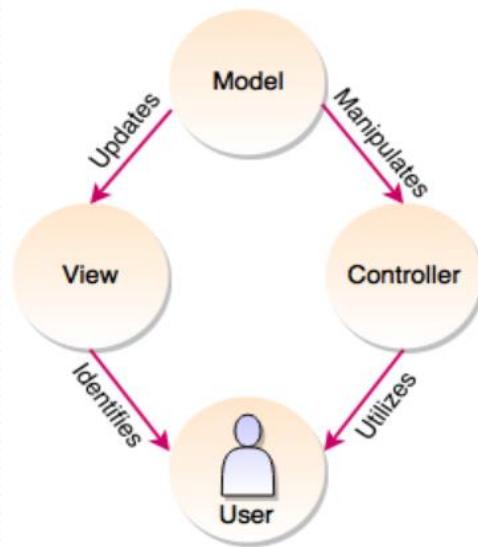
- A distributed architecture involves the existence of multiple databases (located on separate computers) and of applications that manipulate data from different local workstations using database management systems (DBMSs).

4. Architectural patterns: Model View Controller (MVC)

- It is a software architectural design for implementing user interfaces and Web frameworks on computers and is a **standard design pattern**.
- MVC architecture helps to write better organized and more maintainable code
- This architecture is used and extensively tested **over multiple languages** (Java, PHP, ASP.NET etc.) and generations of programmers.
- MVC is being used as the powerful framework for **building web applications** using MVC pattern.
- MVC separation helps to **manage complex applications**. It is the main advantage of separation and also simplifies the team development.

MVC Model-View-Controller

- Controller is *in charge*, taking care of the data received from Model and injecting it into View



Model: central component of MVC, it manages the ***data, logic and constraints*** of an application.

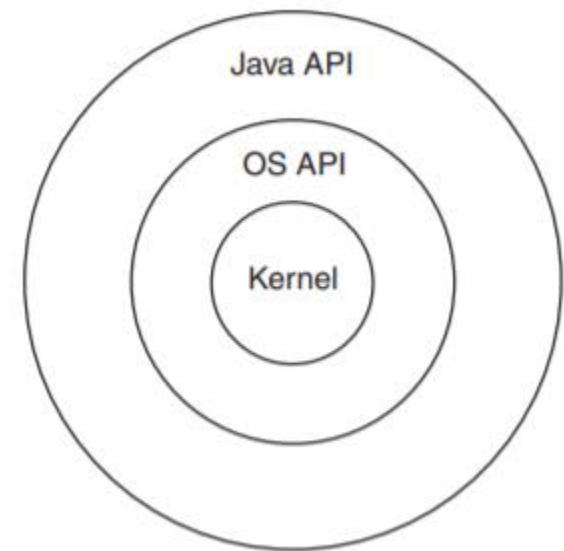
It captures the *behavior* of an application

View: responsible for displaying all or a portion of the data to the user. It formats and presents the data from model to user.

Controller: controls the interactions between the Model and View. It acts as an interface between the associated models, views and the input devices.

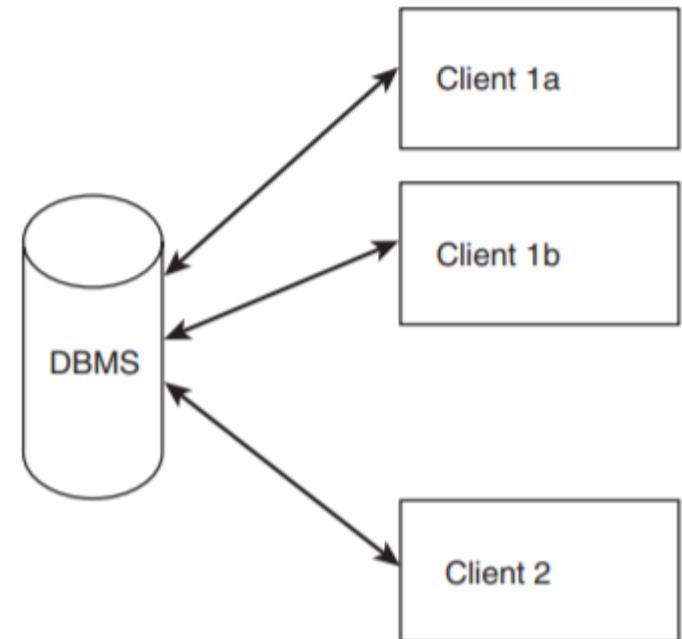
5. Architectural patterns: layered

- components are grouped into layers
- a component communicate only with other components in the layers immediately above and below their own layer
- Layers=tiers that reside on different computers in client server architectures
- keeps the components focused on specific tasks and facilitates the detection of problems



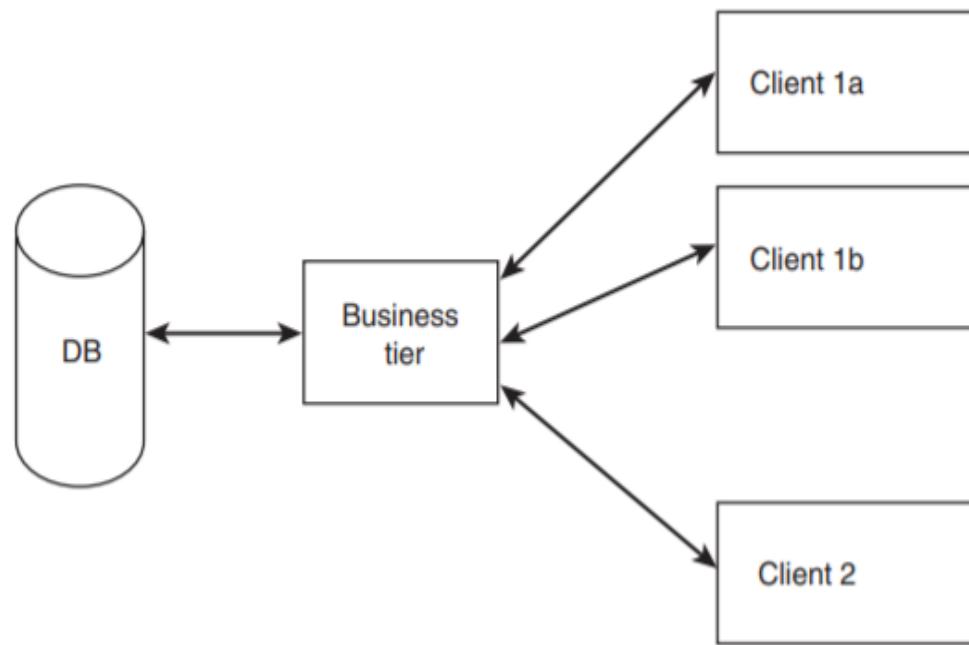
6. Architectural patterns: database centric

- A central database and separate programs access the database
- They communicate only through the database, not directly
 - DBMS - a layer of abstraction for the database that guarantees many user-defined constraints
 - a relatively coupled system of multiple programs to the database



7. Architectural patterns: three- tier

- a variation of the database centric and client-server approaches that adds a middle tier between the clients and servers, implementing much of the business logic
- business logic is hard to express using just the constraints of relational DBMS
- often used as a model for Web based applications
- a variation of the MVC architecture?

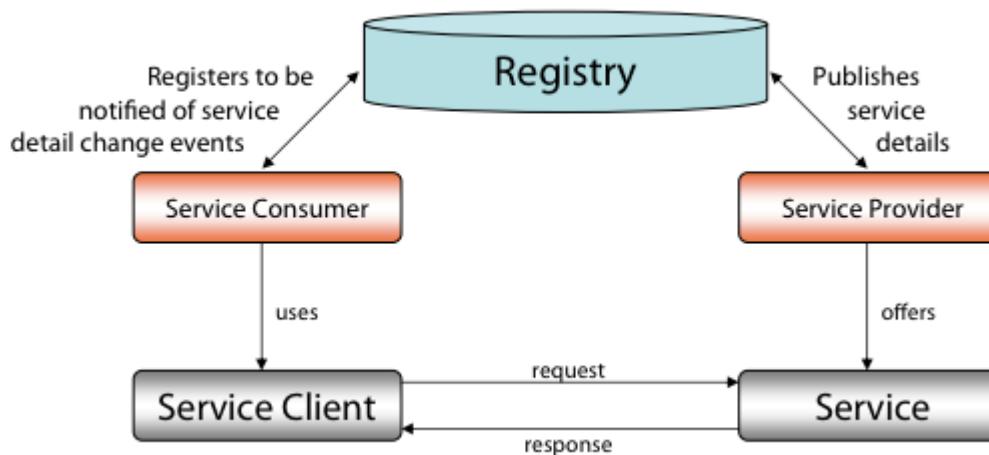


B. SOA- Service Oriented Architecture

- Modern applications are built from **software components** based on **interaction standards**:
 - Simple Object Access Protocol (SOAP), and
 - Java Platform Enterprise Edition (Java EE).
- **Each standard** defines:
 - specific ways in which components locate and communicate with one another.
 - a set of supporting system software to provide needed services (maintaining component directories, enforcing security requirements, and encoding/decoding messages).
- SOA is based on a classic **request/ response pattern**: a service consumer invokes a service provider over the network and waits until the provider has finished the operation.

SOA- Service Oriented Architecture

- SOA divides an application in two parts:
 - **A service coordinator**, representing the user functionality, and
 - **Service providers**, that implements the functionality.
- While **service coordinator** tends to be unique for a particular application, a **service** can be reused and shared by a multiple composite applications.
- The service coordinator explicitly specifies and invokes the required services.



SOA services

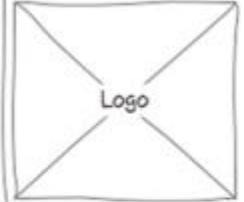
- **Services** are autonomous platform-independent entities that enable access to one or more capabilities, which are **accessible through provided interfaces**.
- A **new designed service** has to meet business requirements that are traditionally specified by a **Business Process Diagram (BPD)**.
- Transform the BPD (in BPMN, see figure) into **UML service diagrams**.
- 2 steps of transformation

iii. Design user interfaces and other system interfaces

HCI, windows, mockups, wireframes, forms,
reports

iii.A. Design user interfaces

- UI consists of all the hardware, software, screens, menus, functions, output, and features that affect two-way communications between the user and the computer.
- The user interface is the key to **usability**, which includes user satisfaction, support for business functions, and system effectiveness
- Dialog designs **begins with requirements**
 - Use case flow of activities
 - System sequence diagrams
 - Design adds screen layout, navigation, look and feel, user experience
- Now we require interface design for many **different environments and devices**:
 - Smart phones
 - tablets, iPads, etc
 - Tools that can quickly create screen mockups, referred to as **wireframes**



Logo

Pine Valley Furniture

Today:

Customer Information:

Customer Number:

Name:

Address:

City:

State:

Zip:

Customer Information Entry

Customer Information Today: 11-OCT-XX

CUSTOMER INFORMATION

Customer Number:

Name:

Address:

City:

State:

Zip:

Human-computer interaction (HCI)

- A user interface is based on basic principles of human-computer interaction.
- HCI describes the relationship between computers and the people who use them to perform their jobs,
- Evolution:
 - **Early UI:** typing complex commands on a keyboard, displayed as green text on a black screen.
 - **GUI (Apple):** used icons, graphical objects, and pointing devices (mouse).
 - **Today:** “almost transparent” UI that users don’t really notice, that translate user behavior, needs, and desires
- HCI has a major impact on user ***productivity***, it gets lots of attention and research efforts
- Ex: software **usability** has a major impact on the medical profession, physicians struggle with electronic health records (EHRs) systems

Seven basic principles of UI design (1)

1. Understand the Business

- understand the underlying business functions
- a good starting point might be to analyze a functional diagrams, use cases and detailed description of functionalities

2. Maximize Graphical Effectiveness

- Studies show that people learn better visually.
- In a graphical environment, a user can display and work with multiple windows on a single screen

3. Think like a User

- understand user experience, knowledge, and skill levels, provide flexibility based on their previous experience
- *Input processes* should be easy to follow, intuitive, and forgiving of errors.
- *Predesigned output* should be attractive and easy to understand, with an appropriate level of detail

Seven basic principles of UI design (2)

4. Use Models and Prototypes

- construct ***models and prototypes*** for user approval and obtain as much feedback as possible, as early as possible (interviews, questionnaires, by observation)
- ***usability metrics***, by using specialized software

5. Focus on Usability

- UI should include all tasks, commands, communications
- offer a reasonable number of choices
- strategy: present the most common choice as a default

6. Invite Feedback : monitor system usage and solicit user suggestions.

7. Document Everything

- all screen designs should be documented for later use
- screen designs should be numbered and saved in a hierarchy

Guidelines for User Interface Design

1. Create an Interface That Is Easy to Learn and Use

1. Focus on system design objectives, rather than calling attention to the interface.
2. Create a design that is easy to understand and remember. Maintain a common design in all modules of the interface, including the use of color, screen placements, fonts, and the overall “look and feel.”
3. Provide commands, actions, and system responses that are consistent and predictable.
4. Allow users to correct errors easily.
5. Clearly label all controls, buttons, and icons.
6. Select familiar images that users can understand, and provide on-screen instructions that are logical, concise, and clear.
7. Show all commands in a list of menu items, but dim any commands that are not available to the user.
8. Make it easy to navigate or return to any level in the menu structure



The figure shows four control buttons, but none of them has an obvious meaning.

Data overload! Please start over.

Data is not suitable.

Check the instructions for this task and the previous task!

Not enough data to proceed.

The data is incorrect.

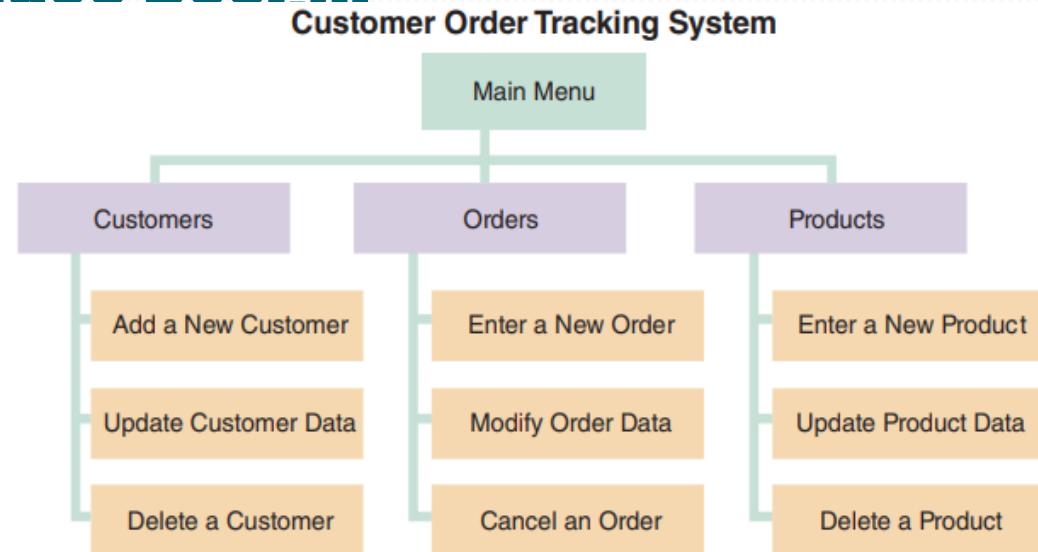
**Please use the proper format:
Three letters followed by three
numbers, with no spaces.
Example: CTS285**

The first five messages provide little or no information. The last message is the only one that is easy to understand.

Guidelines for User Interface Design

2 Enhance User Productivity

1. **Organize** tasks, commands, groups that resemble actual business operations (tree).
2. Create **alphabetical menu lists** or place the selections used frequently at the top of the menu list
3. Provide **shortcuts** for experienced users
4. Use **default values** if the majority of values in a field are the same
5. Provide a **fast-find feature** that displays a list of possible values while typing
6. Consider a **natural language feature** that allows users to type commands or requests in normal text phrases (NLP)



Guidelines for User Interface Design

3. Provide Users with Help and Feedback

1. Ensure that help is always available on demand: user-selected help and context-sensitive help.
2. Include contact information, such as a telephone extension or email address if a department or help desk is responsible for assisting users.
3. Require user confirmation before data deletion (Are you sure?) and provide a method of recovering data that is deleted inadvertently.
4. Provide an “Undo” key or a menu choice
5. When a user-entered command contains an error, highlight the erroneous part and allow the user to make the correction without retying the entire command.
6. Use hypertext links to assist users as they navigate help topics.
7. Display messages at a logical place on the screen, and be consistent.
8. Let the user know whether the task or operation was successful or not

Guidelines for User Interface Design

4. Create an Attractive Layout and Design

1. Use appropriate ***colors*** to highlight different areas of the screen;
2. Use ***special effects*** sparingly (animation, sounds)
3. Use ***hyperlinks*** that allow users to navigate to related topics.
4. Group related objects and information. Visualize the screen the way a user will see it, and simulate the tasks that the user will perform.
5. ***Screen density*** is important: uncluttered, readable design.
6. Display titles, messages, and instructions in a consistent manner and in the same general locations on all screens.
7. Use ***consistent terminology*** (ex: delete, cancel, and erase ?)
8. Ensure that commands always will have the ***same effect***
9. Patterns: red = **stop**, yellow = **caution**, and green = **go**

Guidelines for User Interface

5. Enhance the Interface

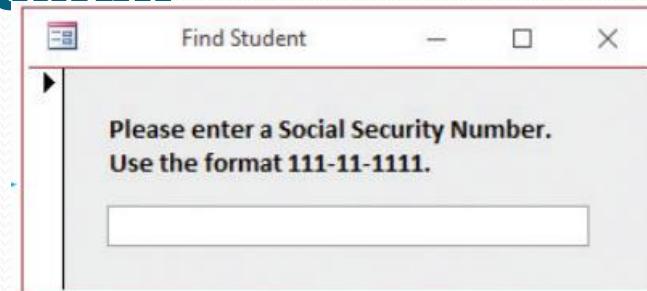


1. The starting point can be a **switchboard** with well-placed command buttons that allow users to navigate the system
2. Use a **command button** to initiate an action such as printing a form or requesting help
3. If variable input data is needed, provide a **dialog box** that explains what is required.
4. A **toggle button** makes it easy to show on or off status
5. Use **list boxes** that display the available choices.
6. Use an option button, or **radio button**, to control user choices.
7. If **check boxes** are used to select one or more choices from a group, show the choices with a checkmark or an X.
8. When dates must be entered, use a **calendar control**

Guidelines for User Interface Design

6. Focus on Data Entry Screens

1. Whenever possible, use a data entry method called **form filling**
2. Provide a way to leave the data entry screen at any time without entering the current record, such as a “***Cancel***” button
3. Provide a ***descriptive caption*** for every field
4. Design the screen form layout to match the layout of the source document
5. Display a ***sample format*** if a user must enter values (data), or it might be better to use an input mask
6. Use a default value when a field value will be constant for successive records
7. Display a list of acceptable values for fields



Guidelines for User Interface Design

7. Use Validation Rules

- one way to reduce input errors is to eliminate unnecessary data entry by using IDs, auto-fill
- correct errors before they enter the system by using data validation rules

No	Validation Rule	Explanations
1	sequence check	the data must be in some predetermined sequence
2	existence check	for mandatory data items
3	data type check	test to ensure that a data item fits the required data type
4	range check	verify that data items fall between a MIN and a MAX value
5	validity check	used for data items that must have certain values
6	combination check	performed on two or more fields to ensure that they are consistent
7	reasonableness check	values that are questionable but not necessarily wrong

Guidelines for User Interface Design

8. Manage Data Effectively

1. to reduce input errors, the system should enter and verify data as soon as possible
2. each data item should have a specific **type** (alphabetic, numeric, etc) and a **range** of acceptable values.
3. collect input data as **close to its source** as possible -ex: salespeople should use tablets to record orders
4. in an efficient design, data is entered only once, stored in a centralized storage area where all systems have access
5. **use codes.** Codes are shorter than the data they represent, and coded input can reduce data entry time.

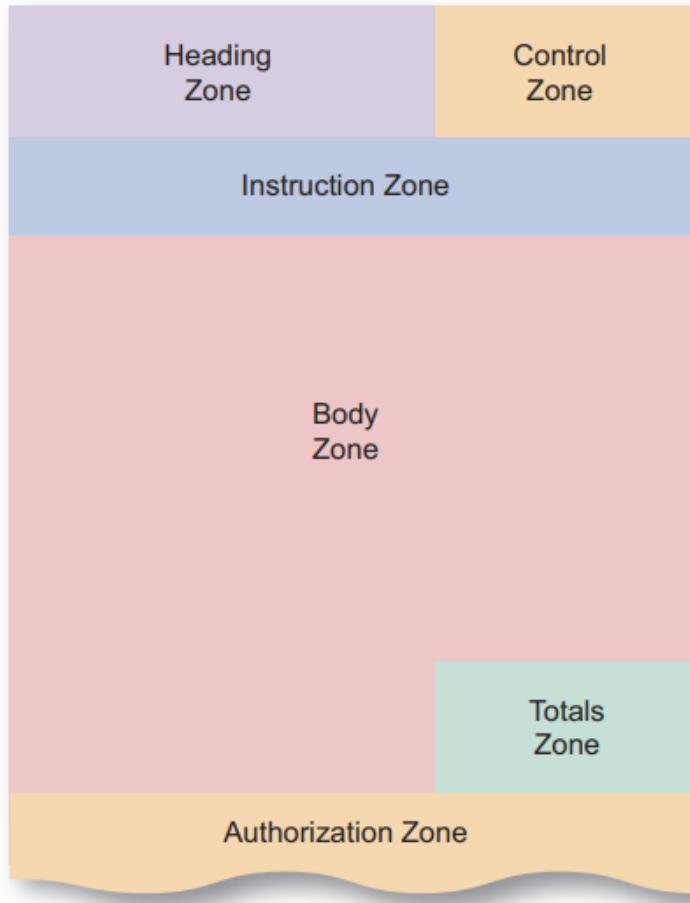
Report design

- few firms have been able to eliminate printed output totally
- designers use a variety of styles, fonts, and images to produce reports that are attractive and user-friendly.
- printed or viewed on-screen reports must be easy to read and well organized.
- there are many **report design tools**, including a Report Wizard. Many online web-based database systems also provide similar report design guidelines
- users should approve all report designs in advance, a sample report, called **a mock-up, or prototype**, should be prepared for users to review

Source document and form design

- the quality of the output is only as good as the quality of the input. The term **garbage in, garbage out (GIGO)**
- **source document (on-paper or online)** collects input data, triggers or authorizes an input action provides a record of the original transaction
- the analyst develops source documents that are easy to complete and use for data entry
- Good form layout makes the form easy to complete and provides **enough space**, both vertically and horizontally, for users to enter the data.
- A form should **indicate data entry** positions clearly using blank lines or boxes and descriptive captions.
- Also consider using **check boxes** whenever possible, so users can select choices easily. However, be sure to include an option for any input that does not match a specific check box.

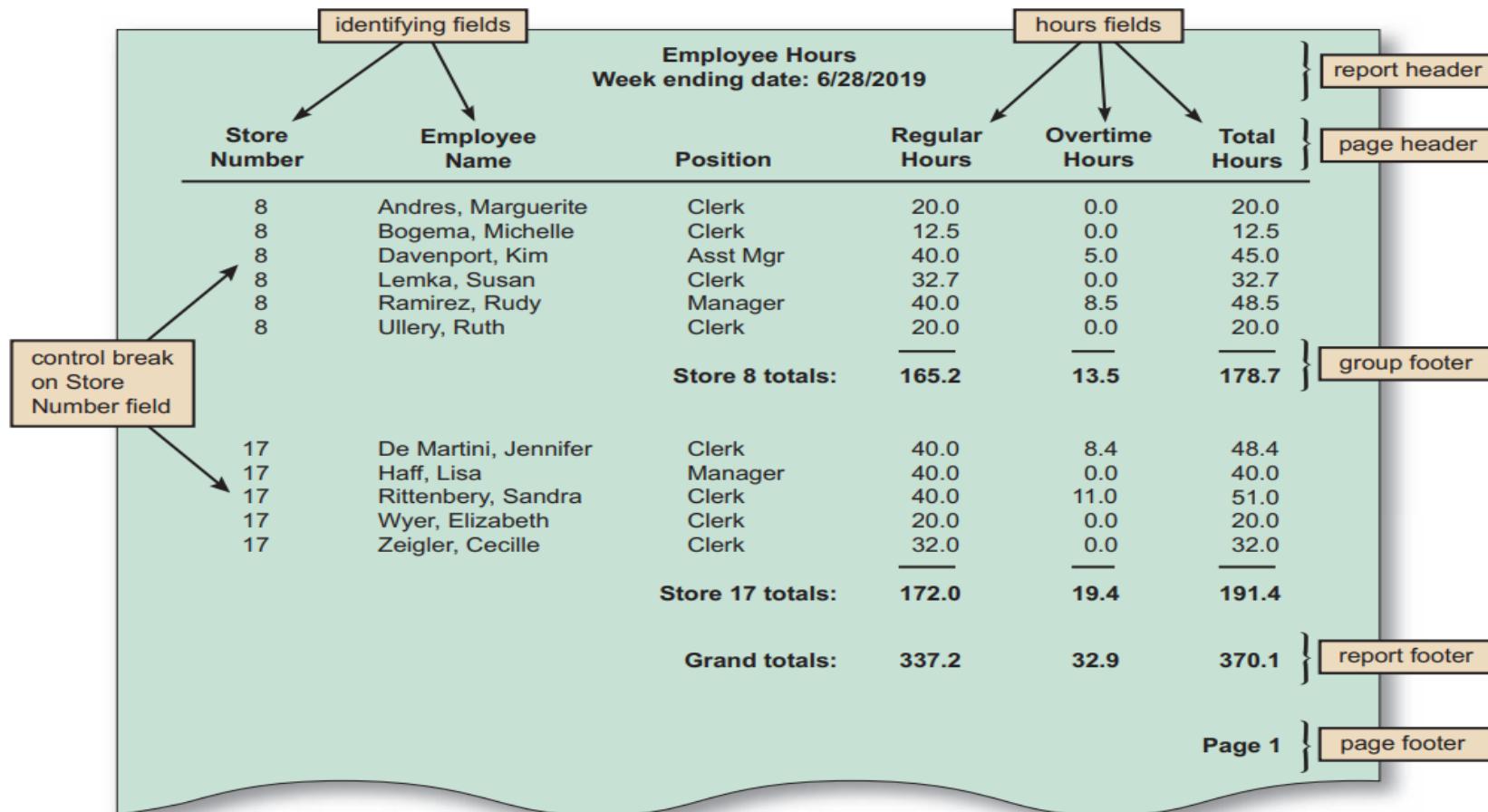
Source documents zones



- Information should flow on a form from left to right and top to bottom.
- Column headings should be short but descriptive, avoiding nonstandard abbreviations, with reasonable spacing between columns for better readability.
- The order and placement of printed fields should be logical, and totals should be identified clearly.

Report design principles

- the analyst must consider design features such as: report headers (report title, date, and other) and footers (grand totals), page headers and footers, column headings and alignment, column spacing, field order, and grouping of detail lines



Vague title

Difficult to read: information is packed too tightly

Pine Valley Furniture

CUSTOMER INFORMATION

CUSTOMER NO:	1273	
NAME:	CONTEMPORARY DESIGNS	
ADDRESS:	123 OAK ST.	
CITY-STATE-ZIP:	AUSTIN, TX 28384	
YTD-PURCHASE:	47,285.00	
CREDIT LIMIT:	10,000.00	
YTD-PAYMENTS:	42,656.65	
DISCOUNT %:	5.0	
PURCHASE:	21-JAN-XX	22,000.00
PAYMENT:	21-JAN-XX	13,000.00
PURCHASE:	03-MAR-XX	16,000.00
PAYMENT:	03-MAR-XX	15,500.00
PAYMENT:	23-MAY-XX	5,000.00
PURCHASE:	12-JUL-XX	9,285.00
PAYMENT:	12-JUL-XX	3,785.00
PAYMENT:	22-SEP-XX	5,371.65
STATUS:	ACTIVE	

No navigation information

No summary of account activity

Easy to read: clear, balanced layout

Clear title

Pine Valley Furniture

Detail Customer Account Information

Page: 2 of 2

Today: 11-OCT-XX

Customer Number: 1273

Name: Contemporary Designs

DATE	PURCHASE	PAYMENT	CURRENT BALANCE
01-Jan-XX			0.00
21-Jan-XX	(22,000.00)		(22,000.00)
21-Jan-XX		13,000.00	(9,000.00)
03-Mar-XX	(16,000.00)		(25,000.00)
03-Mar-XX		15,500.00	(9,500.00)
23-May-XX			5,000.00
12-Jul-XX	(9,285.00)		(13,785.00)
12-Jul-XX		3,785.00	(10,000.00)
22-Sep-XX		5,371.65	(4,628.35)
YTD-SUMMARY	(47,285.00)	42,656.65	(4,628.35)

Help

Prior Screen

Exit

Summary of account information

Clear navigation information

Types of reports

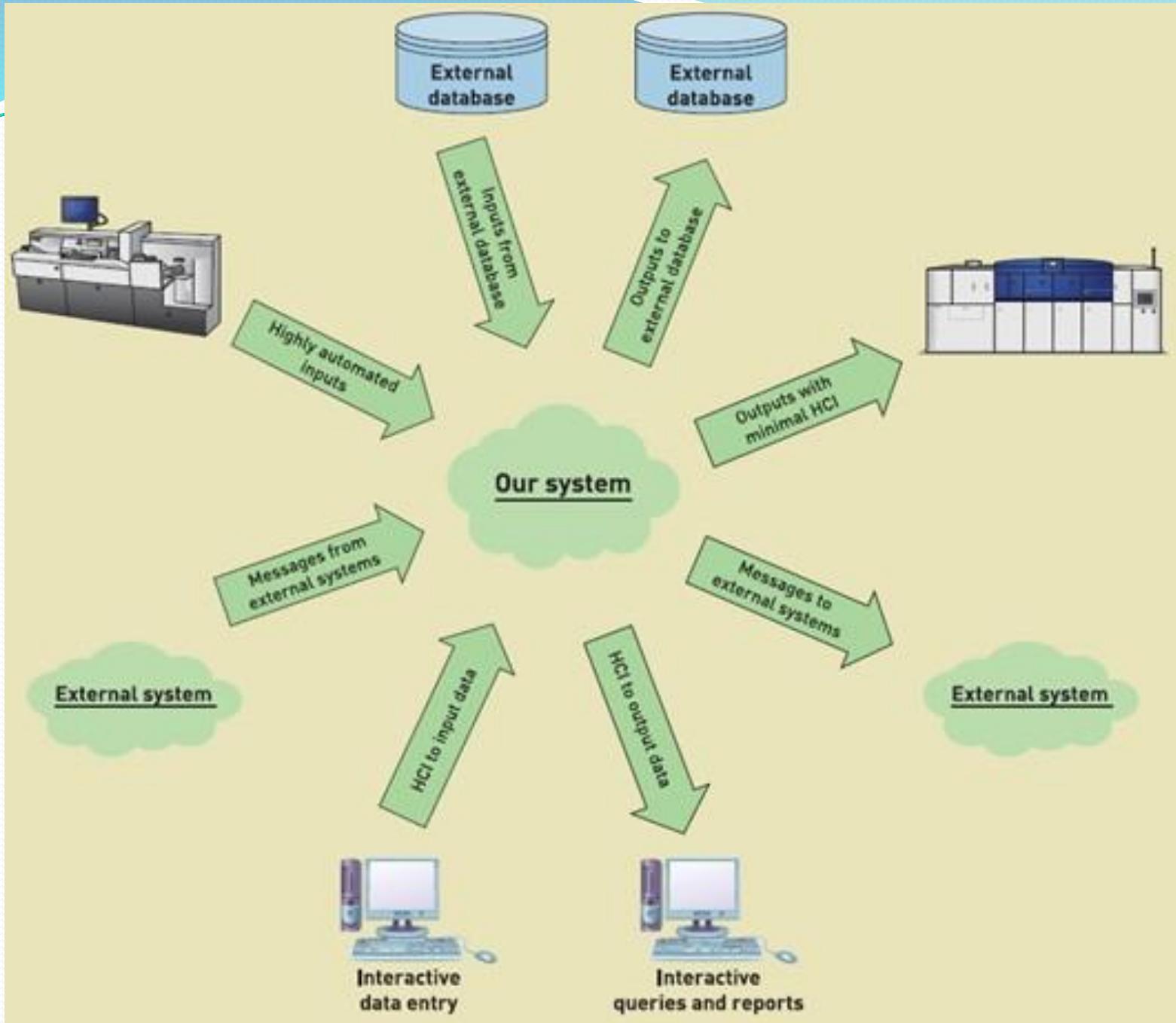
Report Name	Description
Scheduled Reports	Reports produced at predefined intervals—daily, weekly, or monthly—to support the routine informational needs of an organization.
Key-Indicator Reports	Reports that provide a summary of critical information on a recurring basis.
Exception Reports	Reports that highlight data that are out of the normal operating range.
Drill-Down Reports	Reports that provide details behind the summary values on a key-indicator or exception report.
Ad-hoc Reports	Unplanned information requests in which information is gathered to support a nonroutine decision.

iii.B. Design system interfaces

- Computer system interacts with many other systems, internal and external; integration?
- System interfaces can connect with other systems in many ways:
 - Save data another system uses;
 - Read data another system saved;
 - Real time request for information;
 - Software services.

Identifying system interfaces

- Inputs and outputs with minimal human intervention, **highly automated**
 - these are captured by **devices** (scanners, sensors etc) or generated by **persons** who start a process that proceeds without further human intervention
- Inputs **from** and outputs to other systems:
 - these are direct interfaces with other information systems, normally formatted as network messages
- Inputs and outputs **to** external database:
 - these can supply input or accept output from a system



iv. Database design

ERD (Entity relationship diagram)- logical model

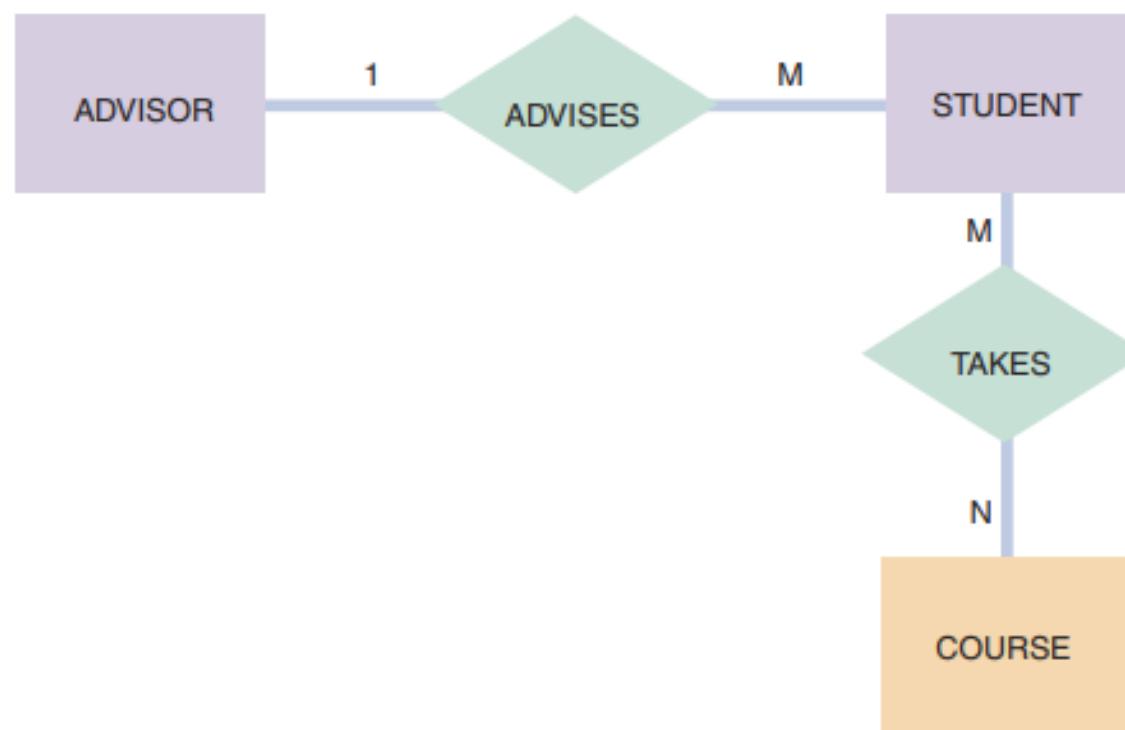
Physical model

iv. Database design

- Starting with the **domain class model** (or ERD)
- Choose **database structure**
 - Usually relational database
 - Could be ODBMS framework or other NoSQL systems
- Design **DB architecture** (distributed, etc)
- Design **database schema**
 - Tables and columns in relational, usually
- Design referential **integrity constraints**
 - Foreign key references/ other types of constraints

Logical design of relational databases

- Entity-relationship schema is a model that shows the logical relationships and interaction among system entities



Un-normalized STUDENT table

- STUDENT (STUDENT NUMBER, STUDENT NAME, TOTAL CREDITS, GPA, ADVISOR NUMBER, ADVISOR NAME, OFFICE, (COURSE NUMBER, CREDIT HOURS, GRADE))
- it contains **a repeating group** that represents the course each student has taken

STUDENT (Unnormalized)

STUDENT NUMBER	STUDENT NAME	TOTAL CREDITS	GPA	ADVISOR NUMBER	ADVISOR NAME	OFFICE	COURSE NUMBER	CREDIT HOURS	GRADE
1035	Linda	47	3.60	49	Smith	B212	CSC151	4	B
							MKT212	3	A
							ENG101	3	B
							CHM112	4	A
							BUS105	2	A
3397	Sam	29	3.00	49	Smith	B212	ENG101	3	A
							MKT212	3	C
							CSC151	4	B
4070	Kelly	14	2.90	23	Jones	C333	CSC151	4	B
							CHM112	4	C
							ENG101	3	C
							BUS105	2	C

Normalization process- 1NF

- A table is in first normal form (1NF) if it does not contain a **repeating group**
- STUDENT (STUDENT NUMBER, STUDENT NAME, TOTAL CREDITS, GPA, ADVISOR NUMBER, ADVISOR NAME, OFFICE, COURSE NUMBER, CREDIT HOURS, GRADE)

in 1NF, the primary key is a **unique** combination of a specific STUDENT NUMBER and a specific COURSE NUMBER

STUDENT in 1NF

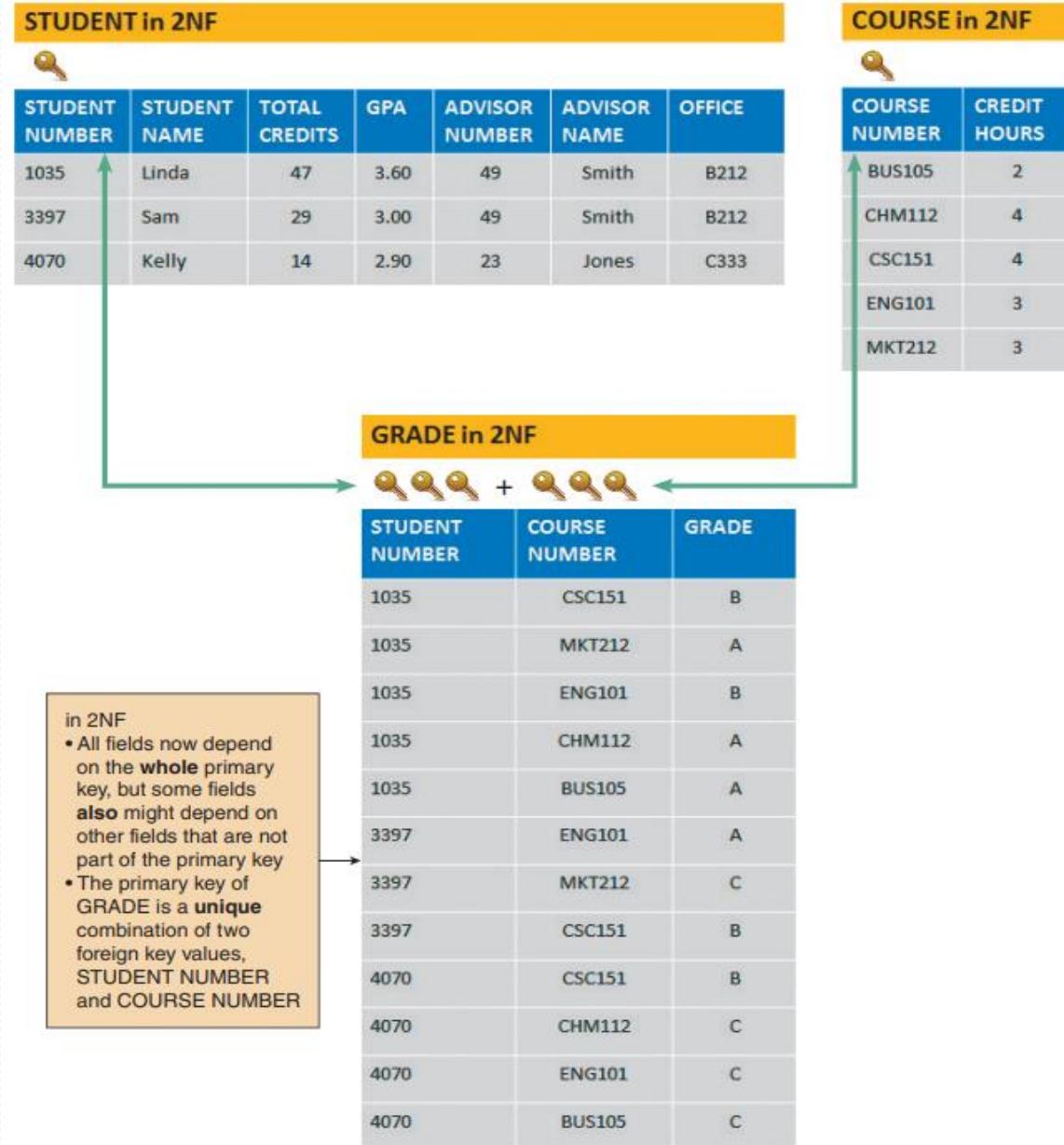
+
COURSE in 1NF

STUDENT NUMBER	STUDENT NAME	TOTAL CREDITS	GPA	ADVISOR NUMBER	ADVISOR NAME	OFFICE	COURSE NUMBER	CREDIT HOURS	GRADE
1035	Linda	47	3.60	49	Smith	B212	CSC151	4	B
1035	Linda	47	3.60	49	Smith	B212	MKT212	3	A
1035	Linda	47	3.60	49	Smith	B212	ENG101	3	B
1035	Linda	47	3.60	49	Smith	B212	CHM112	4	A
1035	Linda	47	3.60	49	Smith	B212	BUS105	2	A
3397	Sam	29	3.00	49	Smith	B212	ENG101	3	A
3397	Sam	29	3.00	49	Smith	B212	MKT212	3	C
3397	Sam	29	3.00	49	Smith	B212	CSC151	4	B
4070	Kelly	14	2.90	23	Jones	C333	CSC151	4	B
4070	Kelly	14	2.90	23	Jones	C333	CHM112	4	C
4070	Kelly	14	2.90	23	Jones	C333	ENG101	3	C
4070	Kelly	14	2.90	23	Jones	C333	BUS105	2	C

in 1NF
• There are no repeating groups
• The primary key is a **unique** combination of two foreign key values: STUDENT NUMBER and COURSE NUMBER
• All fields depend on the primary key, but some fields do not depend on the **whole** key — only part of it

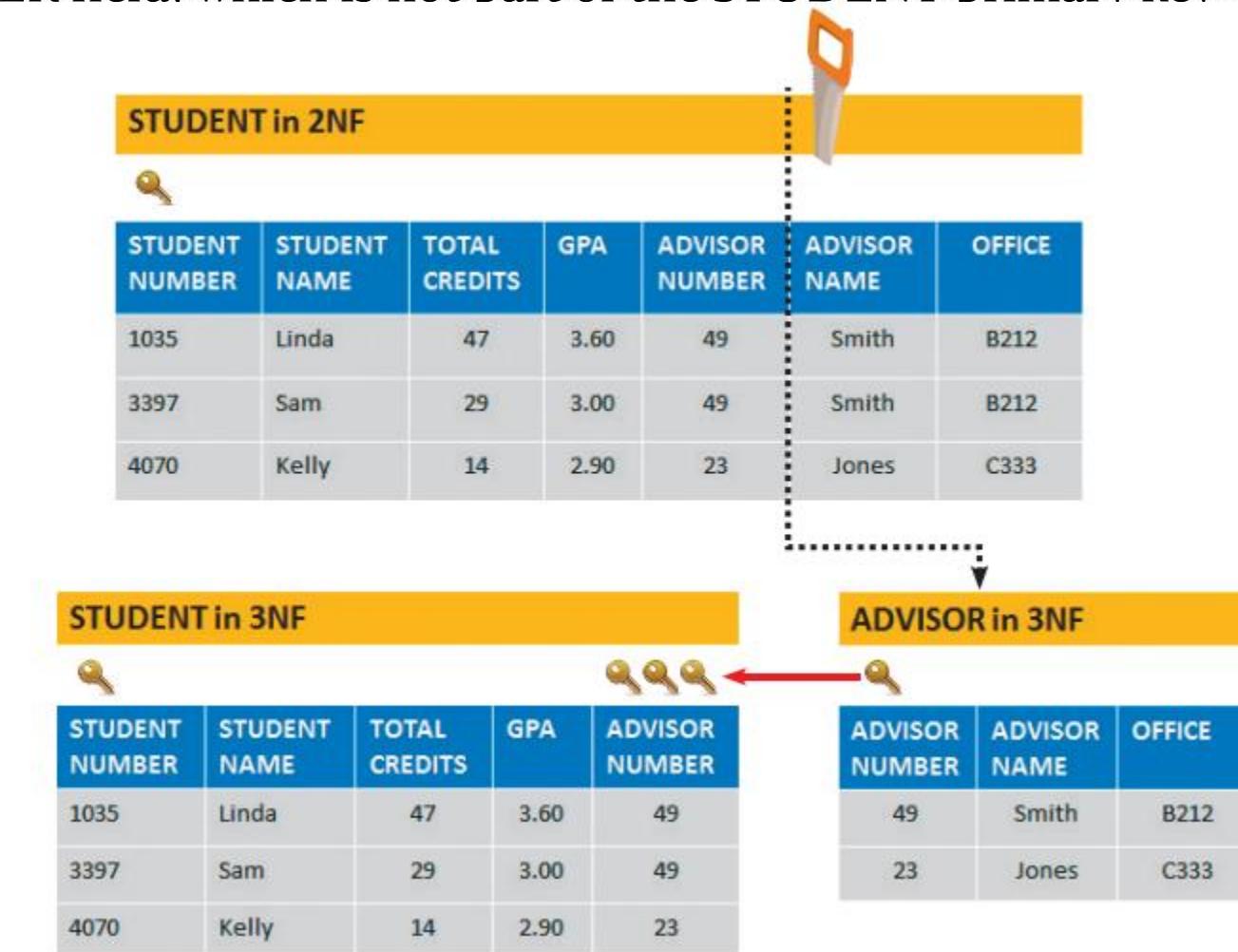
Normalization process- 2NF

- A table design is in second normal form (2NF) if it is in 1NF and if all fields that are not part of the primary key are **functionally dependent** on the **entire primary key**
- STUDENT (STUDENT NUMBER, STUDENT NAME, TOTAL CREDITS, GPA, ADVISOR NUMBER, ADVISOR NAME, OFFICE)**
- COURSE (COURSE NUMBER, CREDIT HOURS)**
- GRADE (STUDENT NUMBER, COURSE NUMBER, GRADE)**

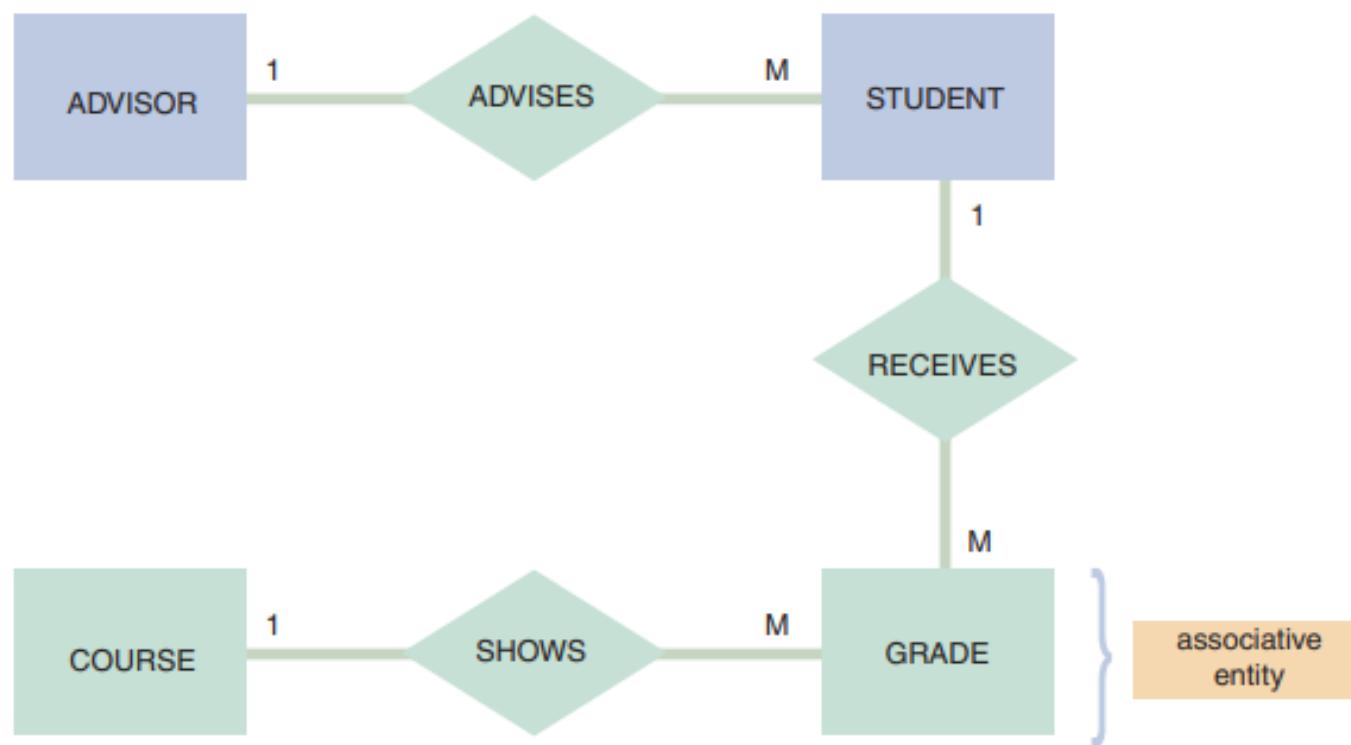


Normalization process- 3NF

- a design is in 3NF if **every nonkey field** depends on the key, **the whole key**, and nothing but the key
- ADVISOR NAME and OFFICE fields depend on the ADVISOR NUMBER field, which is not part of the STUDENT primary key



Normalized DB ERD



E-R Diagrams to Relational Transformation

E-R Structure	Relational Representation
Regular entity	Create a relation with primary key and nonkey attributes.
Weak entity	Create a relation with a composite primary key (which includes the primary key of the entity on which this weak entity depends) and nonkey attributes.
Binary or unary 1:1 relationship	Place the primary key of either entity in the relation for the other entity or do this for both entities.
Binary 1:N relationship	Place the primary key of the entity on the one side of the relationship as a foreign key in the relation for the entity on the many side.
Binary or unary M:N relationship or associative entity	Create a relation with a composite primary key using the primary keys of the related entities, plus any nonkey attributes associative entity of the relationship or associative entity.
Binary or unary M:N relationship or associative entity with additional key(s)	Create a relation with a composite primary key using the primary keys of the related entities and additional primary key attributes associated with the relationship or associative entity, plus any nonkey attributes of the relationship or associative entity.
Binary or unary M:N relationship or associative entity with its own key	Create a relation with the primary key associated with the relationship or associative entity, plus any nonkey attributes of the relationship or associative entity and the primary keys of the related entities (as foreign key attributes).
Supertype/subtype	Create a relation for the supertype, which contains the primary relationship key and all nonkey attributes in common with all subclasses, plus create a separate relation for each subclass with the same primary key (with the same or local name) but with only the nonkey attributes related to that subclass.

Physical design of DB

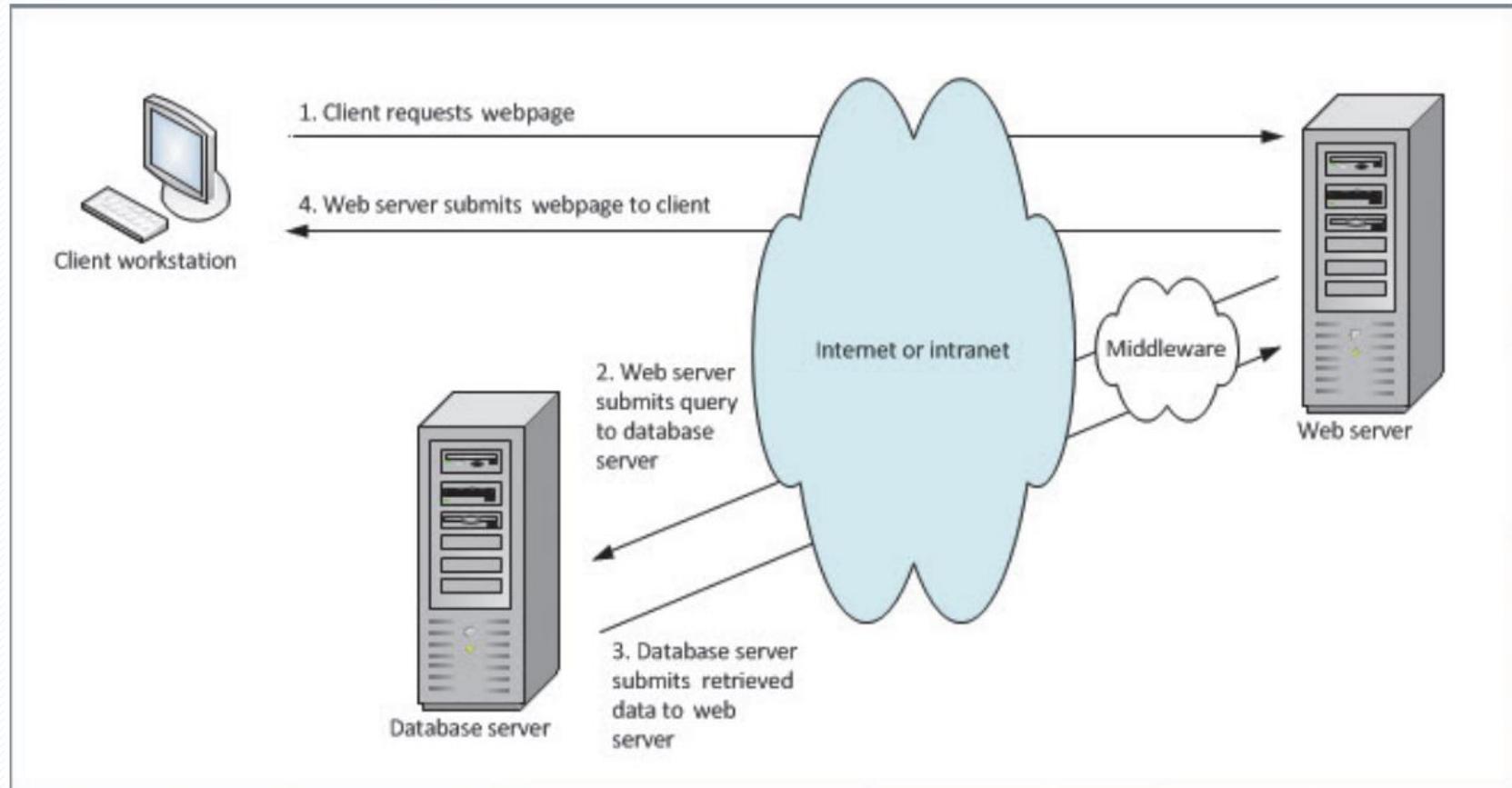
- **Physical storage** is strictly hardware related because it involves the process of reading and writing binary data to physical media
- Includes:
 - design of physical fields: datatype, length, calculated fields
 - controlling data integrity: default value, null values, range control, referential integrity
 - physical table may or may not correspond to one relation, objectives : efficient use of secondary storage and data processing speed
 - Denormalization
 - Partitioning

Physical design of DB

- The physical repository might be ***centralized***, or it might be ***distributed*** at several locations.
- The stored data might be managed by ***a single DBMS***, or ***several systems***.
- For ***database connectivity and access problems***
 - **Open database connectivity (ODBC)** - industry-standard protocol that makes it possible for software from different vendors to interact and exchange data (uses SQL statements)
 - **Java database connectivity (JDBC)** a common standard that enables Java applications to exchange data with any database that uses SQL statements and is JDBC-compliant.

DB in Web based designs

- An Internet connection is necessary to access DB
- **Middleware:** a software that integrates different applications and allows them to exchange data.



Data Control

- file and database control must include all measures necessary to ensure that data storage is correct, complete, and secure.
- **Permissions** can be associated with different users
- **Encryption** to prevent unauthorized access to the data
- **Backup copies** must be retained for a specified period of time. In the event of a file catastrophe, **recovery procedures** can be used to restore the file or database to its current state at the time of the last backup.
- **Audit log files**, which record details of all accesses and changes to the file or database

v. Design security and controls

v. Design security and controls

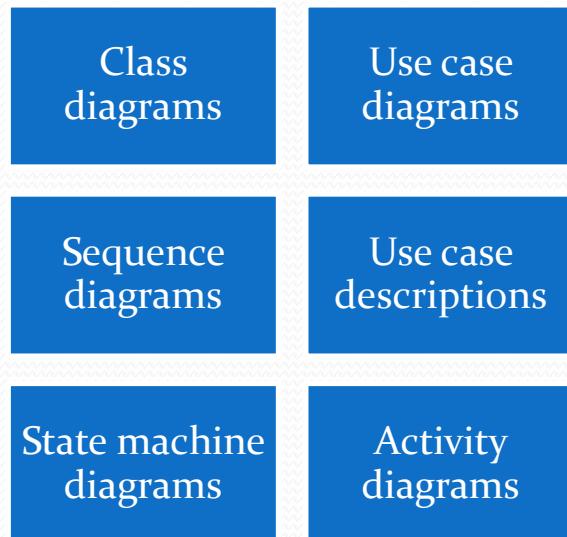
- Protect the organization assets
- Becomes crucial in Internet and wireless environment
- We need:
 - User interface controls;
 - Application controls;
 - Database controls;
 - Network controls.

. Design security and controls(2)

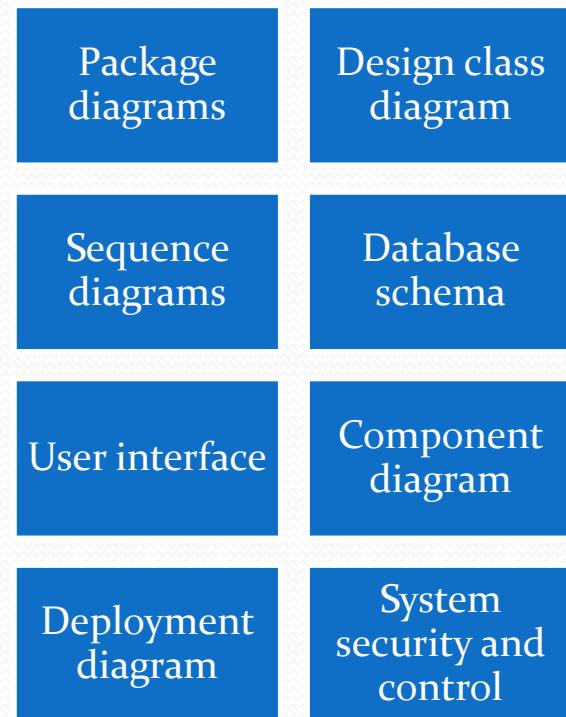
- **Security and controls** - should be included in all other design activities: user interface, system interface, application architecture, database, and network design.
- **User-interface controls** limit access to the system to authorized users.
- **System-interface controls** ensure that other systems cause no harm to this system.
- **Application controls** ensure that transactions are recorded precisely and that other work done by the system is done correctly.
- **Database controls** ensure that data is protected from unauthorized access and from accidental loss due to software or hardware failure.
- **Network controls** ensure that communication through networks is protected.

Analysis models and Design models

ANALYSIS



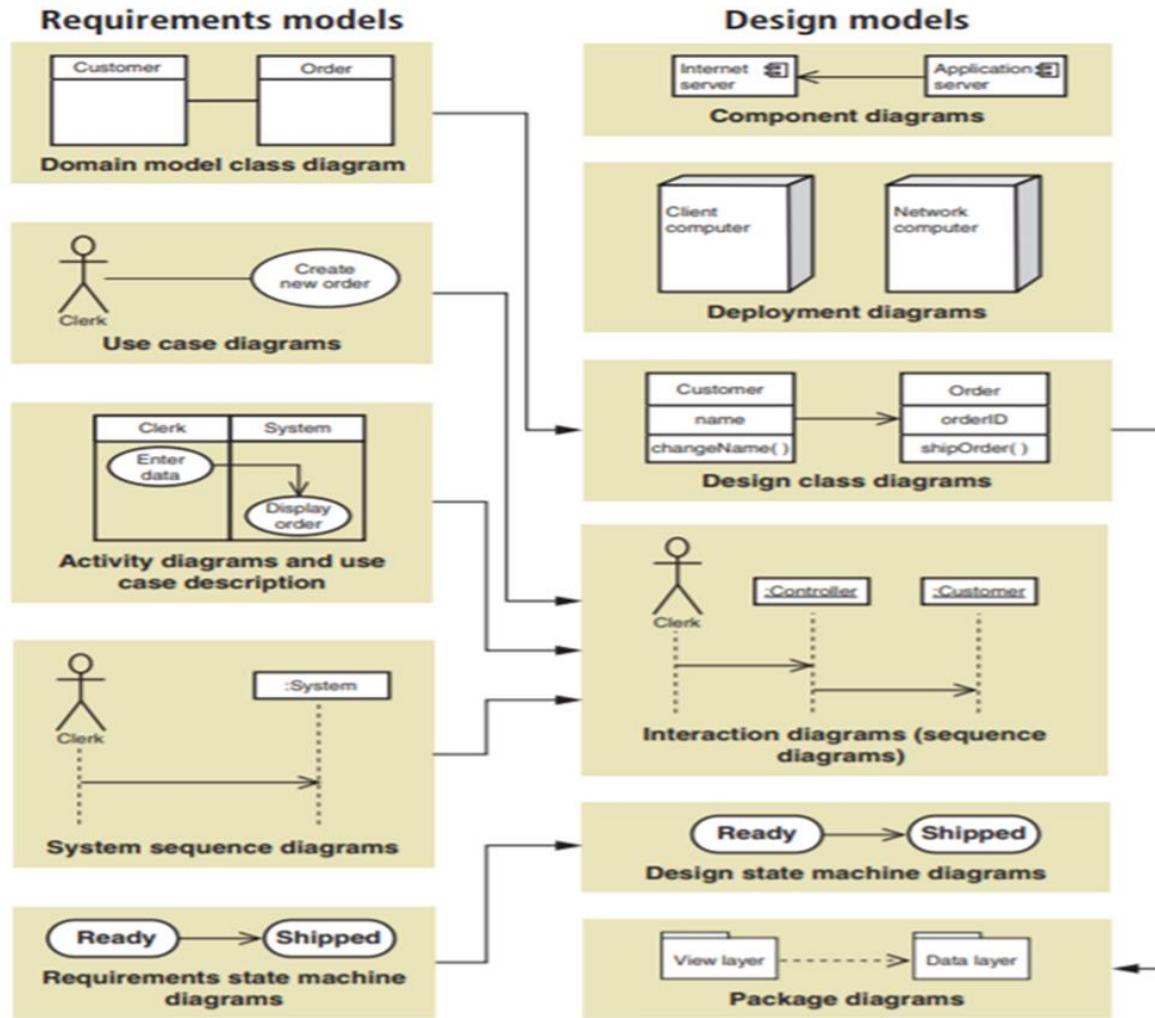
DESIGN



10th Lecture: Detailed system design

- Re-iterate and improve Analysis model.
- Analyze dependencies between diagrams.
- Build a design model of a computer system, considering various architecture types.

Relations between UML diagrams



Modeling purpose

- After system analysis, we proceed with system modeling (or design) in order to:
 - Facilitate the **correct understanding** of the system;
 - Remove information and operation **redundancy**;
 - **Reduce the run time** for various queries and updates of database;
 - **Reduce** the customer **waiting time**.

Realization of the use cases

- **Design and implementation of use cases**

- **Sequence diagram**– it is expanded starting from the system sequence diagram, adding a controller and domain classes;
- **Design class diagram**– it is expanded starting from the domain classes model and updated based on sequence diagrams
 - Messages (calls) that are received by an object should become methods of that class
- **Class definitions**– they are written in the programming language chosen for implementation of design and controller classes.
- **User interface classes**– forms or pages added to manage user interface between actors and controller classes.
- **DB access classes**– they are added to manage requests for querying or saving data into the DB

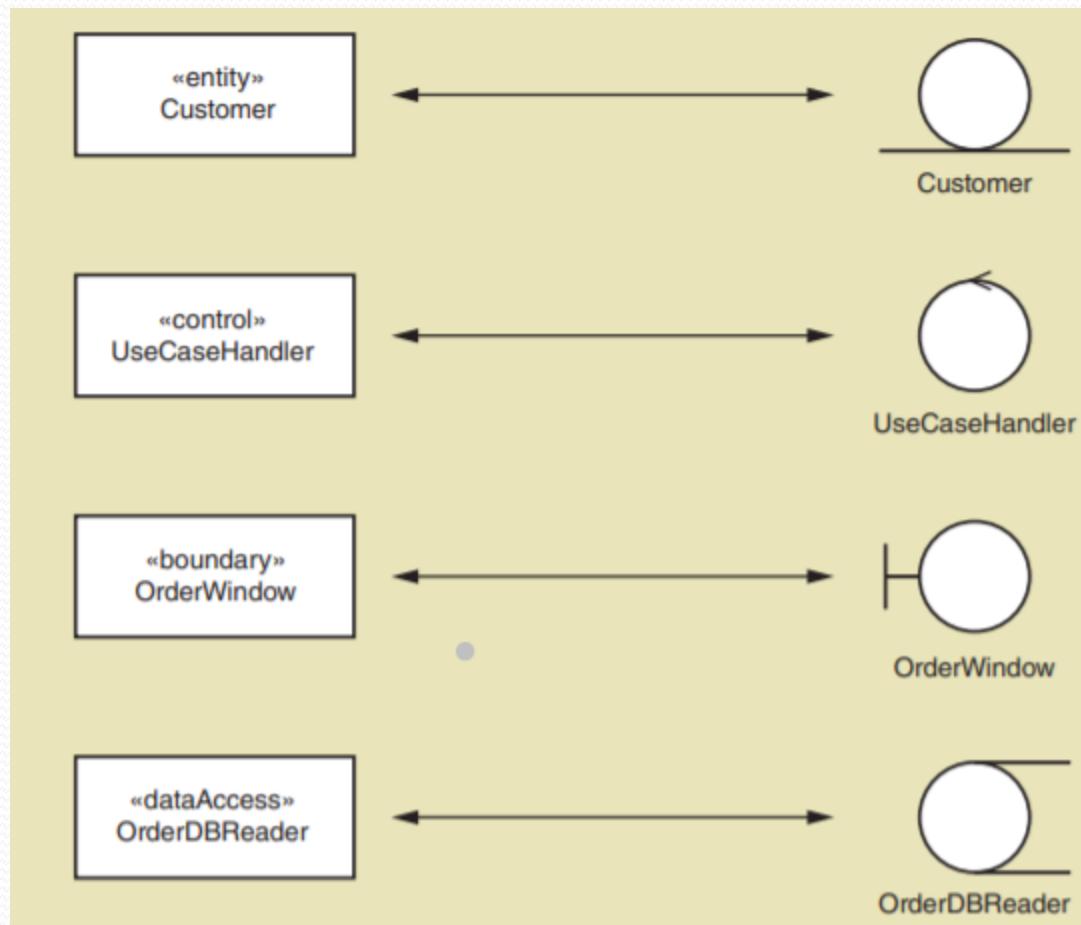
- **Detailed design of the static model**

- A detailed version of the class diagram is designed, including **navigation visibility** for associations.
- Based on the sequence diagrams, the signature of the class methods is completed
- The solution is partitioned in packages (grouping elements), as deemed necessary

1. Design class diagram - stereotypes

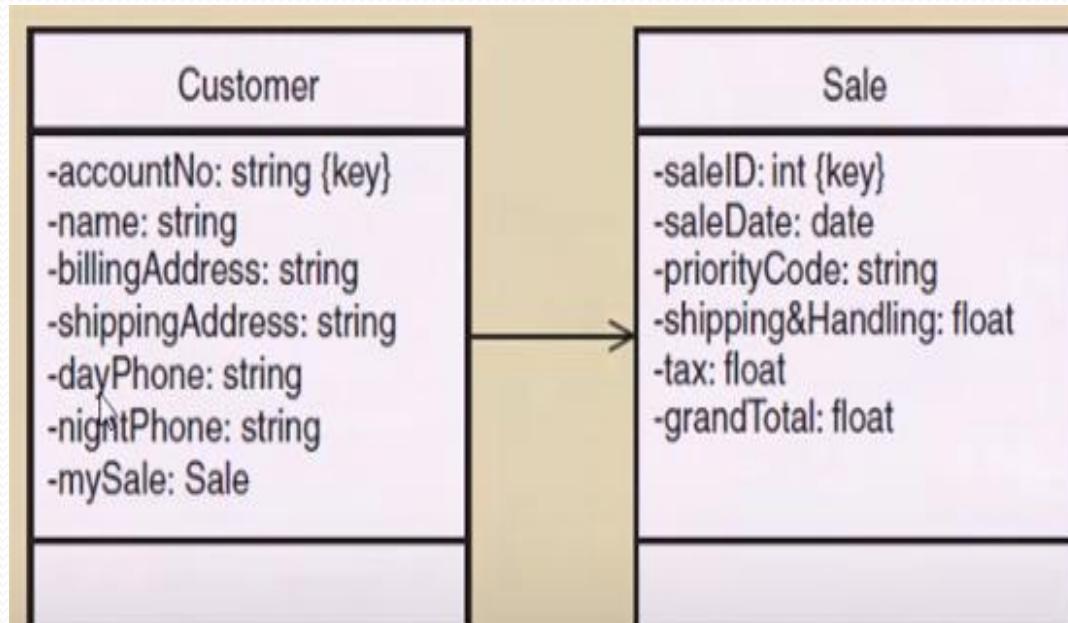
- **Stereotypes** are added for classes
 1. **Persistent class**- a class whose objects have to exist after the system is turned off
 2. **Entity class**- an identifier for a problem domain class.
 3. **Boundary / view class** – a class that is situated at the automated boundary of a system, such as in input form or a Web page
 4. **Control class** – a class that mediates between boundary and entity classes, acting as a control panel between user level (visualization) and business level
 5. **Data access class**- a class that is used to receive or send data from/to a database

1. Design class diagram - stereotypes



1. Design class diagram: navigation visibility

- It is an object ability to view and interact with another object
- It is implemented by adding into a class an object reference variable;
- It is symbolized as an arrow on one of the association ends – Client can view and interact with Sale

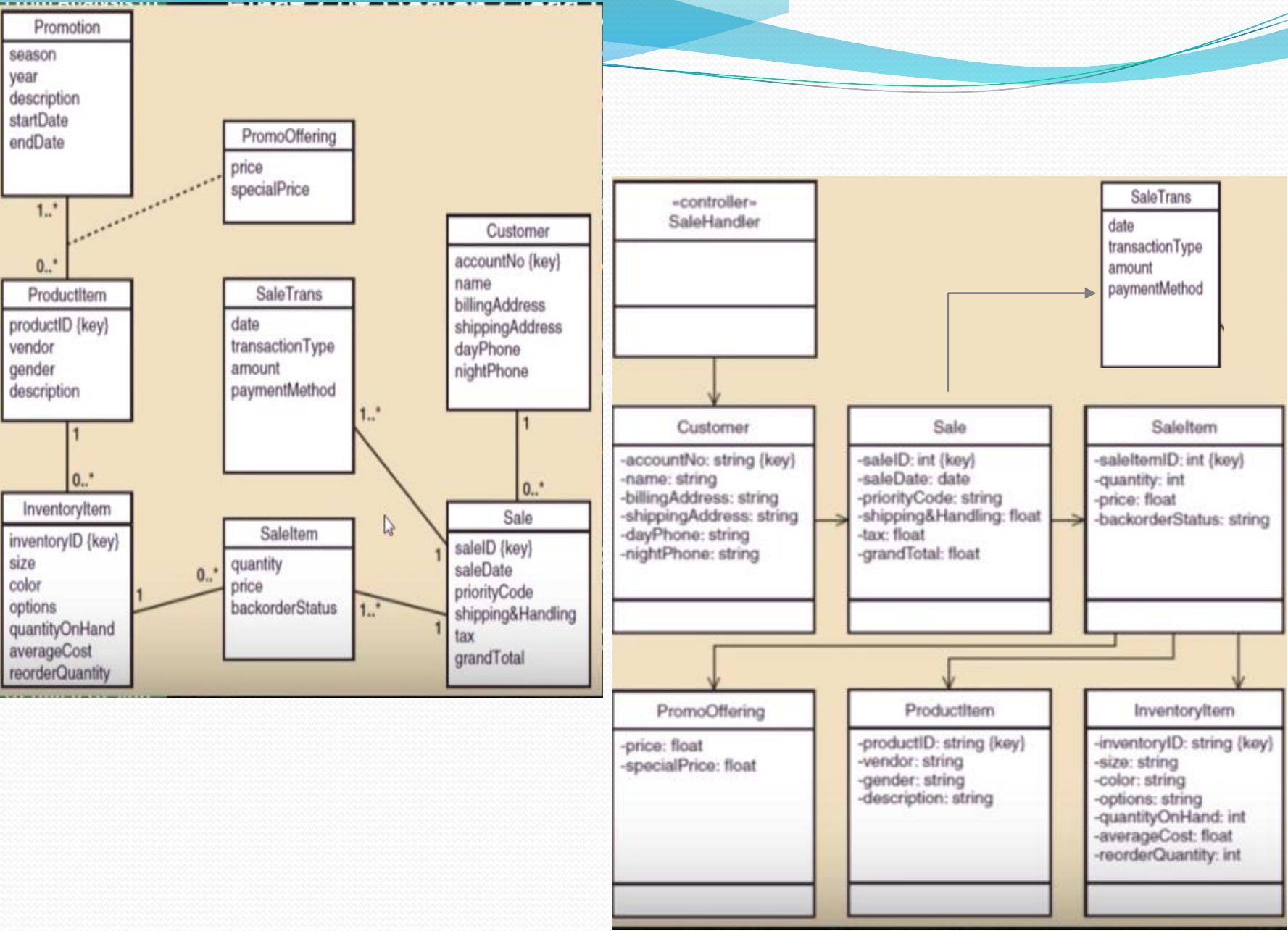


1. Design class diagram: navigability rules

1. **One-many association** that indicates a superior-subordinate relationship ensures navigation from superior to subordinates
2. **Mandatory associations** in which objects of a class cannot exist without objects of another class, usually provide navigation from the most independent class to the most dependent one.
3. When an **object needs information from another object**, a navigable association might be necessary

1. Design class diagram: Detailed Class Diagram

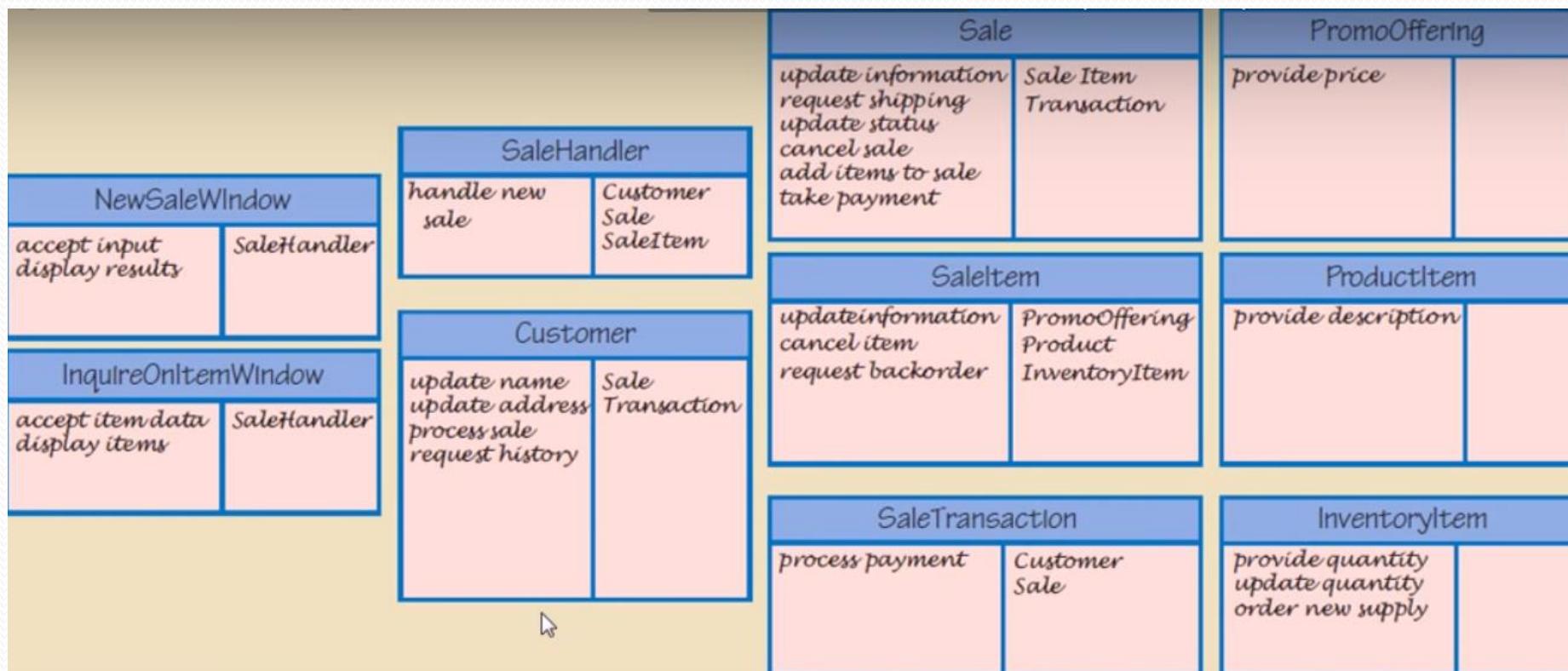
- Go through the **use cases**
- Select the **domain classes** that are involved in each use case; Check the *preconditions* and *postconditions* for use case completion.
- Add a **controller class** to be responsible for the use case
- Determine the requirements for **navigation visibility**
- Fill in the **attributes** of each class with *visibility* and *type*
- **Note:** Many times multiplicities are removed from the diagram to focus on navigation, but they can be retained

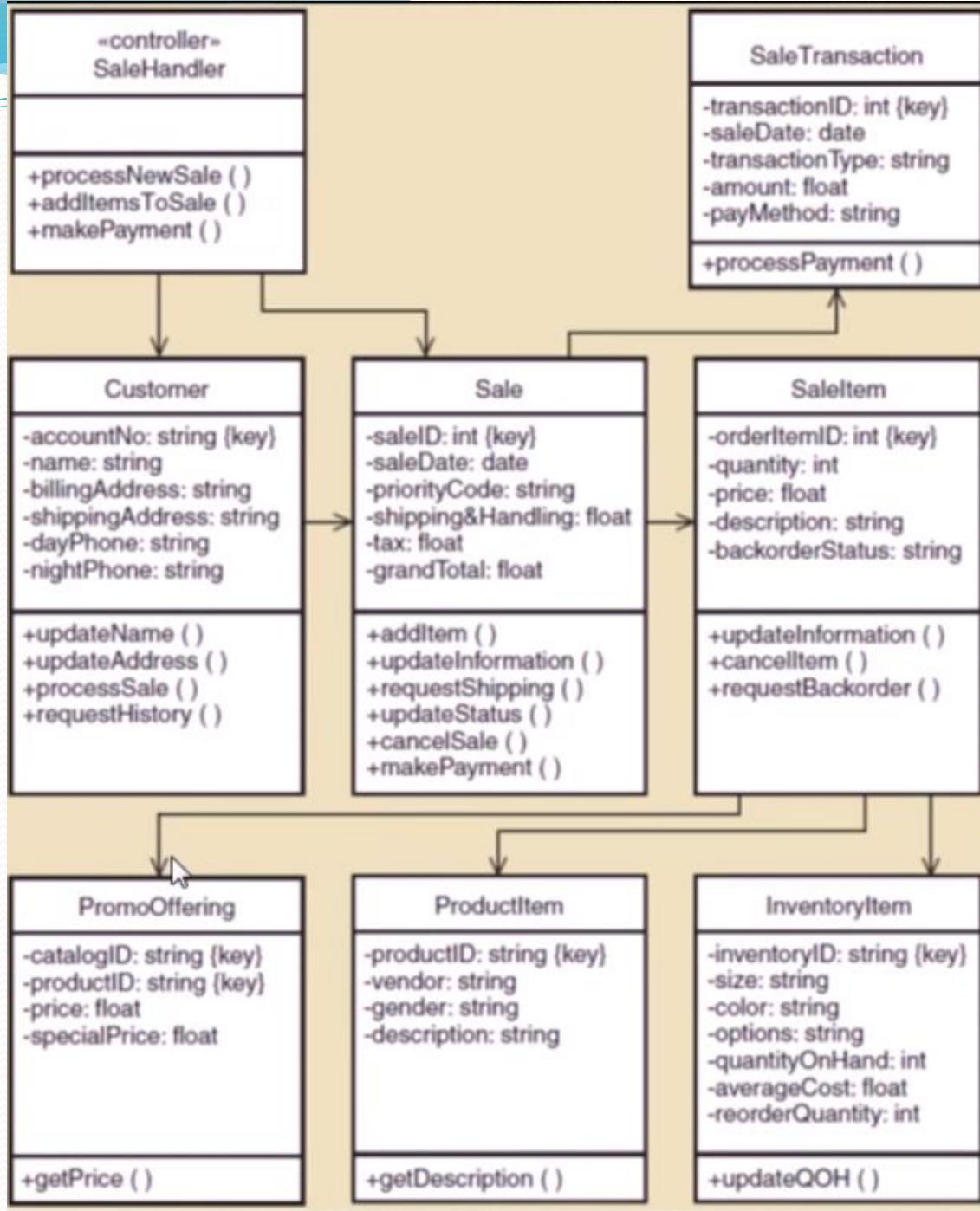


1. Design class diagram: Class Methods

- You can use the **CRC - Class, Responsibility, Collaboration** cards technique
 - What are the responsibilities of a class and how it collaborates with other classes to realize the use case
- It is obtained through brainstorming
- You can use **detailed sequence diagrams** - each message received by an object of a class must have a corresponding method in that class

CRC examples



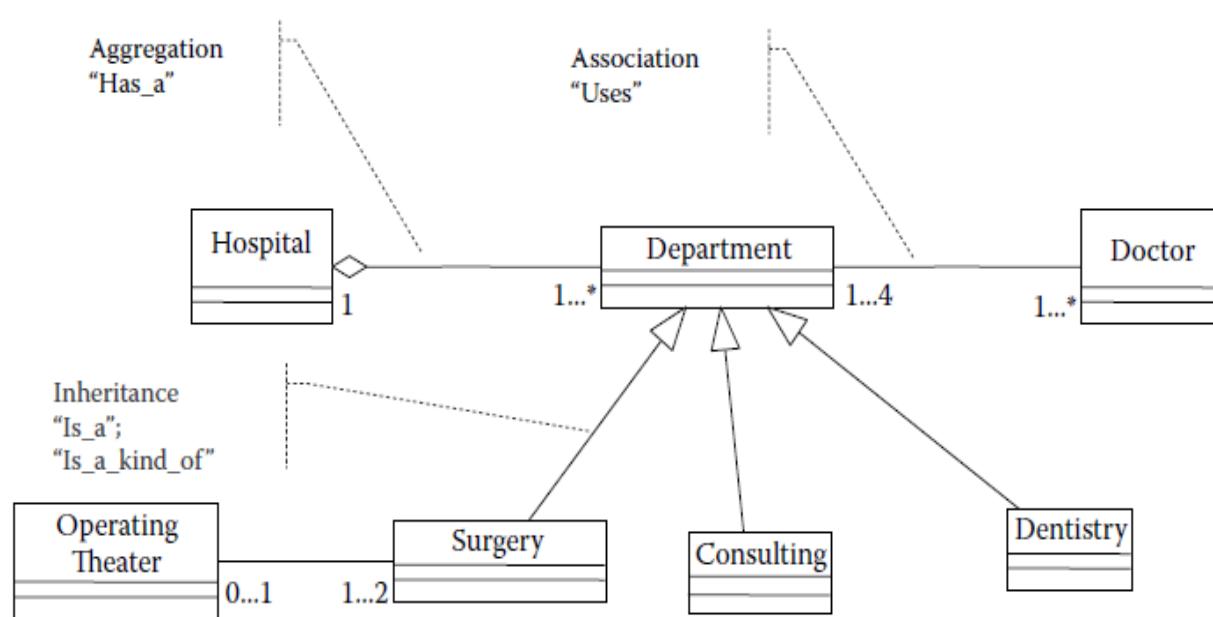


1. Design class diagram: Protection against change

- A design principle is to separate the parts that are stable from the parts that undergo numerous changes.
- Separate **forms** and **pages in the user interface** that have a high probability of changing from the logic of the application.
- The **connection to the database** and **SQL logic** that are likely to change is kept in **separate classes** from the application logic
- Use **adapter classes** that can change for interaction with other systems
- If you choose between two design variants, choose one that offers greater protection against change

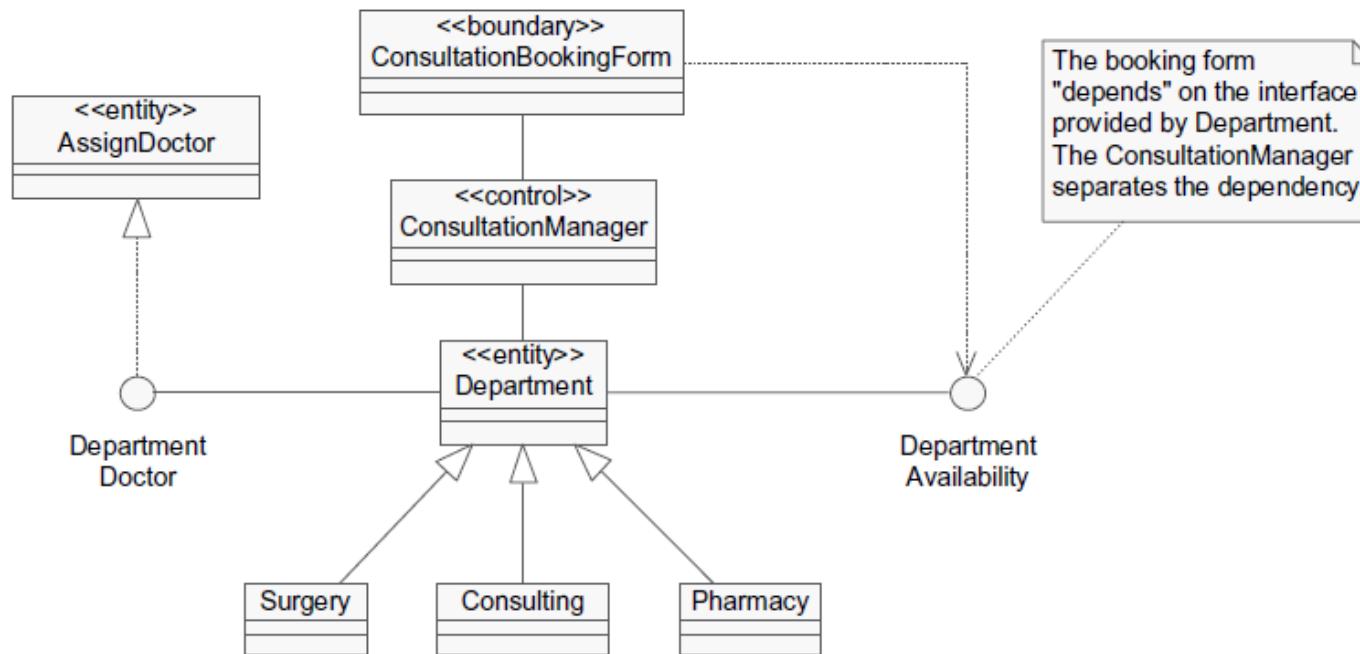
1. Design class diagram: Advanced relationships

- Consider the following example:



Advanced relationships in design class diagram

- The Department class two interfaces: **DepartmentDoctor** and **DepartmentAvailability**. Each interface represents a subset of the operations of the **Department** class. They do not implement the operations.



Association relationships in design

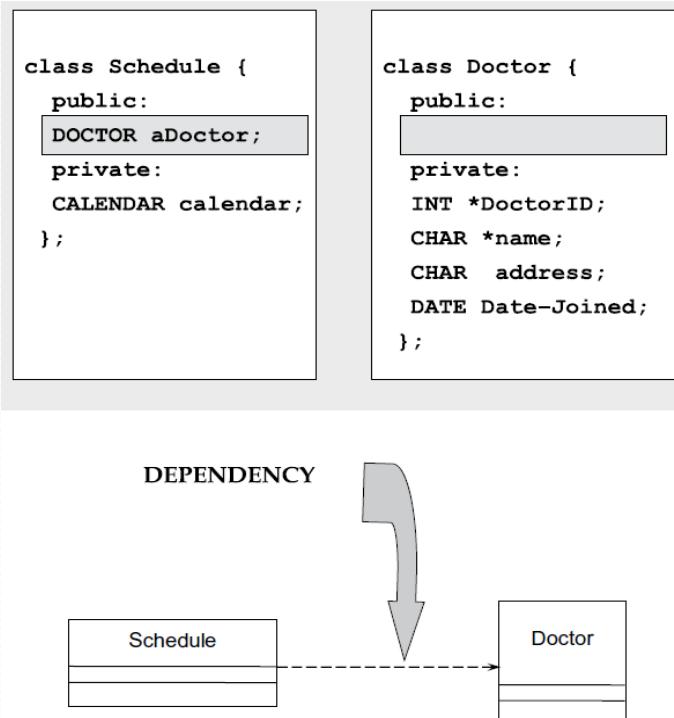
- represented by a straight line, indicates a bidirectional connection between classes. Associations get implemented through **attributes** in class definitions.
- For example: **Department-Doctor** (The **Department** declares *CHAR *DeptName*. **Department** also has an operation *+getDeptName()*). Operation arguments and return classes can also denote an association.

```
class Department {  
private:  
    INT *DeptID;  
    CHAR *DeptName;  
public:  
    DOCTOR *aDoctor;  
    aDoctor.getDocID();  
  
    +getDeptName() {  
        -code to get DeptName -};  
};
```

```
class Doctor {  
public:  
    DEPARTMENT *aDept;  
    aDept.getDeptName();  
    +getDocID() {---  
        -code to get DocID --};  
private:  
    INT *DoctorID;  
    CHAR *Name;  
    CHAR Address;  
    DATE Date-Joined;  
};
```

Dependency relationships in design

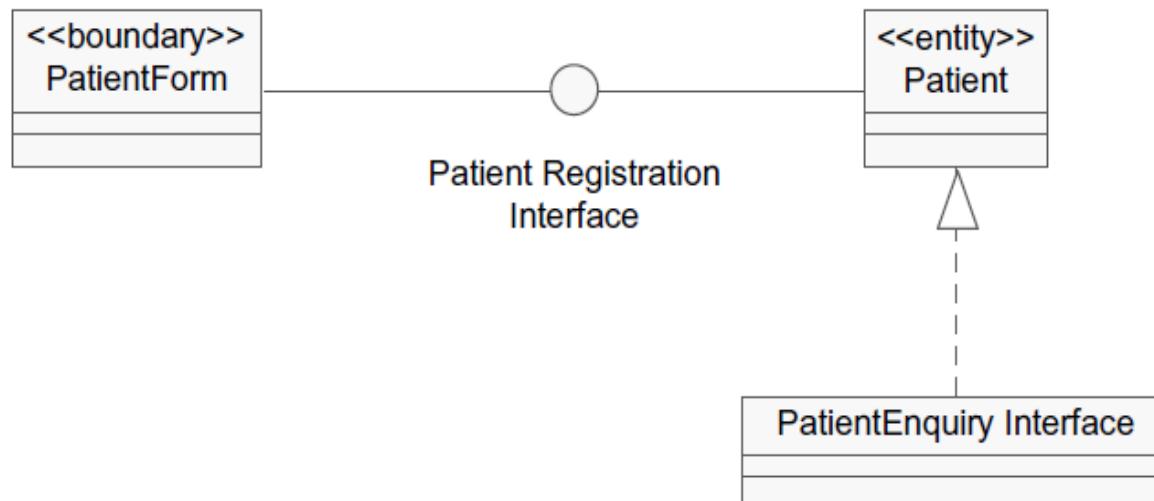
- implies that the objects of a class (called the client class) depend on the objects of another class (called the supplier class). It is drawn as a dashed arrow.
- In figure, the class **Schedule** is shown as being dependent on the class **Doctor**.



- Objects of the client class change their state as a result of changes in the state of the supplier class.
- Operations of the client class create objects of the supplier class.
- The operations of the client class have signatures whose return class or arguments are instances of (or references to) the supplier class.

Interface and realization relationships

- a complex class provides a small part of its public operations as a “subset” for use by other classes -an “**interface**.” This functionality can then be called upon by the consumer classes to seek the services of the provider.
- not all of these **Patient** operations are important when it comes to **PatientForm**. To achieve this, **Patient** provides an interface called **PatientRegistrationInterface**, which can be used by **PatientForm**, versus the entire **Patient** class. Interfaces are also classes, albeit without the details of the operation implementation and without all the attributes.



Aggregation relationships in design

- a whole–part relationship wherein an entity is “made up of” or “composed of” other entities, or classes.
- Hospital *is made up* of departments. Technically, in the solution space, this aggregation means the Hospital object is composed of Department objects.
- Two aggregation types:
 - **Shared aggregation** - the “main” (or higher) class is made up of objects from the “other” class, it can still exist independently during execution.
 - **Composed aggregation** (composition) - the main class is made up entirely of objects in the other class and cannot exist independently.

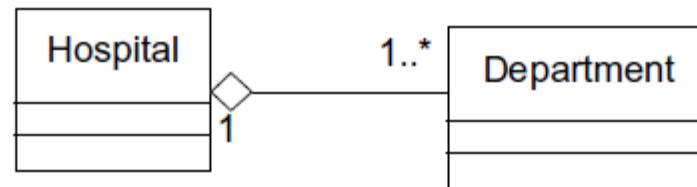
Implementing the relationships: by reference

- **Shared aggregation:** the Hospital object points to or references the Department. As a result, the Department object is unique in the system. Other objects that need the Department object reference it rather than make a copy of it.

“By Reference” implies only a reference (or pointer) to the object (e.g., Department). Therefore, it is possible for other objects to point to the same Doctor object.

Note: Association is also implemented “By Reference”

By reference

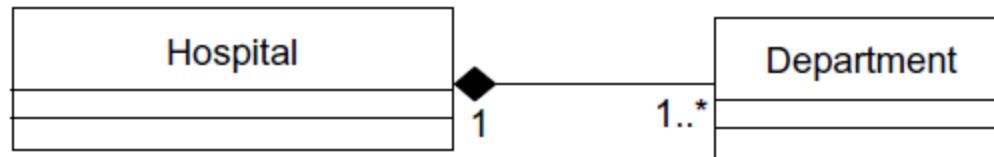


Implementing the relationships: by value

- **Composition:** the Hospital object can use a copy of the Department object. the original object continues to exist as is. Potential for confusion as various objects that try to use Department try and update their own copies of Department.

In implementing “by value,” a copy of the aggregate parts (e.g., Department) is made. The “contained” Department is not shared by other objects in the system

By Value



Java implementation for aggregation

```
class Department
{
    int DeptID;
    String DeptName;
    int [] AllDoctors;
    int HospitalID;
    public String getDeptName()
    {
        //code to get DeptName
    }
    public int getHospitalID()
    {
    }
    public String getHospitalName()
    {
    }
    public int getDoctors(Doctor DoctorArray[])
    {
    }
}
```

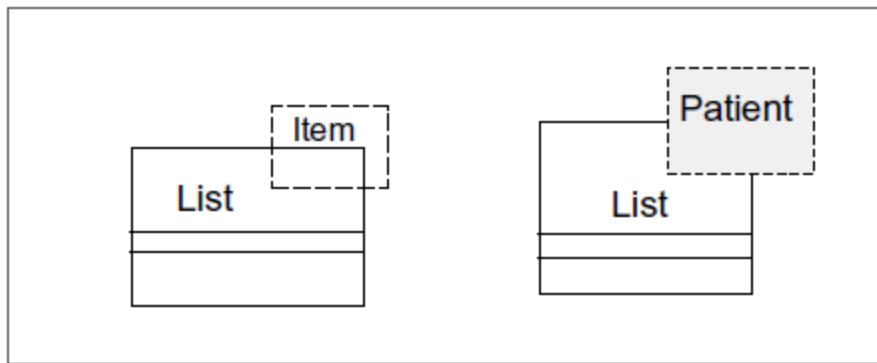
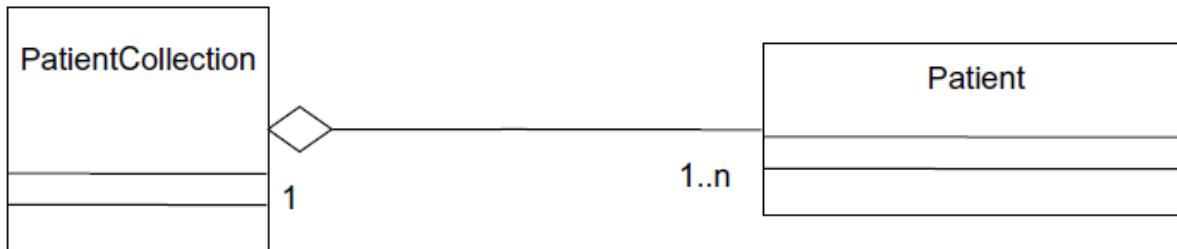
```
}class Hospital
{
    int HospitalID;
    String HospitalName;
    Department [] AllDepartments;
    Address HospitalAddress;
    public String getHospitalName()
    {
    }
    public int getDepts(Department DepartmentArray[])
    {
        //get list of departments in the hospital
    }
}
```

Collection class and multiplicities

- a multiplicity of more than one indicates a range of numerous objects instantiated for that class. For example, if class **Patient** has a multiplicity of more than one, then there are numerous Patient objects instantiated from that class.
- two categories of operations:
 - a) Operations that deal with *an individual* Patient object, associated with the behavior of a single patient, such as *createPatient()*, *calculateAge()*, *getDiagnosis()*, or *changeDetails()*.
 - b) Operations that deal with a *group or collection* of Patient objects: *listPatientByName()* and *totalPatients()*. They do not apply to an individual Patient.
- Operations that apply to a collection of objects point to the need for a “container” or **Collection class**. Collection classes are modeled on the container classes available in the development environment : **sets, arrays, lists, dictionaries, stacks, and queues**.

Collection classes in UML

- Container classes are often modeled as parameterized classes in the UML.



Common errors in designing attributes

- Naming a class as an attribute, e.g., making **Address** an attribute of **Patient**, whereas **Address** is more suited as a class in its own right
- Naming an operation as an attribute, e.g., **getName** is a bad name for an attribute as it implies an action and not a characteristic of a class
- Naming an attribute value as an attribute, e.g., Sam is the value of an attribute called **Name** but is not an attribute itself
- Not initializing attributes when required, e.g., **DateOfBirth** is an attribute of patient class that should be initialized as “oo/oo/oo.” A counter and should be initialized (i.e., should be **PatientCount:= 0**)
- Not providing appropriate visibility—attributes should have a private visibility by default, although occasionally global attributes may have public visibility. Not providing visibility or providing inappropriate visibility is a design error.
- Giving the wrong attribute type for an attribute, e.g., **AccountBalance**: CHAR; attribute type for account balance should be a CURRENCY or DOUBLE.
- Not providing attribute **stereotypes** when required. Stereotype is a grouping mechanism in UML, and if there is a large number of attributes in a class, it is always a good idea to group them by their stereotypes. Common stereotype examples for attributes are <<entity>>, <<business>>, <<date>>, <<counter>>, and so on.

Common errors in modeling advance class design

<i>Common Errors</i>	<i>Rectifying the Errors</i>
Not completing attribute details in design	Ensure each attribute is formally defined with its type, visibility, and initial value.
Not having full operation signatures	Ensure operations have parameter list, visibility, and return values.
Designing solution-level class diagrams directly from the basic class designs	Ensure sequence and state machine designs are drawn as part of the process of developing models before moving to system design.
Overuse of multiple inheritance	Avoid multiple inheritance where possible at least in the problem space (entities)
Exploring polymorphism without good inheritance design	Carefully created inheritance hierarchies with proper operator overloading is required for polymorphism.
Confusing by reference and by value	By reference is where the same data are pointed to (referenced) by multiple sources; by value is where a copy of the data is made before processing.

Designing the interfaces

- After completing the use case diagrams and the first stable version of interaction and class diagrams are developed, it is recommended to implement a **prototype of the IT system interface**.
- This prototype is called **interface prototype** because it has the role to:
 - Refine the relationships between actors and interface classes;
 - Obtain feedback from the client (client) on the visual aspect of the application.

Specifying interface requirements

- Interface specifications include details of what is required for the actor to interact with the system. : name, title, and some descriptive information on how the interface is used by the actor.
- Thus, an interface specification document includes:
 - The goals of the actor in interacting with the system (also documented in use cases),
 - List of interfaces (based on the variety and number of interactions an actor will have with the system), and
 - Navigation and dependencies (based on the process flows documented in use cases but now revised based on the interfaces).
- The UML standard plays a minimal role in modeling a UI. There is no UML diagram to model an interface.
- UIs can be specified using templates, and

Interface specification template

Interface Identifier:

<This is the number and name of the interface being specified>

Actors:

<A list of the actors who interact with the system through this interface>

Use Cases:

<List of use cases or one-line description of the use case in which this interface appears>

Short Description:

<Description of the interface in a few lines>

User Interface Identifier: UI10-PatientRegistrationForm

Actors: A10-Patient, A80-Administrator

Use Cases: UC10-RegistersPatient; UC12-MaintainsPatientDetails

Short Description: This UI enables the creation of registration details for a first-time patient at the hospital. This same UI also enables maintenance of a patient's registration details. Specific registration details of the patient for this interface are available in the respective use cases.

Designing the interfaces

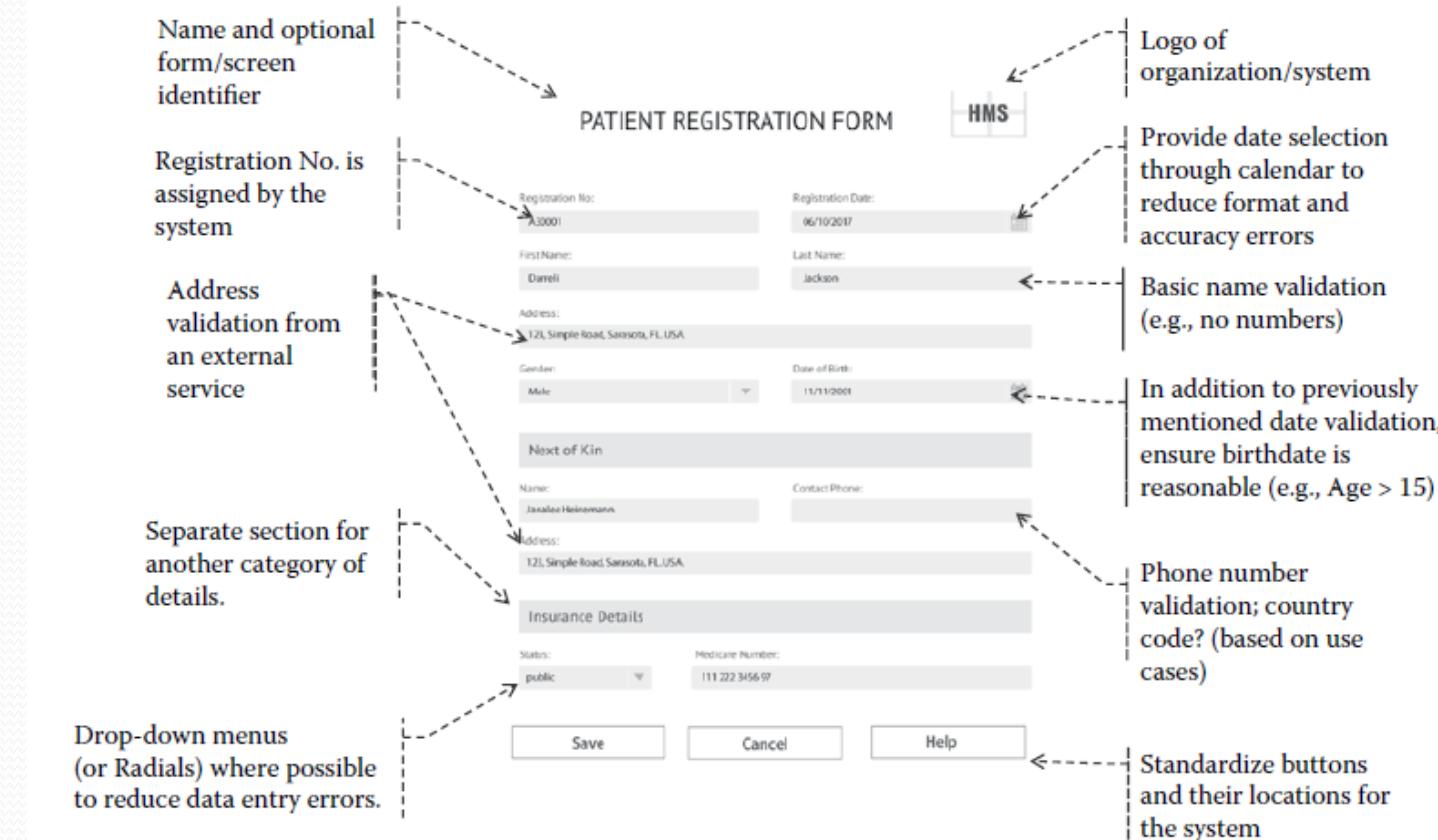
- The first step - investigating *the actors' expectation on the interface* by completing specific **questionnaires** consisting of the following questions:
 - What level of training (computer science) the actor requires to achieve a certain functionality?
 - Does the actor have working experience in window-based environments?
 - Does the actor have experience in using other automated process modeling systems?
 - Is it necessary to consult documents / catalogs in parallel with the use of the application?
 - Does the actor want to implement 'rescue / restoration' facilities?

Designing the interfaces

- The goals of the prototype are:
 - Setting *interface requirements* for key application functionalities;
 - It demonstrates to the client (in a visual form) that the project requirements have been well understood and are achievable;
 - The start of the development phase of the standard interface elements.

Details for interface design

- Each UI has its data described as attributes within classes/programs that manipulate and display those data. These interfaces are iteratively designed and developed. For example, an initial sketch in the first iteration in the problem space, it does not provide specific design-level details such as color, size of text boxes, and so forth.

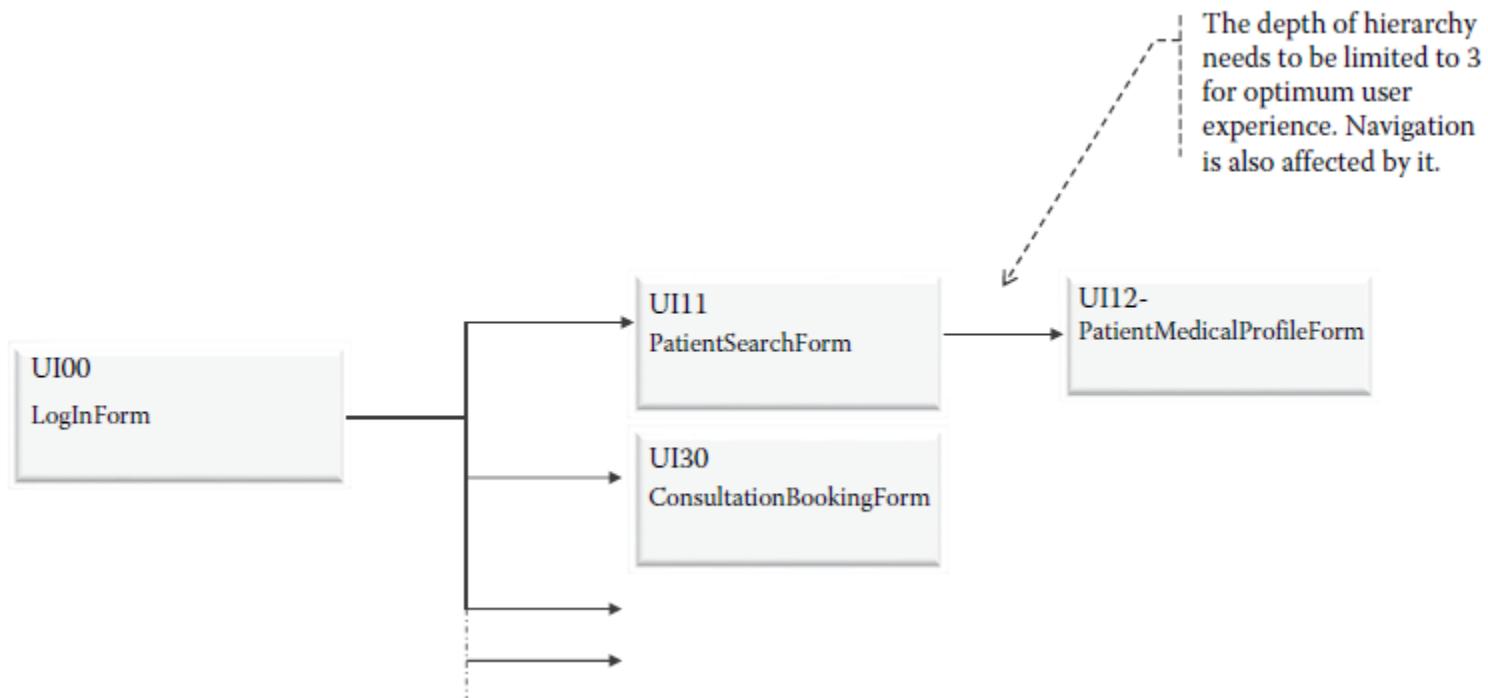


Designing the interfaces

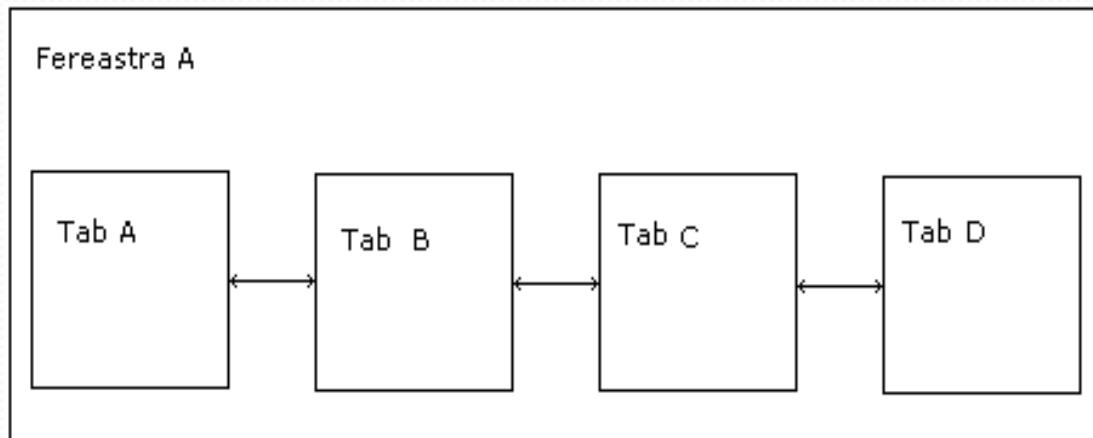
- **Screen structure maps** (charts) are used to describe the flow of the application following the main ways of use.
- Representation :
 - **Square shapes** for ***modal window*** representation (requires a user response to continue an activity).
 - **Square shapes with rounded corners** for the representation of ***non-modal windows***
- The crossing direction shows the window navigation path.

Specifying the flow of user interfaces – navigation map for Doctor

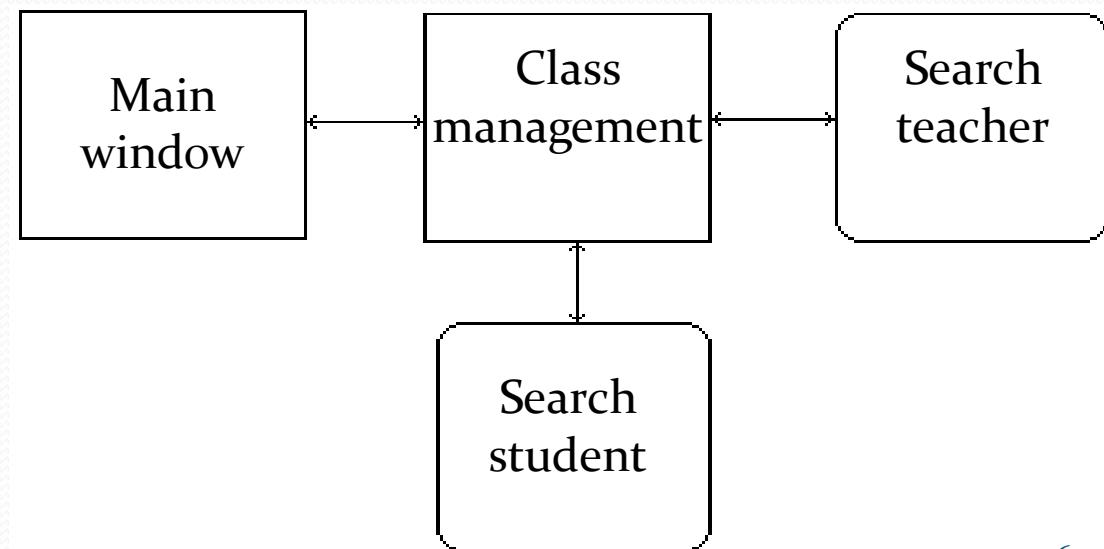
- UI flow diagrams, occasionally also called **storyboards**, **screen navigation diagrams**, or **navigation maps**, model the relationships and dependencies between UIs. Since the UML provides an **activity diagram** as a flowchart, we can create this navigation map. The screens are represented by **objects** in this activity diagram.



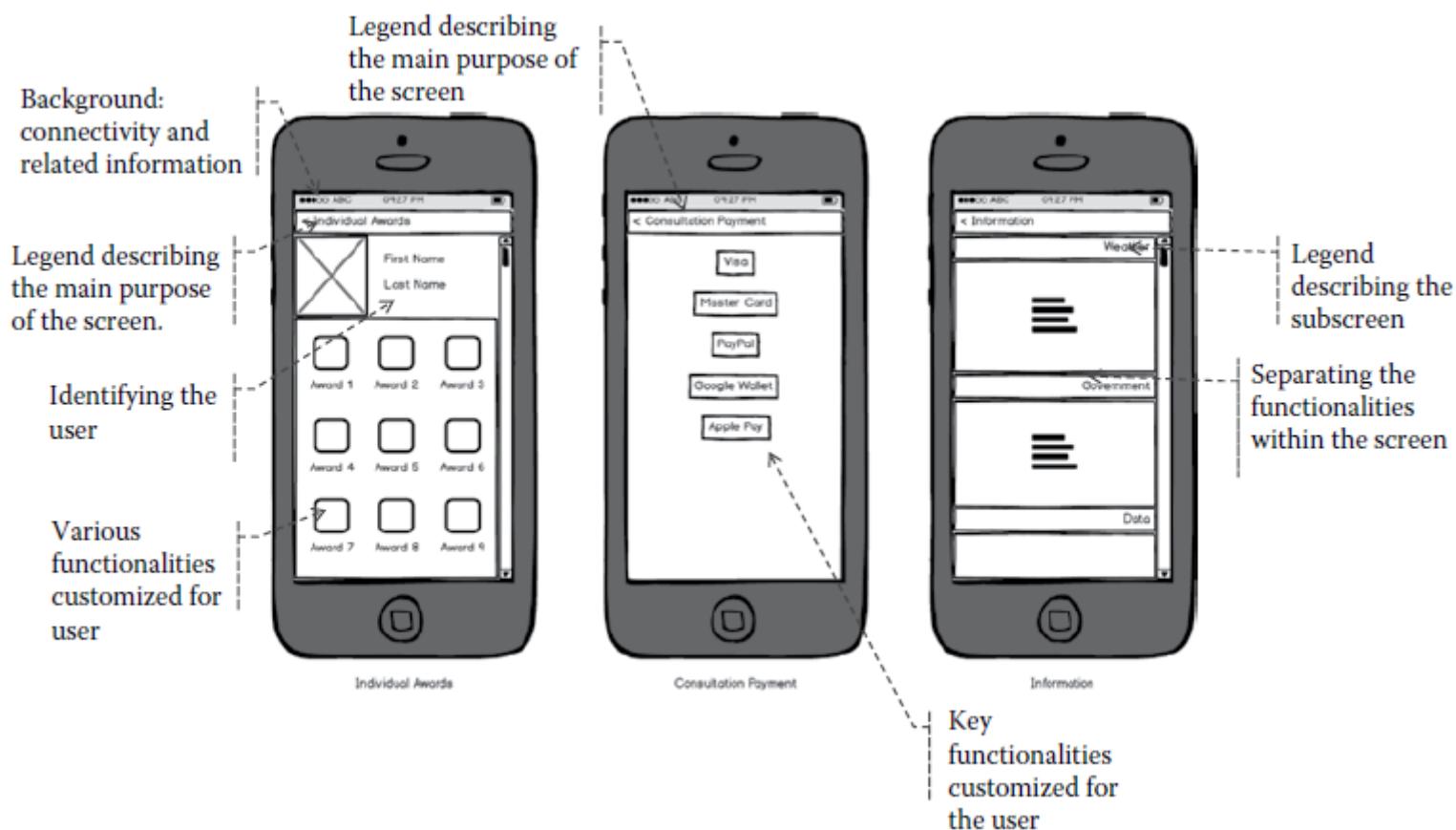
Windows that contain Tabs



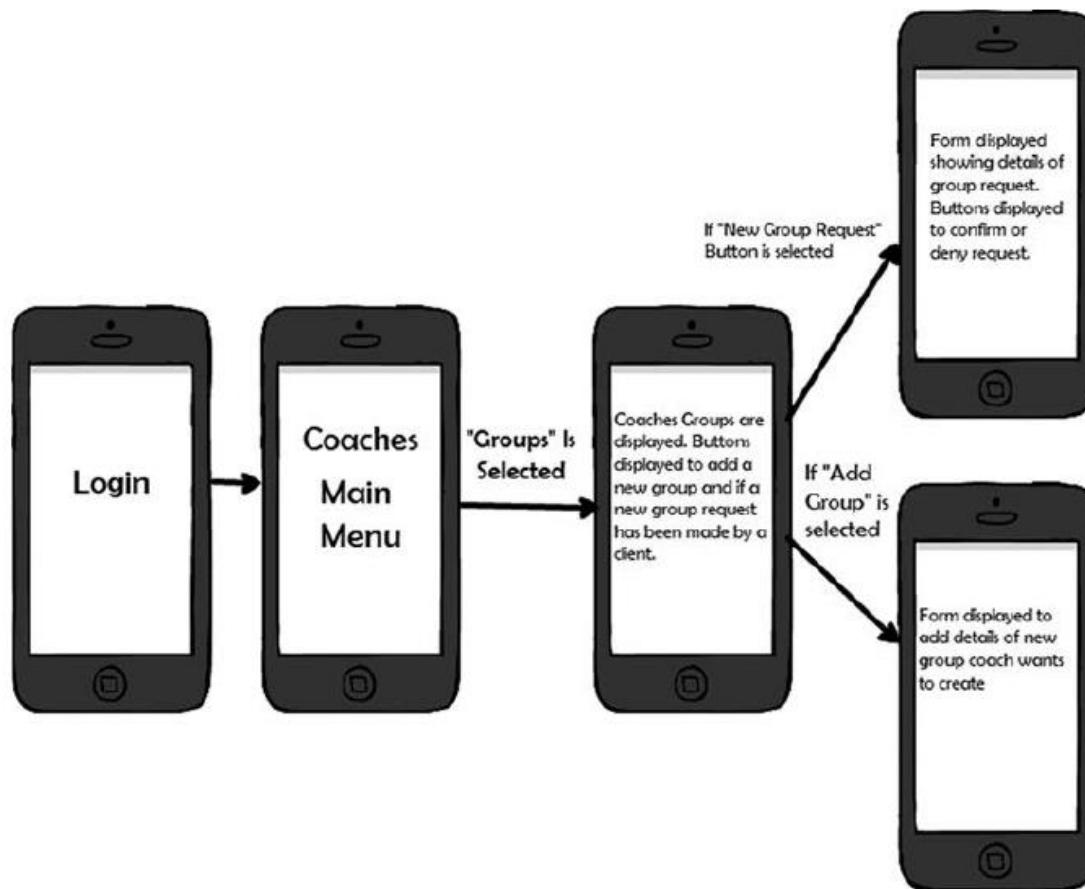
Screen structure diagram



Designing mobile app interfaces with mockups



Designing mobile app interfaces with storyboards

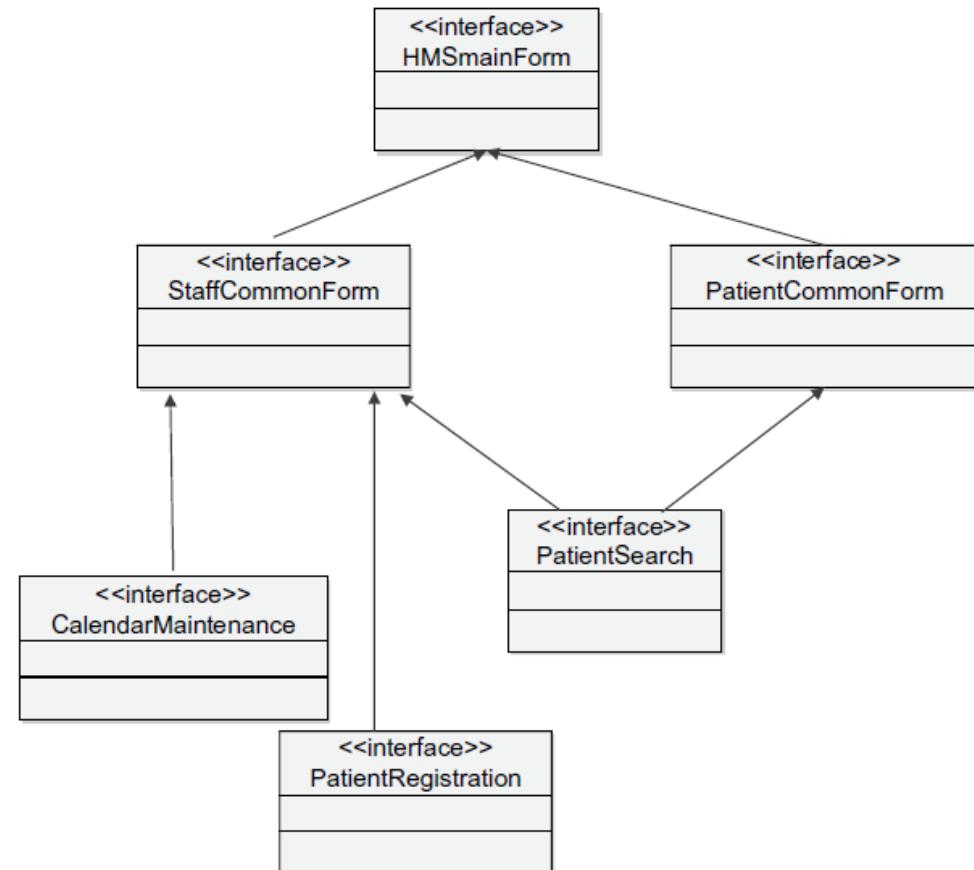


User interface design considerations

- The business analyst is initially responsible for the UI specifications. These specifications are analyzed during UI design to create detailed and implementable interface classes.
- The resultant GUIs can then be organized according to the actors (users) who will be using them.
- The GUIs are usually derived from existing graphic support provided by the development environment : windows, scroll bars, and radio buttons, etc. A FORM object is usually available and provides most GUI functionalities
- Interfaces can be also created and reused by designers themselves: a common interface class is created that represent the
- A common interface class is created that represents the primary window for the actor interaction. Then the rest of the interfaces required for that actor can inherit the common functionality provided by the common interface class

Deriving modeling of interfaces: inheritance hierarchy

- All boundary classes are derived from a main **HMSmainForm** class (based on classes provided by the development environment). The actor-specific classes that provide the common interfaces for the actor are the **StaffCommonForm** and the **PatientCommonForm**.



Thirteen principles of display design

- Christopher Wickens et al. defined 13 principles of display design in their book *An Introduction to Human Factors Engineering* (2004)
- can be utilized to create an effective display design.

POTENTIAL BENEFITS

- a reduction in errors, in required training time,
 - an increase in efficiency,
 - an increase in user satisfaction
-
- Grouped under 4 categories

a. Perceptual principles

- 1. *Make displays legible (or audible).* A display's legibility is critical and necessary for designing a usable display.
- 2. *Avoid absolute judgment limits.* Do not ask the user to determine the level of a variable on the basis of a single sensory variable (e.g. colour, size, loudness).
- 3. *Top-down processing.* Signals are likely perceived and interpreted in accordance with what is expected based on a user's experience. If a signal is presented contrary to the user's expectation, more physical evidence of that signal may need to be presented to assure that it is understood correctly.
- 4. *Redundancy gain.* If a signal is presented more than once, it is more likely that it will be understood correctly. This can be done by presenting the signal in alternative physical forms (e.g. colour and shape, voice and print, etc.), as redundancy does not imply repetition.
- 5. *Similarity causes confusion: Use distinguishable elements.* Signals that appear to be similar will likely be confused. The ratio of similar features to different features causes signals to be similar. For example, A423B9 is more similar to A423B8 than 92 is to 93.

b. Mental model principles

- 6. *Principle of pictorial realism.* A display should look like the variable that it represents (e.g. high temperature on a thermometer shown as a higher vertical level). If there are multiple elements, they can be configured in a manner that looks like it would in the represented environment.
- 7. *Principle of the moving part.* Moving elements should move in a pattern and direction compatible with the user's mental model of how it actually moves in the system. For example, the moving element on an altimeter should move upward with increasing altitude.

c. Principles based on attention

- 8. *Minimizing information access cost* or interaction cost. When the user's attention is diverted from one location to another to access necessary information, there is an associated cost in time or effort. A display design should minimize this cost by allowing for frequently accessed sources to be located at the nearest possible position. However, adequate legibility should not be sacrificed to reduce this cost.
- 9. *Proximity compatibility principle*. Divided attention between two information sources may be necessary for the completion of one task. These sources must be mentally integrated and are defined to have close mental proximity. Information access costs should be low, which can be achieved in many ways (e.g. proximity, linkage by common colours, patterns, shapes, etc.). However, close display proximity can be harmful by causing too much clutter.
- 10. *Principle of multiple resources*. A user can more easily process information across different resources. For example, visual and auditory information can be presented simultaneously rather than presenting all visual or all auditory information.

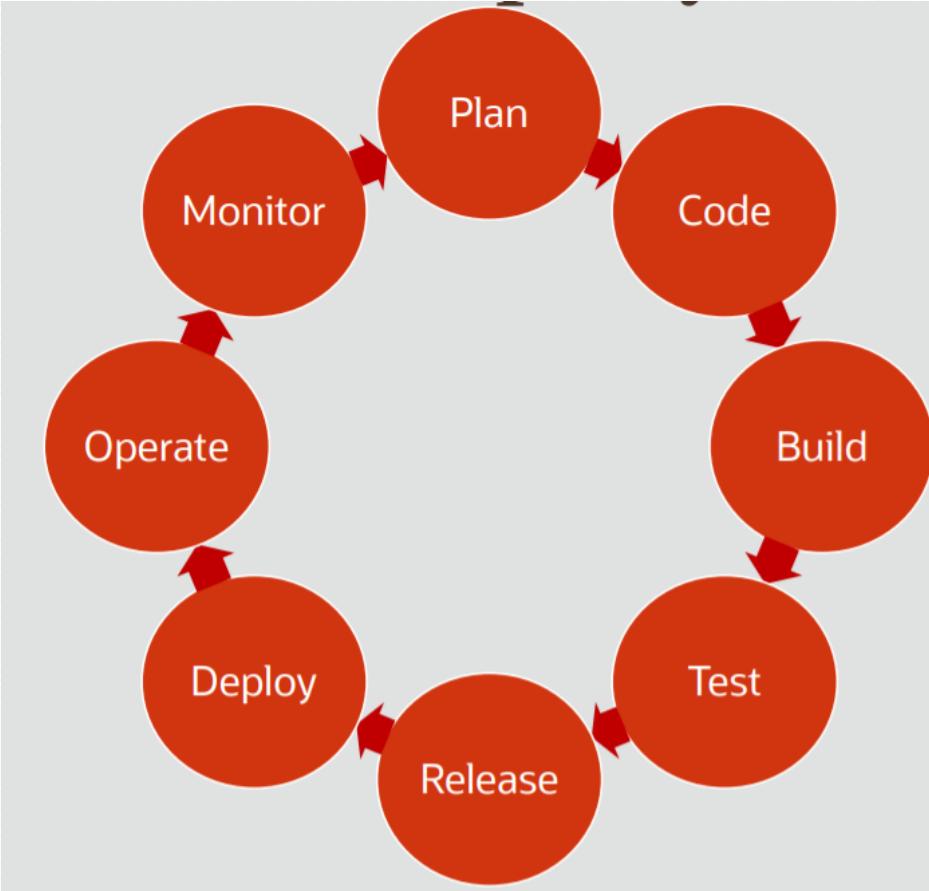
d. Memory principles

- 11. *Replace memory with visual information: knowledge in the world.* A user should not need to retain important information solely in working memory or retrieve it from long-term memory. A menu, checklist, or another display can aid the user by easing the use of their memory. The use of knowledge in a user's head and knowledge in the world must be balanced for an effective design.
- 12. *Principle of predictive aiding.* Proactive actions are usually more effective than reactive actions. A display should attempt to eliminate resource-demanding cognitive tasks and replace them with simpler perceptual tasks to reduce the use of the user's mental resources. This will allow the user to focus on current conditions, and to consider possible future conditions. An example of a predictive aid is a road sign displaying the distance to a certain destination.
- 13. *Principle of consistency.* Old habits from other displays will easily transfer to support processing of new displays if they are designed consistently. A user's long-term memory will trigger actions that are expected to be appropriate. A design must accept this fact and utilize consistency among different displays.

Information system design

DevOps approach

DevOps



DevOps is a *culture, movement or practice* that emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes. It aims at establishing a culture and environment where *building, testing, and releasing software, can happen rapidly, frequently, and more reliably.*

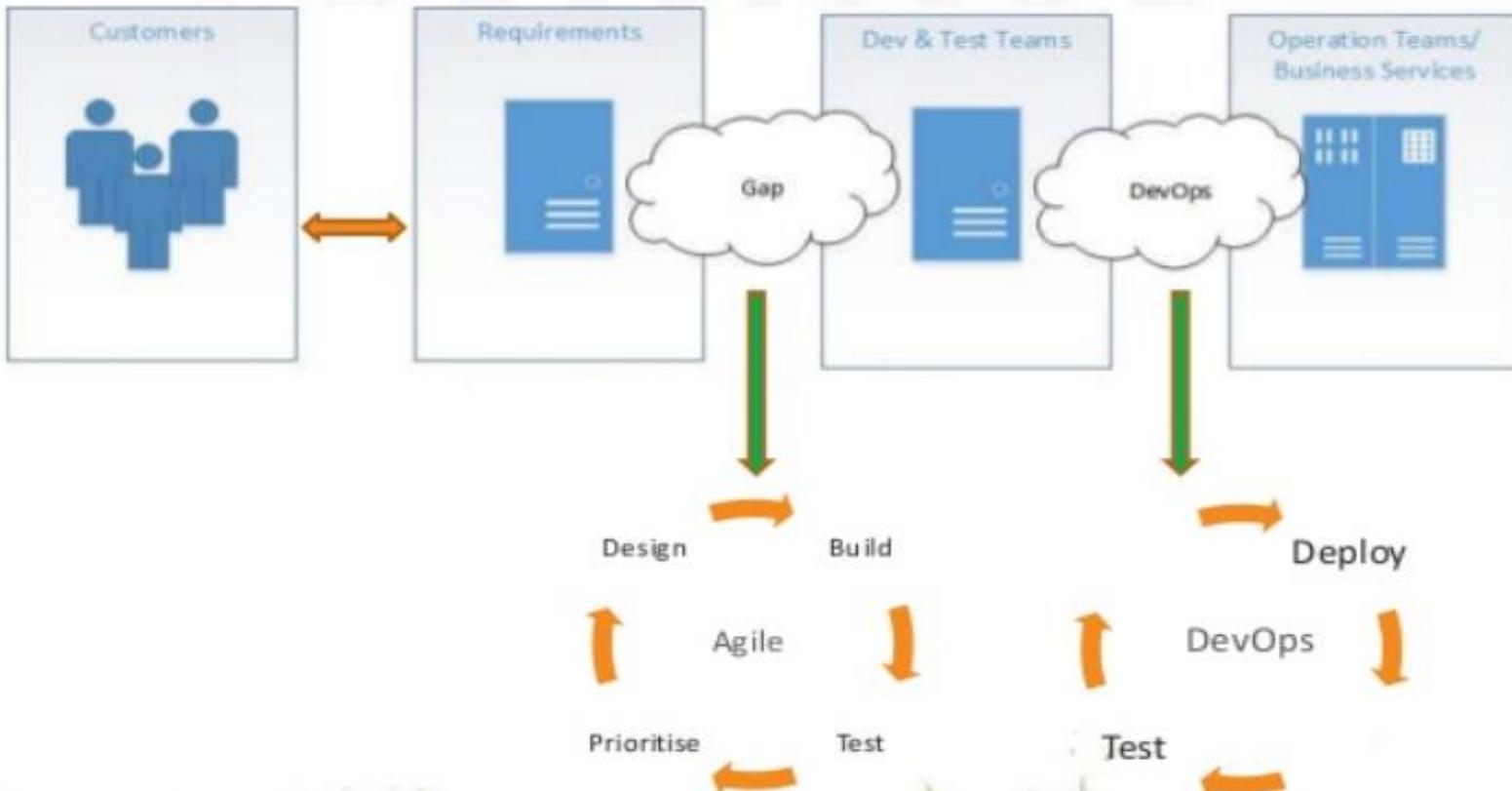
Wikipedia

DEVELOPMENT + OPERATIONAL

DevOps - definition

- The classic approach to IT projects is for the **development team** to be different from the **testing, deployment and operations team**.
- DevOps is a set of practices that aims to replace the classic approach with a single **multidisciplinary team** that works more efficiently and faster.
- The goals of this approach are for software to be released faster, product issues to be resolved faster, and unplanned events to be resolved more easily.
- *“the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support”*

DevOps and agile



- Addresses the gap between customer requirements and dev+ testing team
- Cross-functional teams to design, develop and test features prioritized by the customer
- Focuses more on functional and non-functional readiness

From classic to Agile approach

- In the classic approach, we had a business analyst team that discussed with the beneficiary and formulated the requirements.
- After receiving these requirements, the implementation team no longer had a permanent dialogue with customers, most of the time until the end of the project.
- Here is the area where an agile approach eliminates this gap and improves customer interaction and adjusting initial requirements.
- So, the agile approach addresses the gap between customer requirements and dev + testing team. Cross-functional teams are recommended to design, develop and test features prioritized by the customer

Addressing delivery challenges



- Addresses the gap between Development_ Testing and Operational
- Automated release management
- Focuses on functional and non-functional plus operational and business readiness
- Intensifies reusability and automation

From agile to DevOps

- The second gap we have represented in this image is the one between the development and testing teams and the Operation Teams.
- This is the area addressed by Devops through Automated release management.
- The classic approach to IT projects is for the development team to be different from the testing, deployment and operations team.
- DevOps is a set of practices that aims to replace this approach with a single multidisciplinary team that works more efficiently and faster.
- The goals of this approach are to get the software up and running faster, to solve product problems faster, and to fix unplanned events more easily.

Improving in software delivery process

- DevOps is a concept that emerged from the process of improving the continuous delivery of software.

	1970-1980	1990-2000	2000-present
Age	Mainframes	Client / Server	Cloud & DevOps
Technologies	COBOL, DB2 MVS	C ++, Oracle, Solaris etc.	Java, MySQL, Azure, AWS
Time per cycle	1-5 years	3-12 months	2-12 weeks
Cost	\$ 1M - \$ 100M	\$ 100k - \$ 10M	\$ 10k- \$ 1M
Risk	The whole company	The division of a company	Problems with a software product
Failure costs	Bankruptcy, dismissals etc ...	Profit loss, CIO fired	Negligible

Principles of DevOps

- **DevOps pillars** are reflected in the **CAMS** acronym:
 - **Culture** represented by human communication, technical processes, and tools
 - **Automation** of processes
 - **Measurement** of KPIs (quality assurance)
 - **Sharing** feedback, best practices, and knowledge
- DevOps =automating all the processes within the team +feedback for newly developed thing
- Delivery? Several times a day

Continuous integration

- One of the practices that has become increasingly known in the software development process is **continuous integration**.
- **Continuous integration** can be achieved in several ways, using different technologies. This is a software development practice that consists of integrating the source code at least once a day per developer automatically.
- It also involves checking the newly added code to determine its impact on existing code, **using automated testing** performed on code integration.

DevOps – Version control

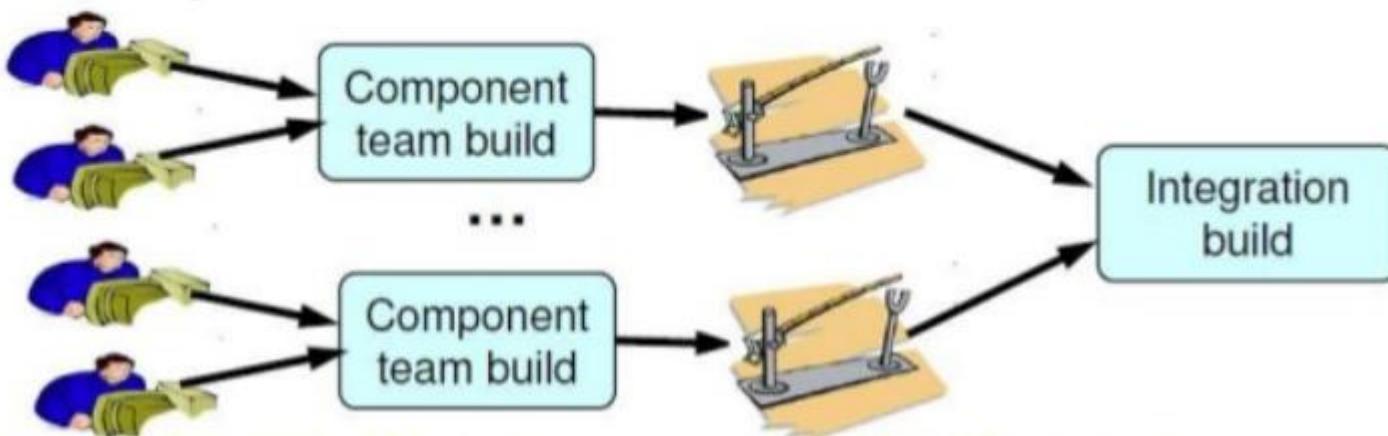
- Continuous development and continuous integration certainly require the implementation of version control.
- A historical versioning is applied to the source code of the software
- In order to adopt DEVOPS, it is necessary to implement a version control system
 - Git,
 - Bitbucket,
 - SVN, etc

DevOps – Continuous testing

- **Automated testing** was first proposed by Grady Booch in 1991.
- Using this approach allows software development teams to receive constant feedback on changes they make to an application, and this is a cheaper way to solve problems when they are found, because the newly changed code is still fresh for developers.
- Test scenarios of the application can be created that can be run automatically every time new code has been developed.
- **Types of automatic testing:**
 - Automatic code testing (unit test).
 - Automatic component testing.
 - Automatic integration testing.
 - Automatic testing of application interfaces (API test).
 - Automatic user interface testing (GUI test)

DevOps – Continuous integration

- Integrate the code changes by each developer so that the main branch remain up-to-date
- This process takes the latest version of the code and automatically executes predefined steps. A standard process contains application building and automated testing steps.

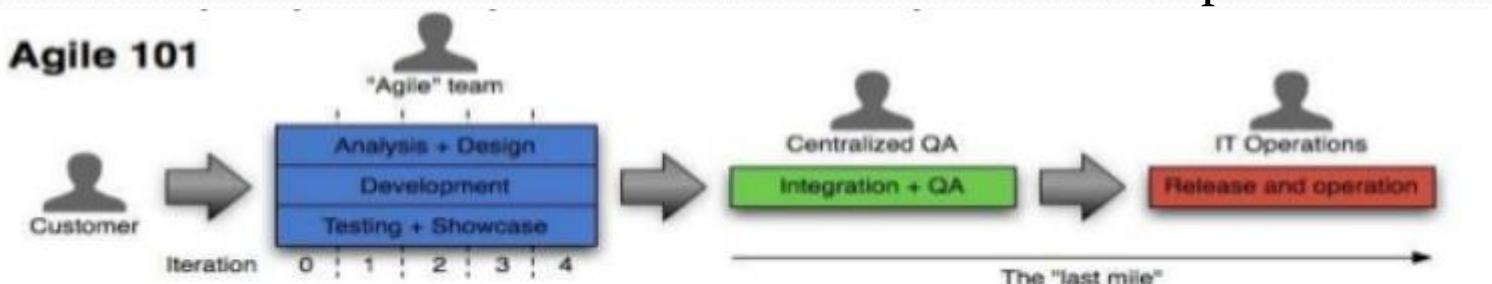


Advantages of continuous integration

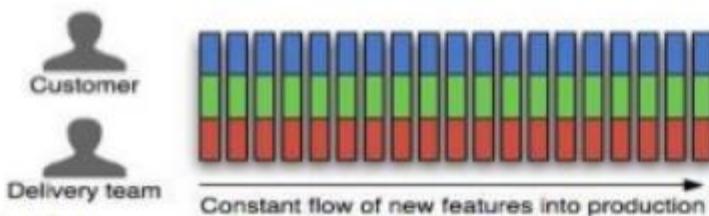
- Product quality is improved, because errors are detected early through the automatic testing process
- Creating new **deliveries** to the production environment is **easier**
- The time spent for **testing is shorter** and implicitly the testing costs are lower
- Developers receive **immediate feedback** and issues can be addressed on the spot,

DevOps – Continuous delivery

- Taking each CI build and run it through deployment procedure on production.
- Steps of this automated process are application creation, testing, configuration and deployment. They can be executed for several intermediate environments until it reaches the final production



Continuous Delivery



Acceptance test driven development process
Tight collaboration between business and delivery teams
Cross-functional teams include QA and operations
Automated build, testing, db migration and deployment
Incremental development on mainline with continuous integration
Software always production ready
Releases tied to business needs, not operational constraints

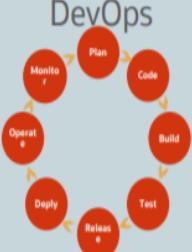
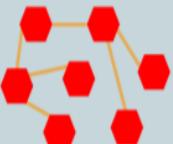
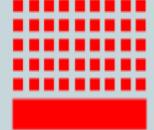
Advantages of continuous delivery

- Abstracts the complexity of a delivery
- It can be **delivered more often**
- Small changes can be **delivered faster**, so iterations are faster
- The customer can see that the improvements to the system are continuous and frequent
- Deliveries of new versions of the product are **less risky**

Tools for CI

- A CI pipeline consists of a set of tools created in order to host, monitor, compile and test the respective code and its changes.
- Among the tools mentioned above, we mention:
 - Server for continuous integration, such as Jenkins, TeamCity, CruiseControl
 - Version control tools, such as Git, SVN, Mercurial
 - Build system, such as Maven, Gradle, or Ant.
 - Framework for automated testing, such as Selenium

Applications' development evolution

	Development Process	Application Architecture	Deployment and Packaging	Application Infrastructure
~ 1980	Waterfall 	Monolithic 	Physical Server 	Data Center 
~ 1990				
~ 2000	Agile 	N-Tier 	Virtual Servers 	Hosted 
~ 2010	DevOps 	Microservices 	Containers 	Cloud 
Now				

Lecture 11

System Design Phase (3rd Part)

Agenda

1. **Persistence in software engineering**
2. Design patterns
3. System structure: Package diagrams
4. Implementation diagrams

Stereotypes for persistence

- Most *<entity>* stereotyped objects have a need to exist even after the system is shut down
- While a persistent object exists beyond the execution of the system, a transient object vanishes when the system is shut down and is recreated when the system is executed again. Most *<control>* and *<boundary>* objects are transient, and there is no need to save these types of objects.

Persistence mechanism - Databases

- Persistence can take various shapes and forms. What follows are some techniques for making objects persistent:
 - Storing in a **simple flat file**. Such storage can contain the data that are used by the system but is not intelligent and does not offer a way to search within the data.
 - Storing in an **indexed sequence access mechanism file** or database that organizes data through indexes that can be used for searching specific data records and updating them.
 - Storing in a **relational database**, which is most appropriate for business data that can be easily formatted into rows and columns, thereby lending themselves to search.
 - Storing in an **object-oriented database**, which are more appropriate for unformatted or scientific information.
 - Storing in a **NoSQL database**, which can handle large, unstructured documents and machine data that can also be optionally searched.

Database design

- Starting with the **domain class model** (or ERD)
- Choose **database structure**
 - Usually relational database
 - Could be ODBMS framework or other NoSQL systems
- Design **DB architecture** (distributed, etc)
- Design **database schema**
 - Tables and columns in relational, usually
- Design referential **integrity constraints**
 - Foreign key references/ other types of constraints

Object oriented databases

- OODB are able to store objects together with their attribute values, operations, and relationships.
- The object structures in the memory during execution can be directly stored “as is” in the database. Binary large objects (*BLOBs*) and complex unstructured data (e.g., video and audio) can also be stored in these databases without the need for conversion
- They also store relationships like inheritance, association, and aggregation directly in the database.

NoSQL databases

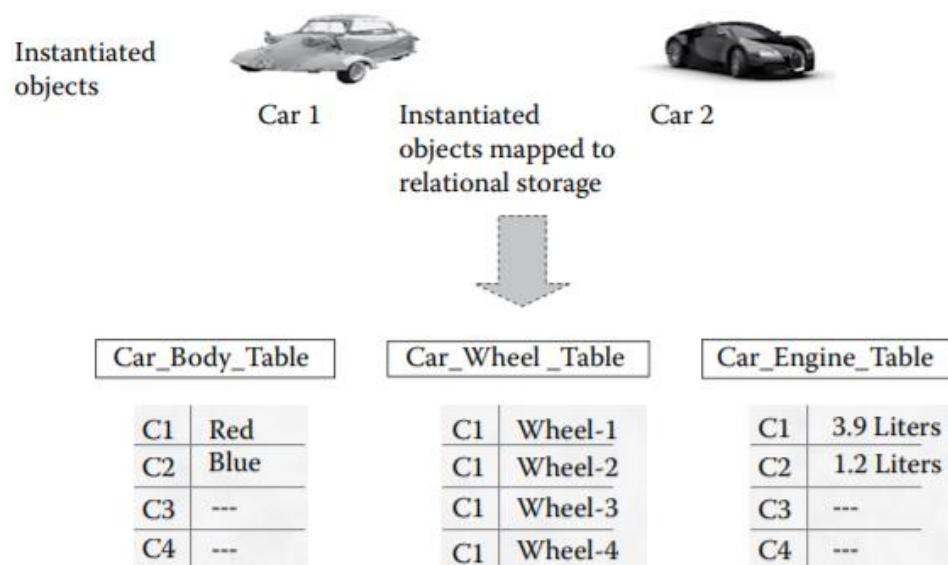
- It does not follow the traditional row-column format of a structured query language (SQL), (relational) database but is able to handle unstructured data.
- The underlying technology enables large-scale data handling.
- NoSQL databases handle a highly complicated and federated database structure that usually resides in the Cloud. The federated database structure of NoSQL databases is also understood as a distributed database architecture
- For example, a **Customer-Account** relationship it is not just an association; the **Account** is a collection of accounts that belong to the customer and will be embedded within each **Customer**

Relational databases

- Most commercial business applications still use relational databases. They provide ideal and mature mechanisms to store, retrieve, and manage data that are structured
- Tables are structurally different from objects and this requires “translation” of classes to tables
- Most UML-based modeling tools enable the architect to mark the classes as persistent, they can then be used by the tools to create an initial relational database schema from the class diagrams

Challenges in storing objects in RDB

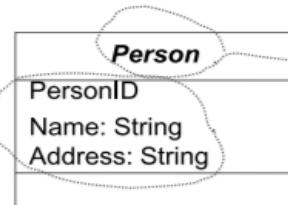
- The key challenge is that objects cannot be directly written to or read from relational databases. While the objects have data and behavior, tables only store data



Mapping OO to tables

- The simplest form would be a one-to-one mapping.
- All the attributes of a class would be converted to the columns of a table.
- Each instance would be stored as a row in the table.

- Class:



Class	-	Table
Attributes	-	Column
Objects	-	Rows

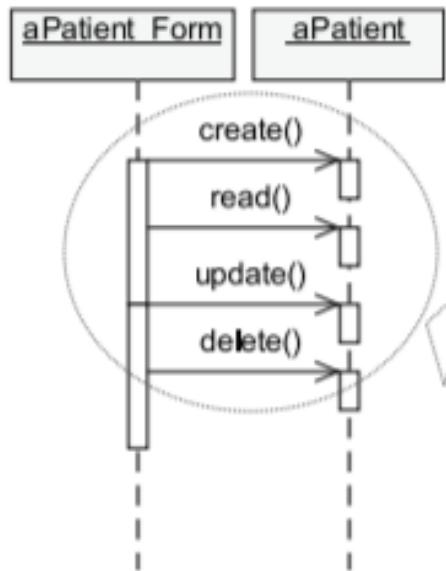
- Database Table:

Person_Table

PersonID	Name	Address
101	Darrell Jackson	83, Walker Street
221	Wei Liu	67 Miller Pde

Basic persistence functions

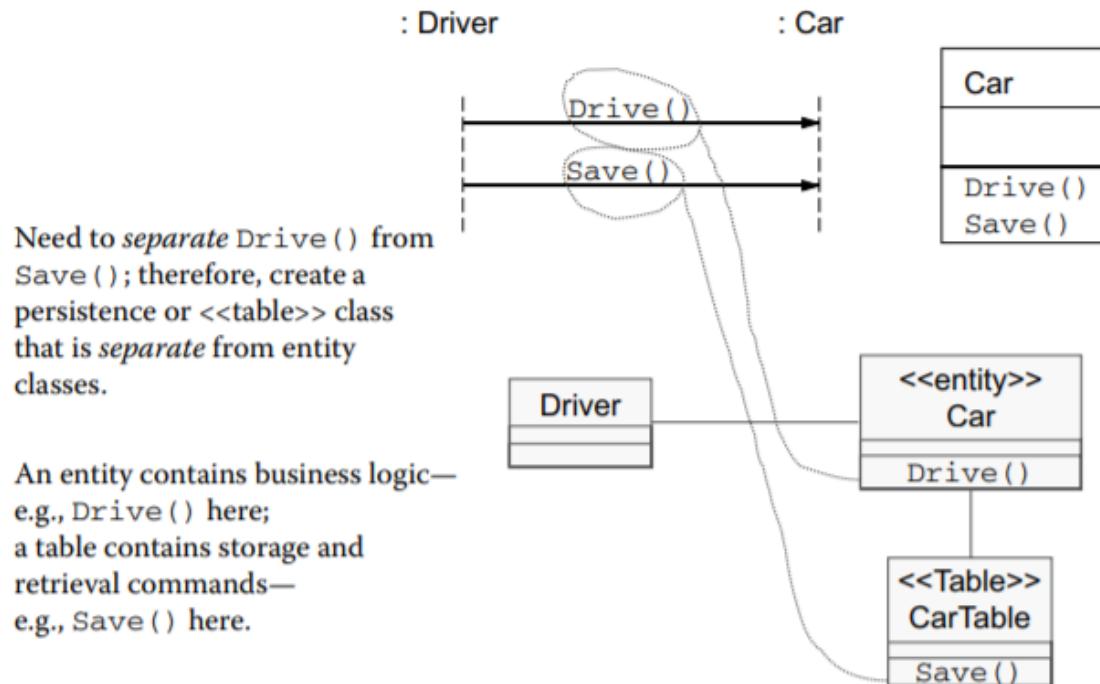
- CRUD functions:
 - **Create**—used in creating an object
 - **Read**—used in searching for an object (record) from storage based on a criterion (key)
 - **Update**—used in searching and updating objects (records)
 - **Delete**—used in locating and removing a persistent object (record)



A *<boundary>* object—
aPatientForm —performs the CRUD operations on another object.
The operations are rarely executed together, usually they may be intermingled with business functions.

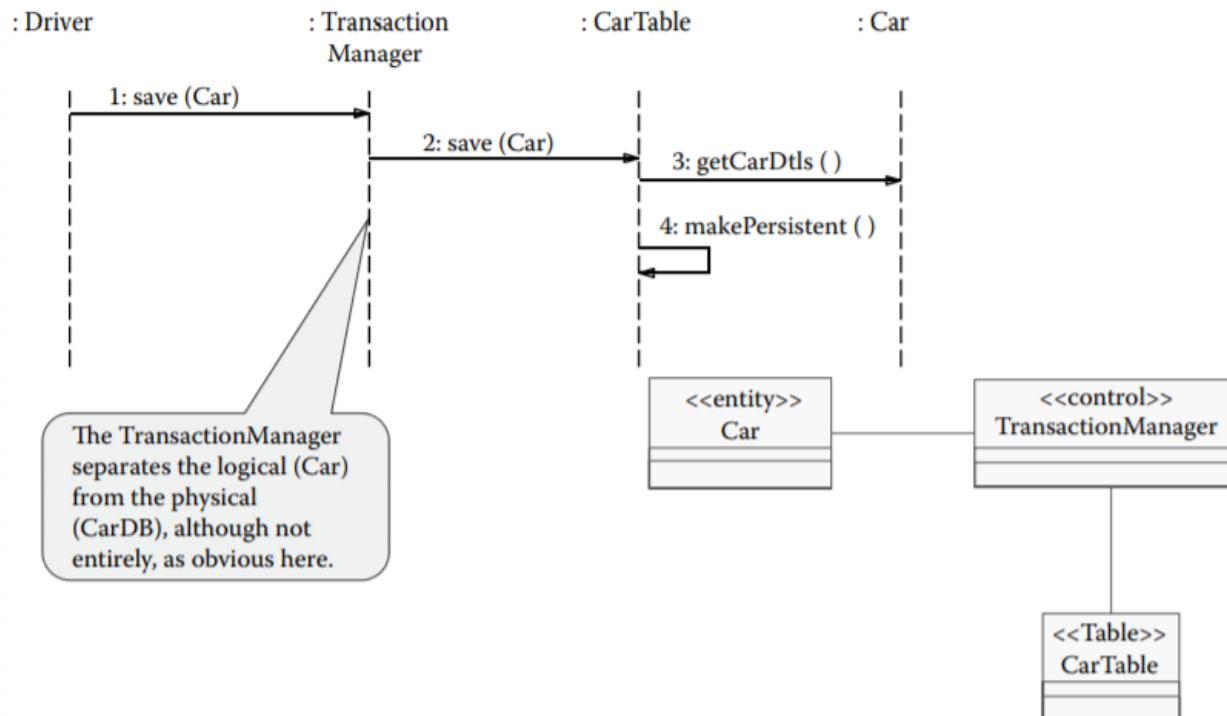
Separating persistence operations from business logic

- Drive and Car are entity classes, but Drive (business behavior) and Save (persistance) are different.



Keeping relational storage and objects separate

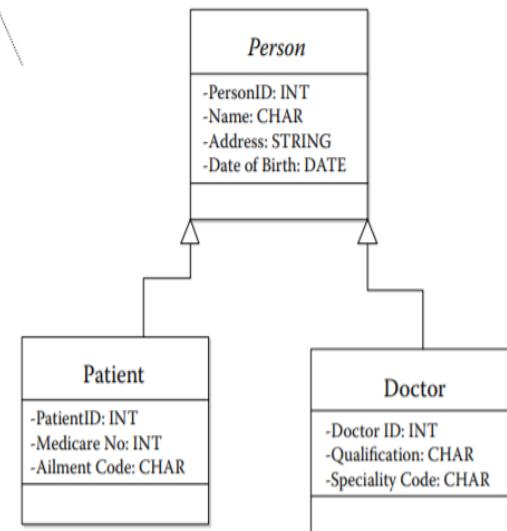
- The object persistence should be separated from the object behavior; the application will remain totally separated from database
- **TransactionManager** is the “control” class that provides the interface to the database



Inheritance relationships and relational tables

What are the possible ways to map Patient and Doctor?

- A single mega table
- Two tables (one for Patient, another for Doctor)
- Three tables?



a

PERSON TABLE

PersonType	PersonID	Name	Address	MedicareNo	Ailmentcode	Qualification
P	P001	Mark	Parrramatta	13254678	PC2003	
D	D001	Bala	Strathfield			M.B.B.S
P	P020	Mariana	Campbelltown	15487962	AS5006	
D	D023	Fiona	Redfern			F.R.C.S

b

PATIENT TABLE

PatientID	Name	Address	MedicareNo	Ailmentcode
P001	Mark	Parramatta	13254678	PC2003
P020	Mariana	Campbelltown	15487962	AS5006

DOCTOR TABLE

DoctorID	Name	Address	Qualification
D001	Bala	Strathfield	M.B.B.S
D023	Fiona	Redfern	F.R.C.S

c

PERSON TABLE

PersonID	Name	Address
P001	Mark	Parramatta
D001	Bala	Strathfield
P020	Mariana	Campbelltown
D023	Fiona	Redfern

PATIENT TABLE

PersonID	PatientID	MedicareNo	Ailmentcode
P001	IP2001	13254678	PC2003
P020	OP2003	15487962	AS5006

DOCTOR TABLE

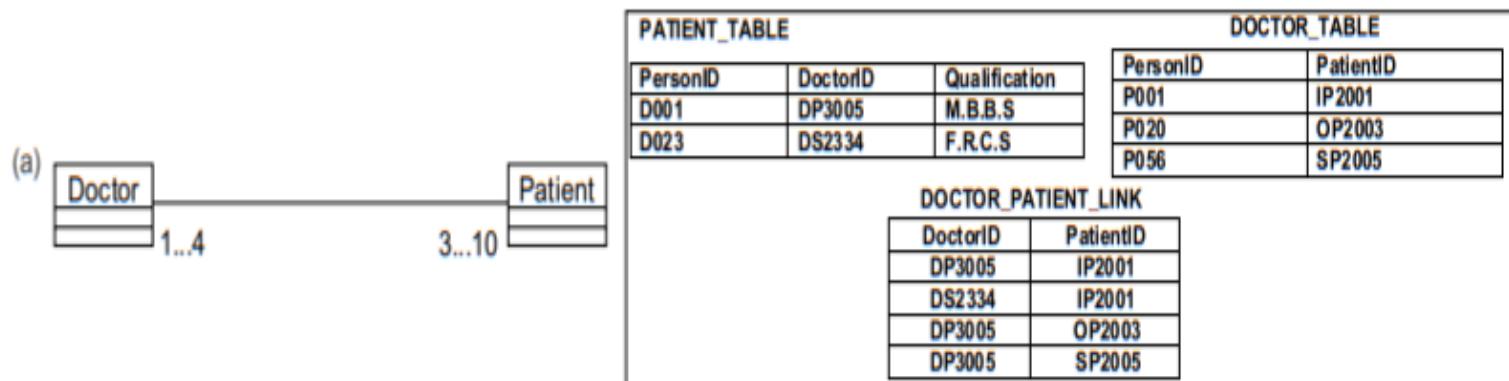
PersonID	DoctorID	Qualification
D001	DP3005	M.B.B.S
D023	DS2334	F.R.C.S

Inheritance relationships

- a) The easiest way is to map all the attributes from the parent class, as well as subclasses, to the columns of a single huge table. Wasted space: when a **Patient** object is stored in this particular table, it would leave the columns specific to **Doctor**
- b) Create tables for all the child classes and append the parent class attributes to it. It becomes more challenging for multiple levels of inheritance
- c) Create separate tables for parent as well as child classes. These tables are then linked using the primary key of the table representing the parent class (PersonID)

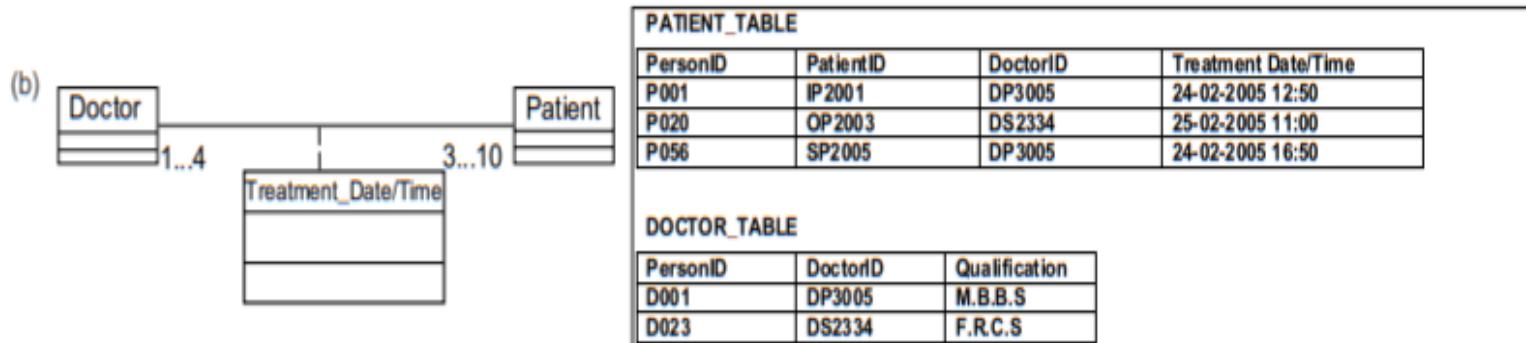
Multiplicities, association class and Link table (a)

- Each class is first translated to a table in the relational database and a new table is created to map the association—many-to-many multiplicities cannot be directly implemented.



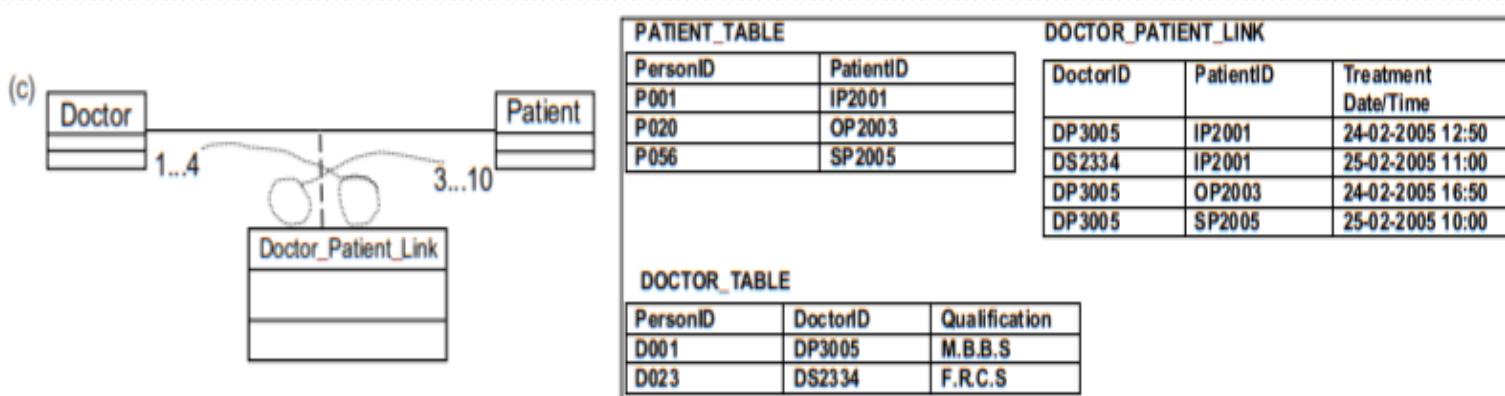
Multiplicities, association class and Link table (b)

- Mapping an association class differs in various cases and depends on multiplicities.
- Mapping will result in two tables, with the key for **Doctor** appended to the table storing patients. However, the association class is not mapped to a table. Instead, the attributes date/time are appended to the **Patient** table



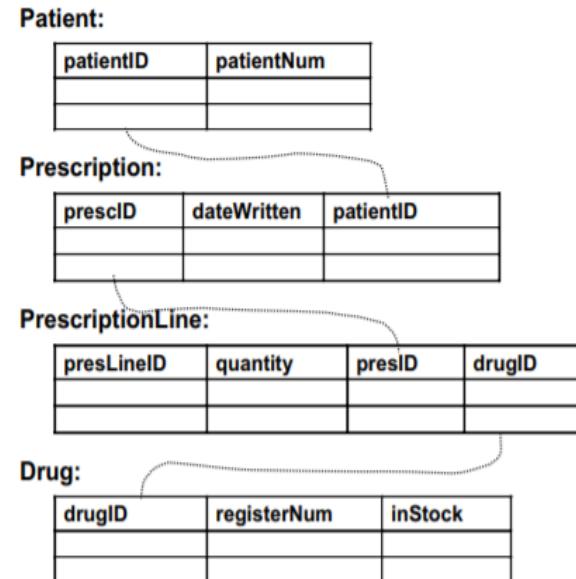
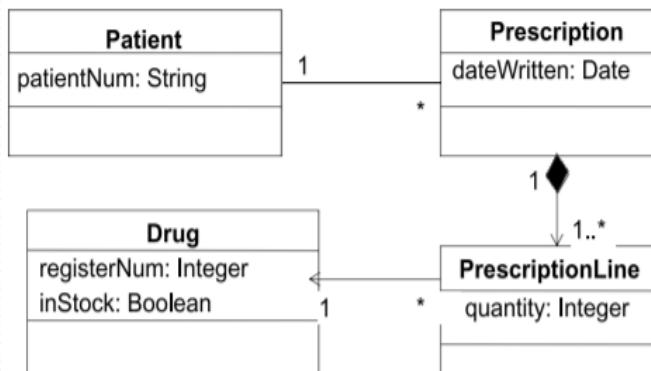
Multiplicities, association class and Link table (c)

- Mapping will change if the multiplicities change. The same association class but different multiplicities in the association will result in three tables instead of two. The association class is directly mapped to a table, which has keys of both **Doctor** and **Patient** along with its own attribute, TreatmentDate/ Time.



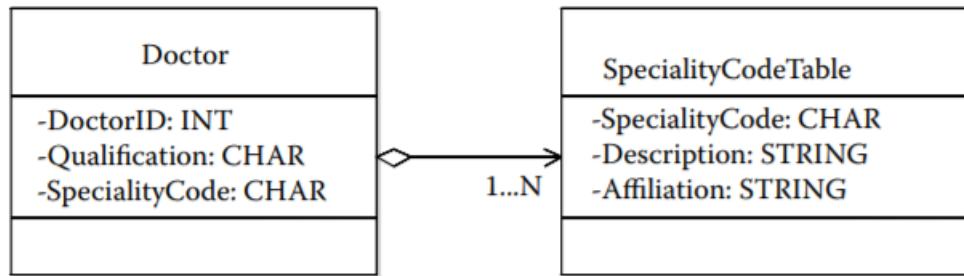
Mapping aggregation (composition)

- Both classes will be mapped to individual tables. The key of the associating class will be appended to the corresponding table: **PrescriptionLine** will contain **PresID**, the key of **Prescription**.
- Whenever a **Prescription** object is destroyed, care has to be taken to destroy all the related **PrescriptionLine** objects. This is achieved will be the primary key of **PrescriptionLine** is a composite key of **PresLineID** and **PresID** together.



Shared aggregation and reference table

- Shared aggregation is a database design concept that helps model (among other things) a “reference table” that contains data referenced by multiple other tables.



- The access key for the **Doctor** table will be appended to the **SpecialityCode** table. However, the key of the **SpecialityCode** will only be **SpecCodeID**. This key will not include the key of **Doctor**

Agenda

1. Persistence in software engineering
2. **Design patterns**
3. System structure: Package diagrams
4. Implementation diagrams

Levels of reuse in software engineering

- **Code-level** - a new class is based on an existing, fully coded class.
- **Design level reuse**— a project bases all its new design on existing designs and available design patterns.
- **Analysis-level reuse**—can occur at an organizational level across multiple projects. Requirements of one project can be reused (with suitable modifications) in another project

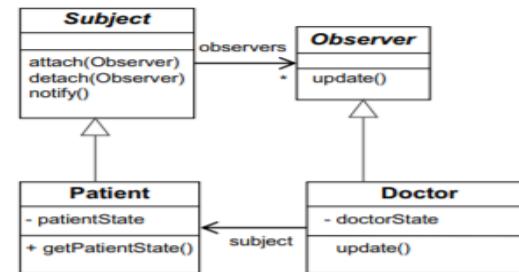
[a]

Code

```
public class Person {  
    public void setPersonDtls() { };  
    public void changePersonDtls() { };  
  
    private char Person_ID;  
    private date DateOfBirth;  
}  
  
-----  
  
public class Patient extends Person {  
    public Patient () { }  
    public void getPatientDtls () { }  
    public void calcPatientAge () { }  
    public void admitPatient () { }  
  
    private char PatientDtls;  
    private bool Admitted;  
}
```

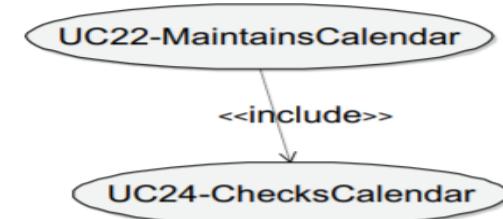
[b]

Design



[c]

Analysis

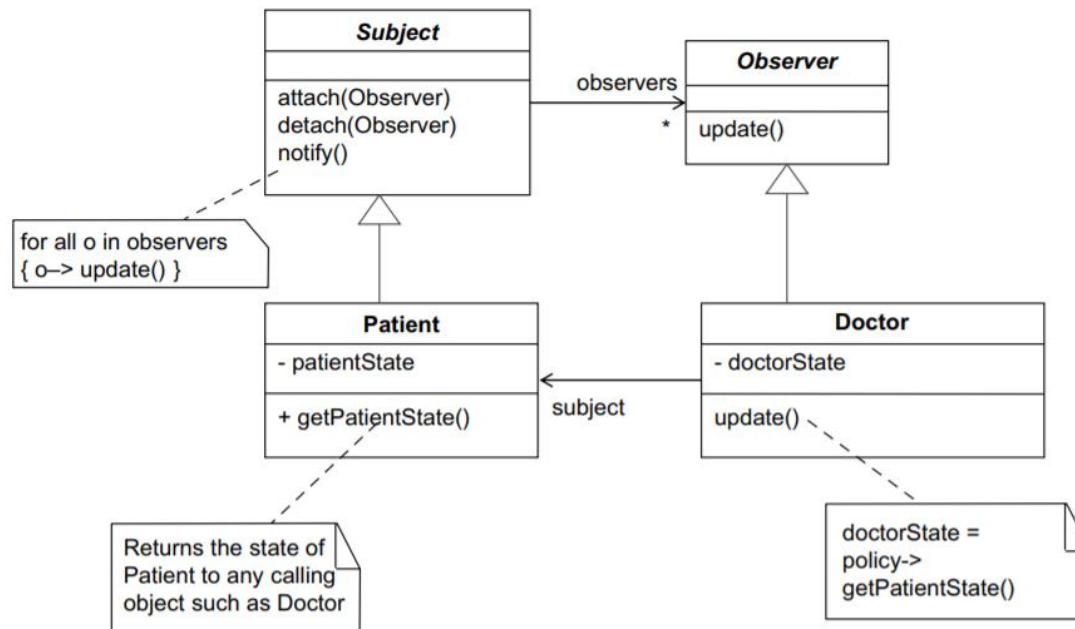


Design patterns in software engineering

- “Higher level” abstractions that can be reused in newer designs
- One of the most popular approaches to reusing software and designs, especially in the context of object orientation
- A pattern can be described by the following four essential elements:
 - The **pattern name** describes the design problem, its solutions, and consequences, in a word or two.
 - The **problem** describes when to apply the pattern.
 - The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.
 - The **consequences** are the results and trade-offs of applying the pattern
- Once described and documented in a **catalog**, design patterns can facilitate reuse through efficient and flexible designs.

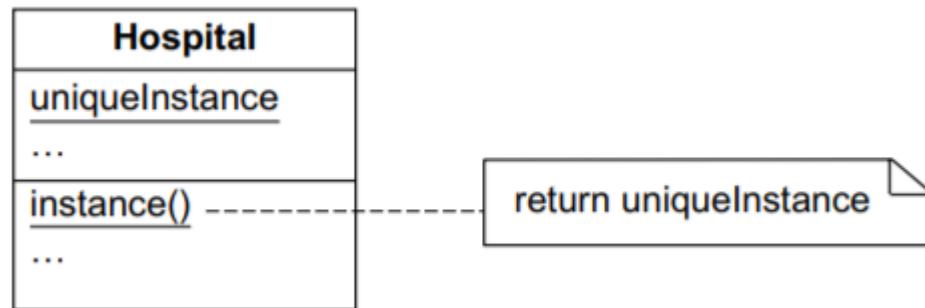
Example: Observer pattern applied

- They capture the essence of a situation where one class (**Observer**) depends on another class (**Subject**). Any change in the **Subject's** state influences the **Observer**.
- The abstract classes representing **Observer** and **Subject** are specialized into **concreteObserver** (**Doctor**) and **concreteSubject** (**Patient**).
- Changes in the state of **Patient** (say, *InTreatment patient*) affect the way the **Patient** is treated by the **Doctor**



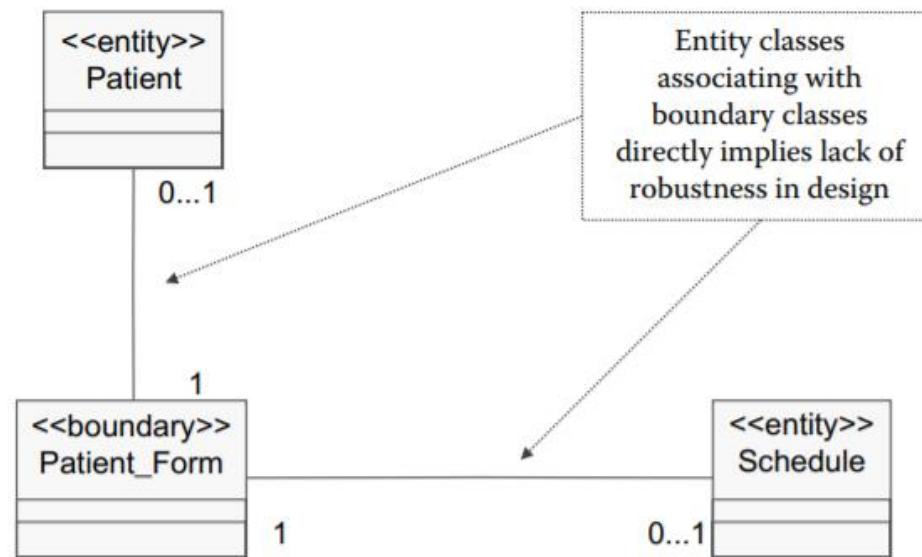
Example: Singleton pattern applied

- The **Singleton** pattern facilitates design creation wherein classes based on this pattern can be instantiated only once. Thus, Singleton patterns result in global objects that are encapsulated and can be checked for the existence of only one object.
- The **Hospital** object is unique, as only one instance of it is required when the system is executed



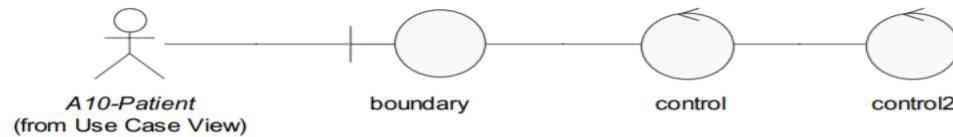
Robustness in design

- Robustness in design is a concept that explores dependencies (coupling) between classes.
- The higher the dependency of classes on one another in a system, the less robust the system is. If the classes are less dependent on one another, the system is said to be robust.
- Robustness is a specific design construct in which the controller (or manager) class is inserted between entity, boundary, and database classes.



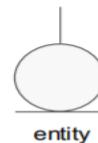
MVC pattern and robustness

- The concept of robustness derives from an earlier, well-known pattern called **MVC*(model view controller)**.
- In MVC, the model (or <<entity>> stereotypes in UML-based designs) and the view (or <<boundary>> stereotypes) are separated by the controller (or <<controller>> stereotype).
- **Robustness rules**
 - Boundary (interface) classes cannot talk to (associate with) each other
 - Entity classes cannot talk to (associate with) each other
 - Boundary and entity classes can talk to control classes
 - Control classes can talk to (associate with) each other

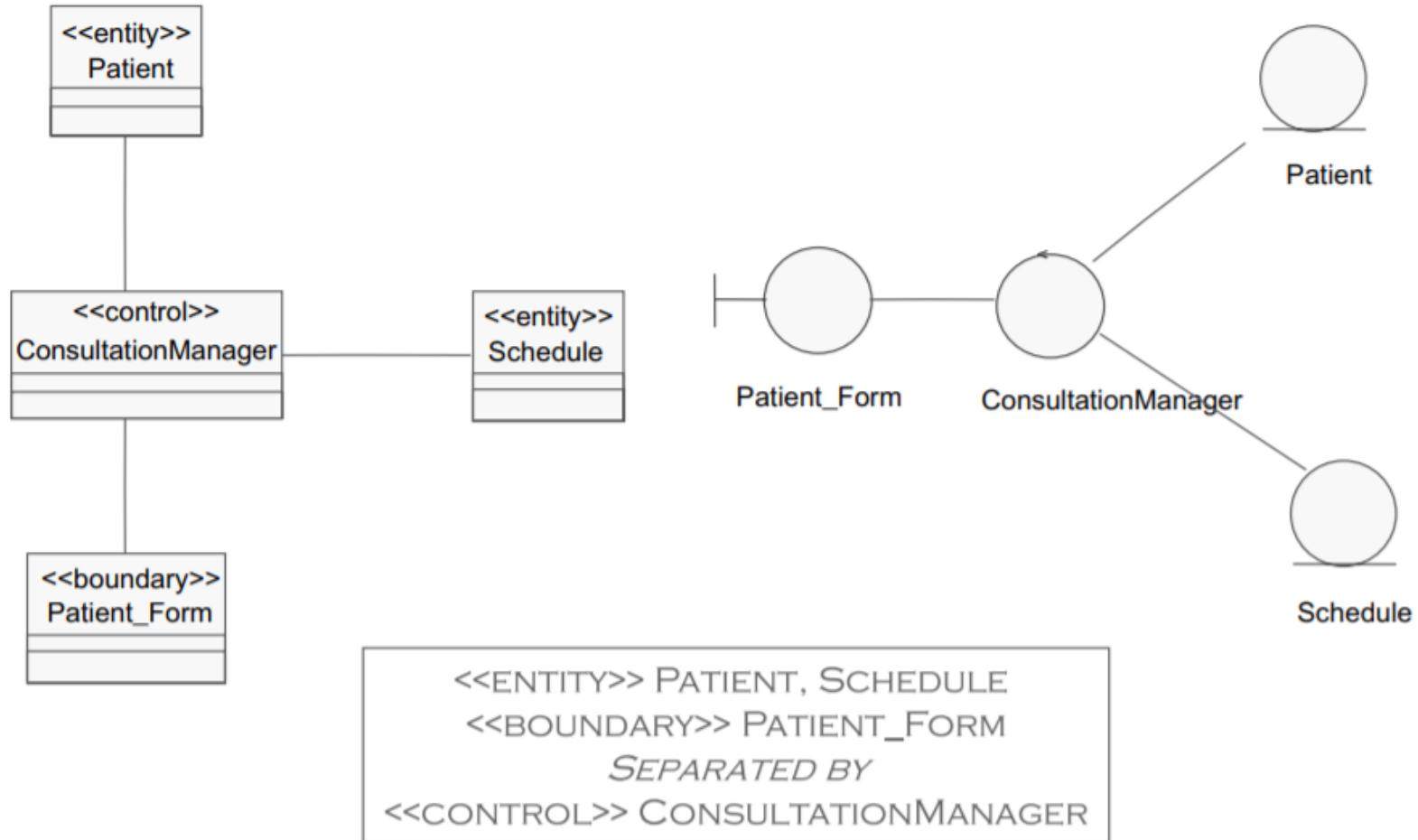


Allowable Connections to ensure Robustness

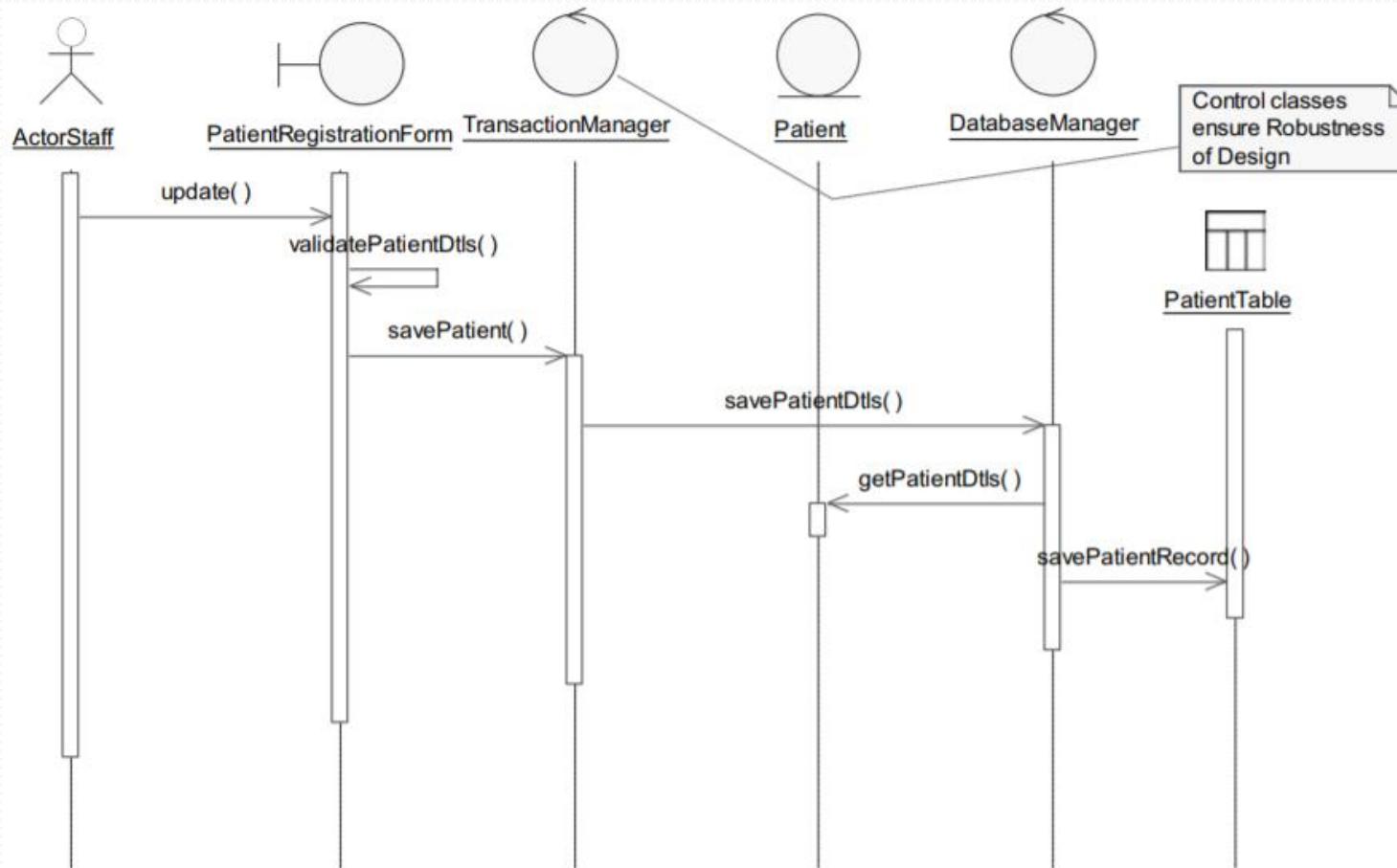
actor - boundary
boundary - controller
controller - entity
controller - controller



Example: robustness in class diagram



Example: robustness in sequence diagram



Agenda

1. Persistence in software engineering
2. Design patterns
- 3. System structure: Package diagrams**
4. Implementation diagrams

Package diagram

- Elements of the diagrams can be grouped into **packages**.
- A package is a mechanism intended for general purposes, which organizes the elements in groups.
- A package may include other packages, classes, use cases, collaboration etc.
- A package shows only the structures it contains, not the behavior of its elements.
- A modeling element belongs to a single package, but other packages can view this element. If the content of the package is explicitly shown than the package name is written on its **label**.
- Each package has a **name** that can be simple or can include the path



Package—
represents a subsystem



Dependency—
optional

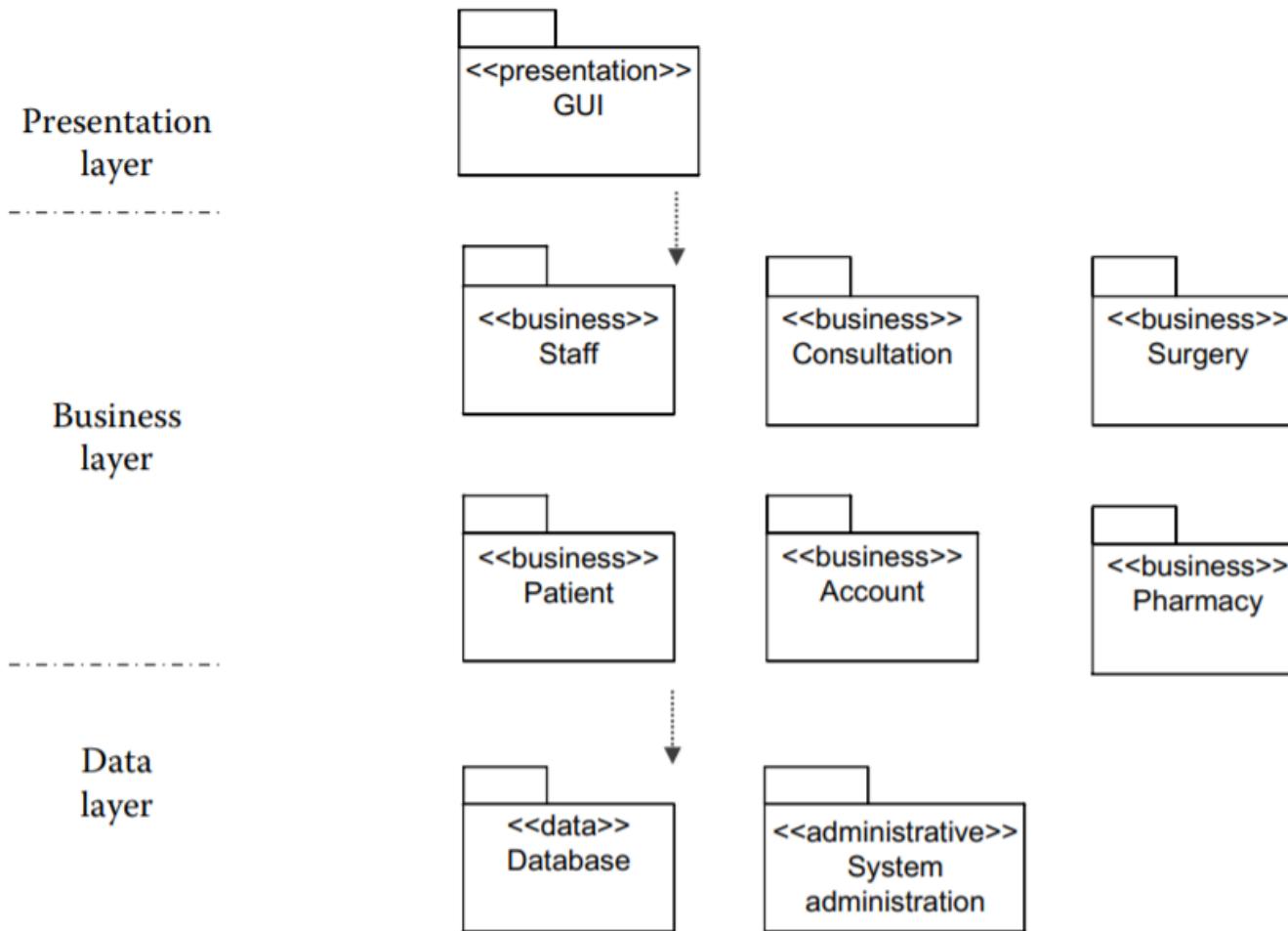


Note—
clarifies the diagram

Package diagram

- Packages are named after the subsystems or large area of work they represent.
- They are then prioritized. This provides a basis for scheduling their development.
- In terms of **visibility**, packages behave like classes.
- UML defines five stereotypes that apply to packages:
 - ***facade*** - A package that is only a view of another package;
 - ***framework*** - A package that contains mainly patterns;
 - ***stub*** - Is a proxy for the public content of another package;
 - ***subsystem*** - A package that is an independent part of the modeled system;
 - ***system*** - A package that represents the whole model system

Example of package diagram



Agenda

1. Persistence in software engineering
2. Design patterns
3. System structure: Package diagrams
4. **Implementation diagrams**
 - Component diagram = software architecture
 - Deployment diagram = hardware architecture

Component diagram

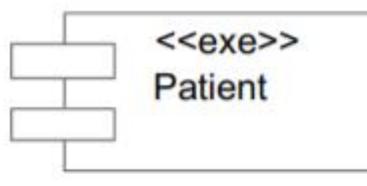
- Models ***overall system architecture*** and the ***logical components within it.***
 - Logical, reusable, and transportable system **components** that define the system architecture.
 - Well-defined interfaces, or public methods, that can be accessed by other programs or external devices: the **application program interface (API)**
- A **component** is an executable module or program (source code, binary code, dll, executable, script etc) and it consists of all the classes that are compiled into a single entity.

Types of components

- **Design time**—these components are made up of a collection of classes that are already put together in a design that can be readily used.
- **Link time**—these components are ready to use libraries that can be directly inserted in the current system design by linking to them.
- **Runtime**—these components are executables that can be used to provide some service to the system.
- **Distributed (DCOM/CORBA) components**—facilitate interaction across dispersed software systems.
- **Service-oriented components**—are offered as a service

Component diagram – component symbol

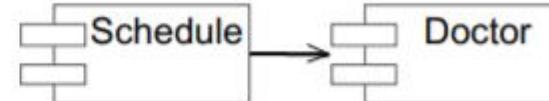
- Usually, the name of a component is the name of the file represented by the component.
- Objects implemented by a component instance are represented graphically inside the component instance symbol.



Component



Dependency



Schedule → Doctor



Interface

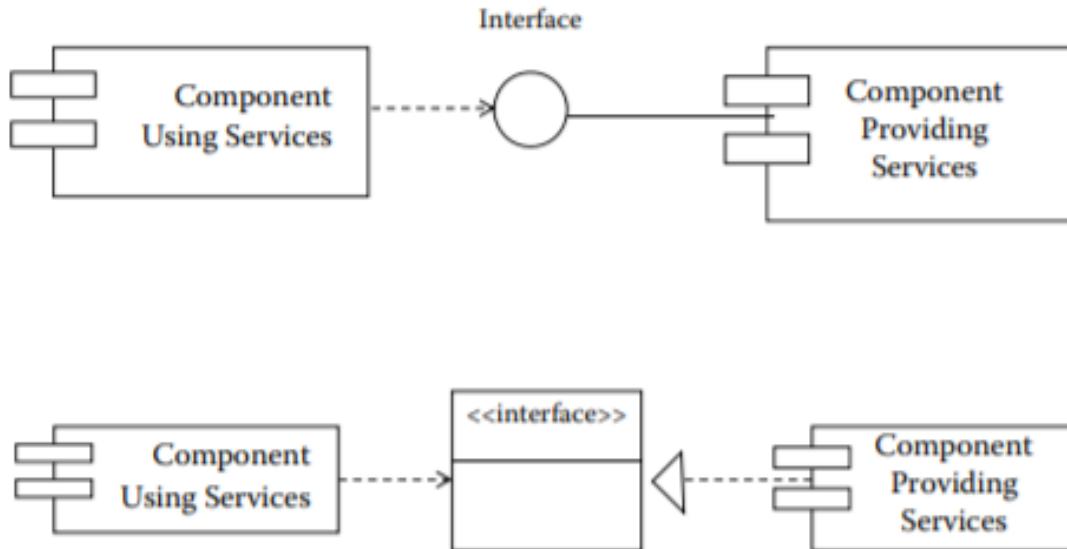
This is an explanatory note

Notes

A Patient component will become Patient.EXE or Patient.DLL

Component diagram: interfaces

- Components relate to each other in practice through interfaces
- There can be multiple interface classes for a component and multiple relationships with the interfaces
- The interface contains a named set of operations that characterize the behavior of an element



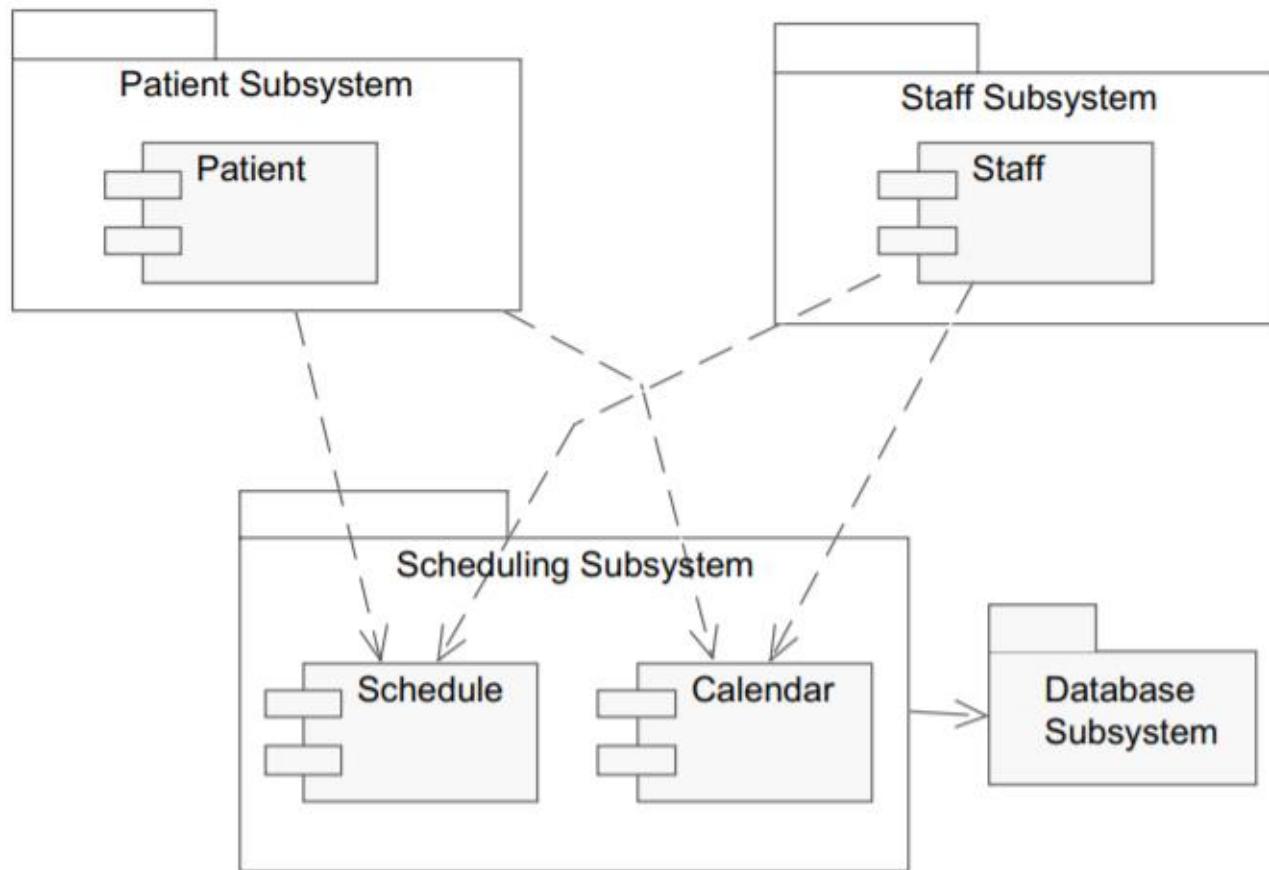
Relationships in component diagram

- **Dependency**
 - interrupted line targeting the provider component
 - classes included in the client component can **inherit, instantiate or use classes** included in the provider component.
 - may also be relationships of dependence between **components** and **interfaces of other components**,
- **Composition** relationships (components physically included in other components).

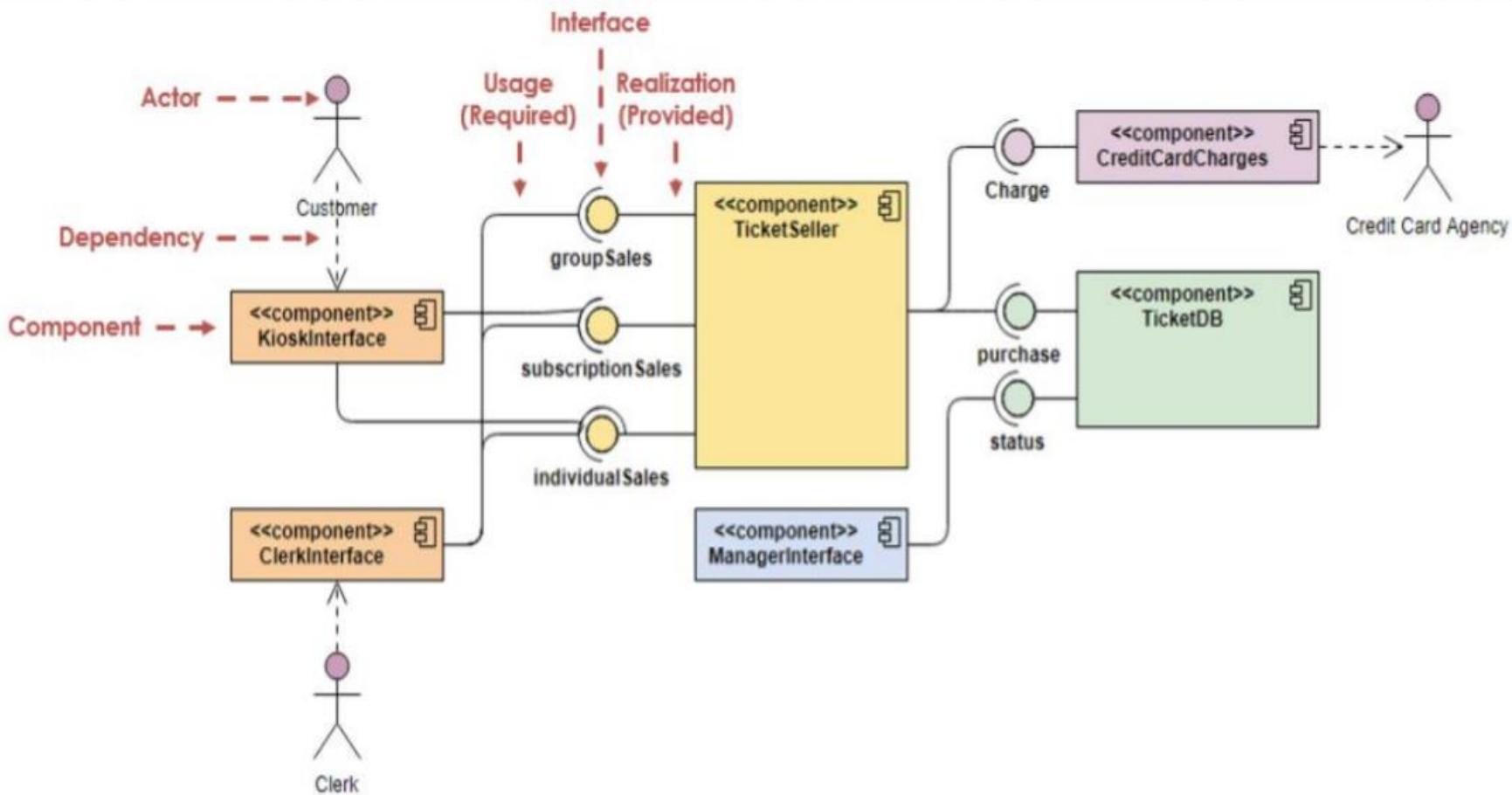
Components categories (stereotypes)

- **Application components** deal with a collection of classes containing business logic in them.
- **Database components** deal with the storage and retrieval of data.
- **Security components** enable provision of security-related interfaces that can be directly incorporated into a design.
- **GUI components** provide interface standards and partial implementation.
- **Printer components** enable faster implementation of printer classes by making printing routines available.
- **Utility components** support component-based development by providing utility support, such as dates, math, and rate calculations.
- **Analytical components** provide data analytics such as from Big Data.
- **Internet of Things components** encapsulate analytics within devices.

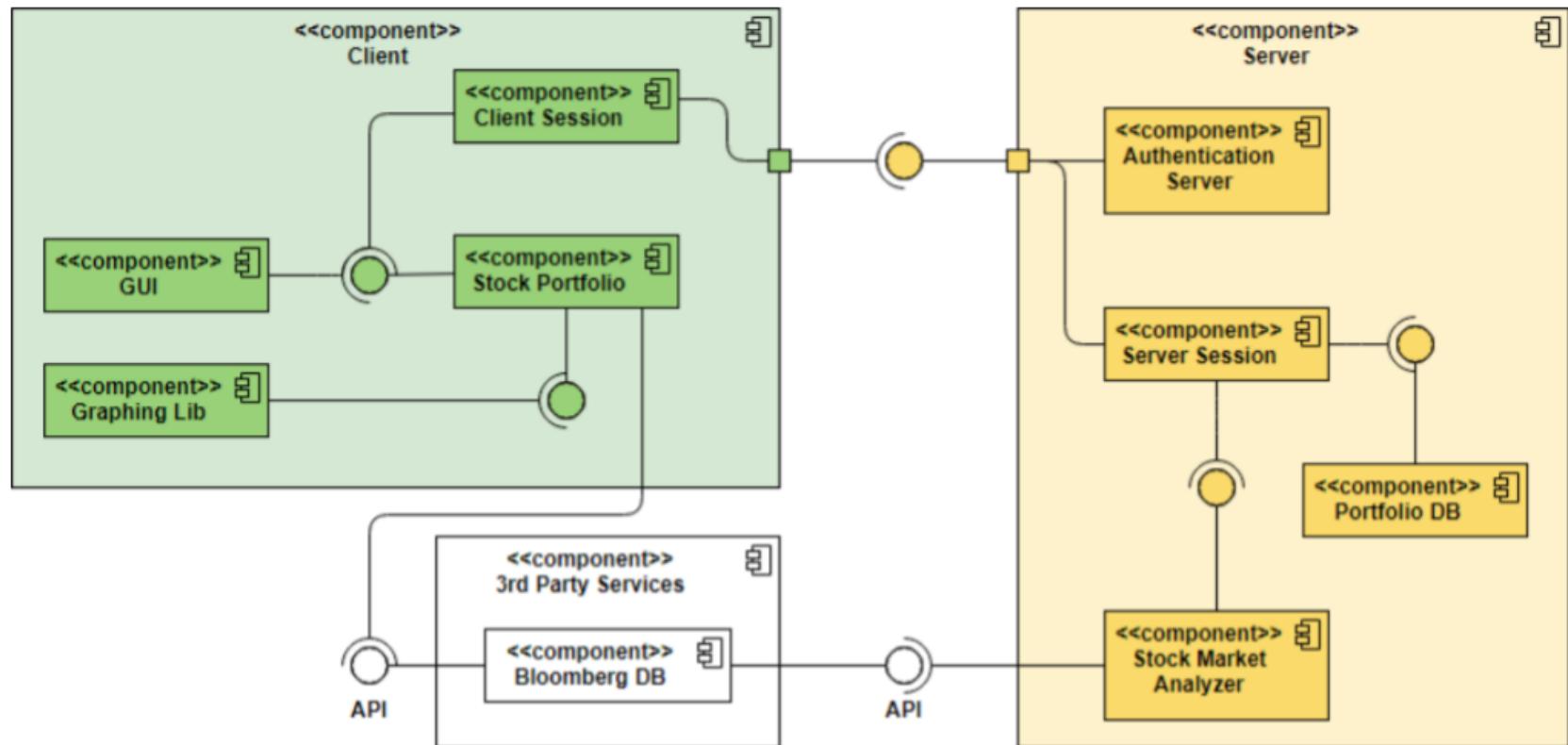
Interdependencies and packages



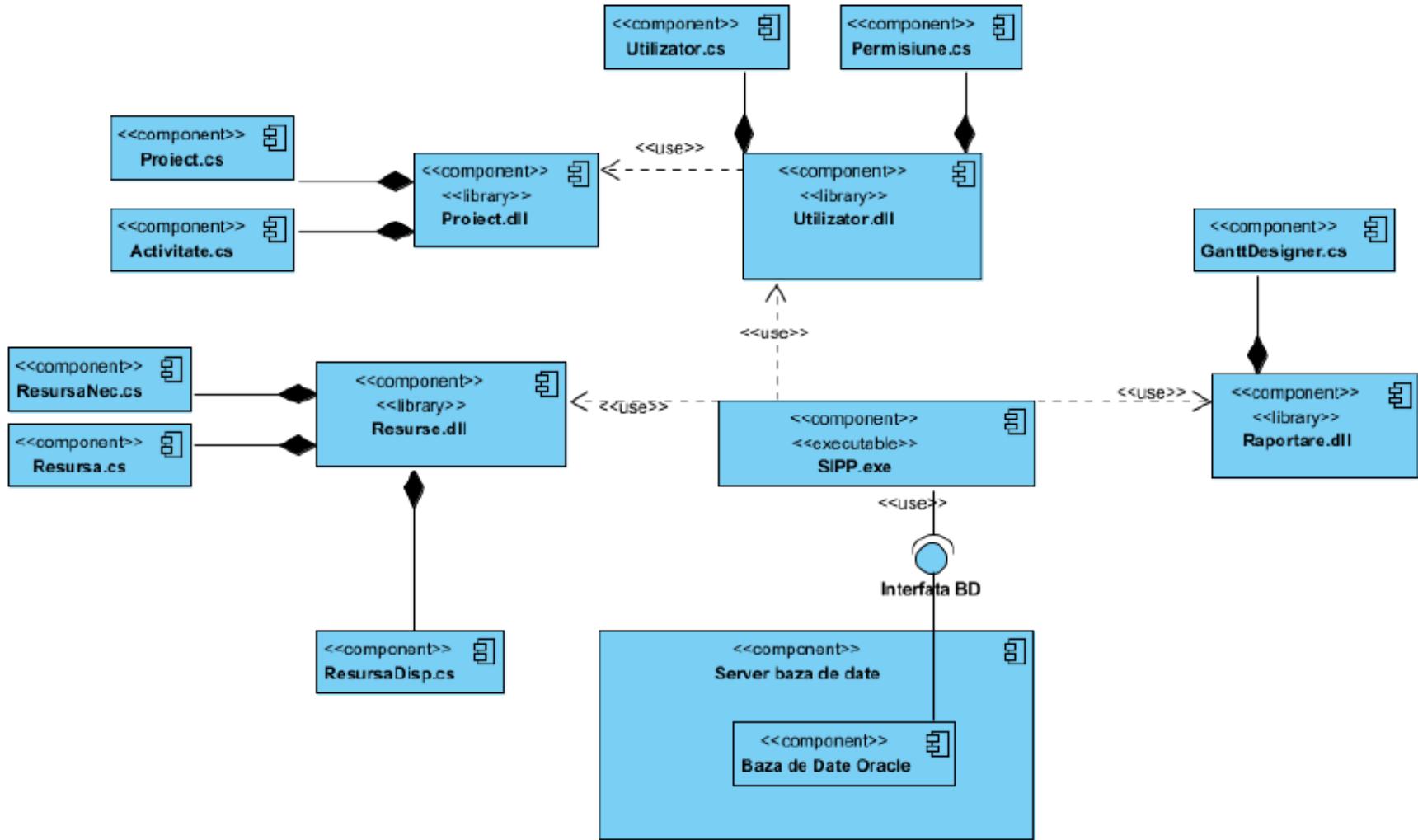
Example 1: Ticket Selling System



Example 2: client-server app



Component diagram example

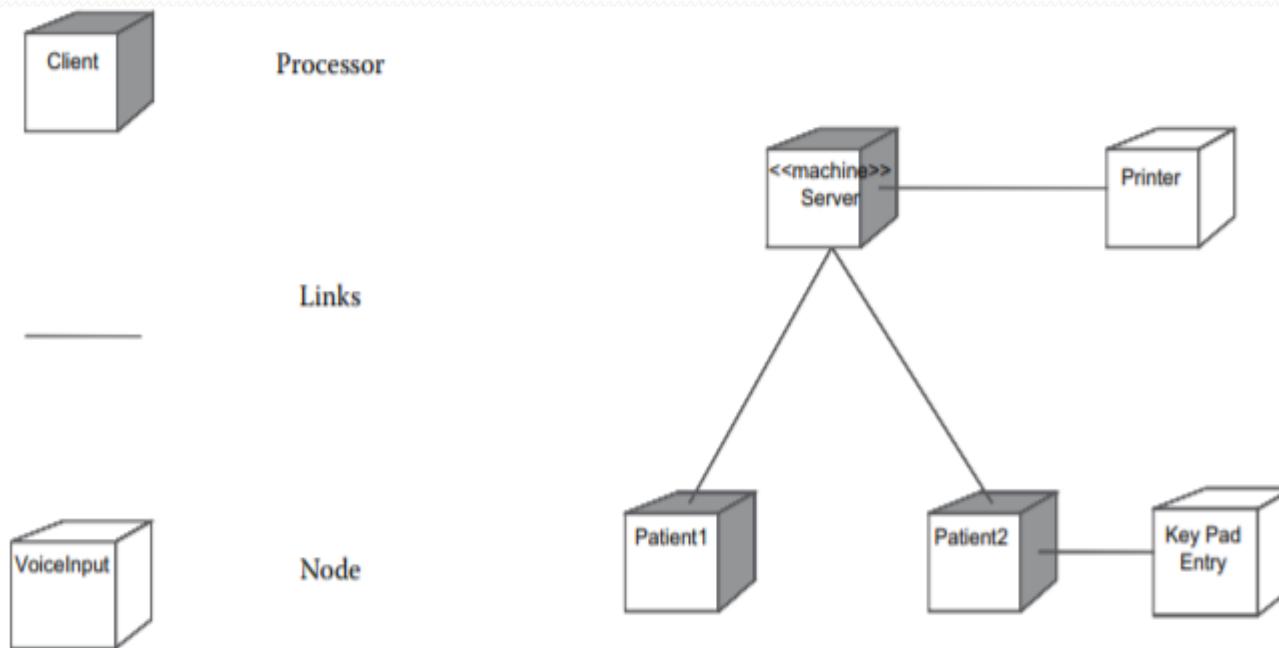


Deployment diagram

- Deployment diagrams show the **configuration** of the **processing elements** during execution and the components, processes and objects they contain.
- A deployment diagram is a graph of **nodes** connected through communication **associations**.
- A **node** is a physical entity that is a processing resource with memory and processing capabilities (computing devices, human resources, mechanical processing resources).
- A node is graphically represented by a **parallelepiped**.
- A node type has a **name** associated, and an instance of a node has (optionally) an instance name and a type name (**instance name: type name**).
- An association between two nodes indicates the existence of a communication path between nodes.

Notations in deployment diagrams

- It is an instance-level diagram, the processors shown are real instances of the system.
- It is not practically possible to show all the hardware elements of a system in this one diagram.

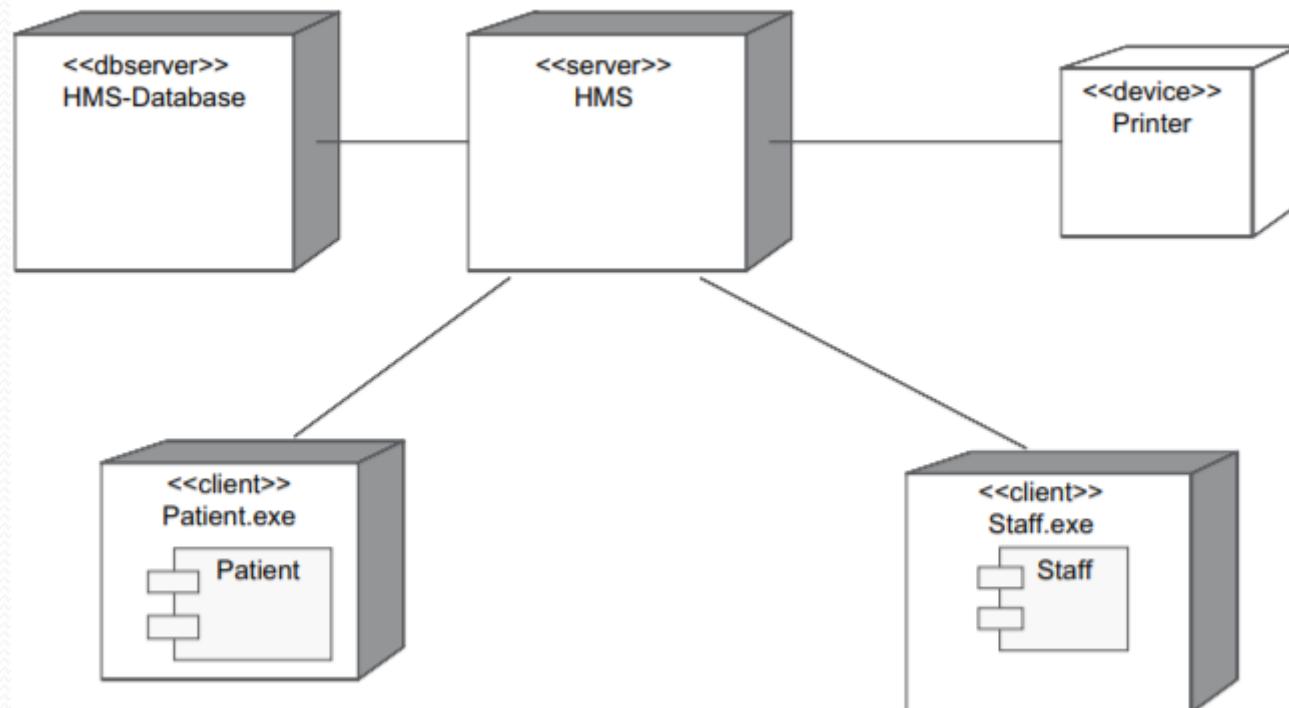


Deployment diagram

- Deployment diagrams contain two types of nodes:
Execution environments and devices.
 - **Execution environments** are hardware components capable of running programs.
 - **Devices** are non-computing hardware components. The name associated with a device is generic (for example printer, modem, terminal etc.)
- A **connection** is a physical link (generally bidirectional) between two devices or processors.

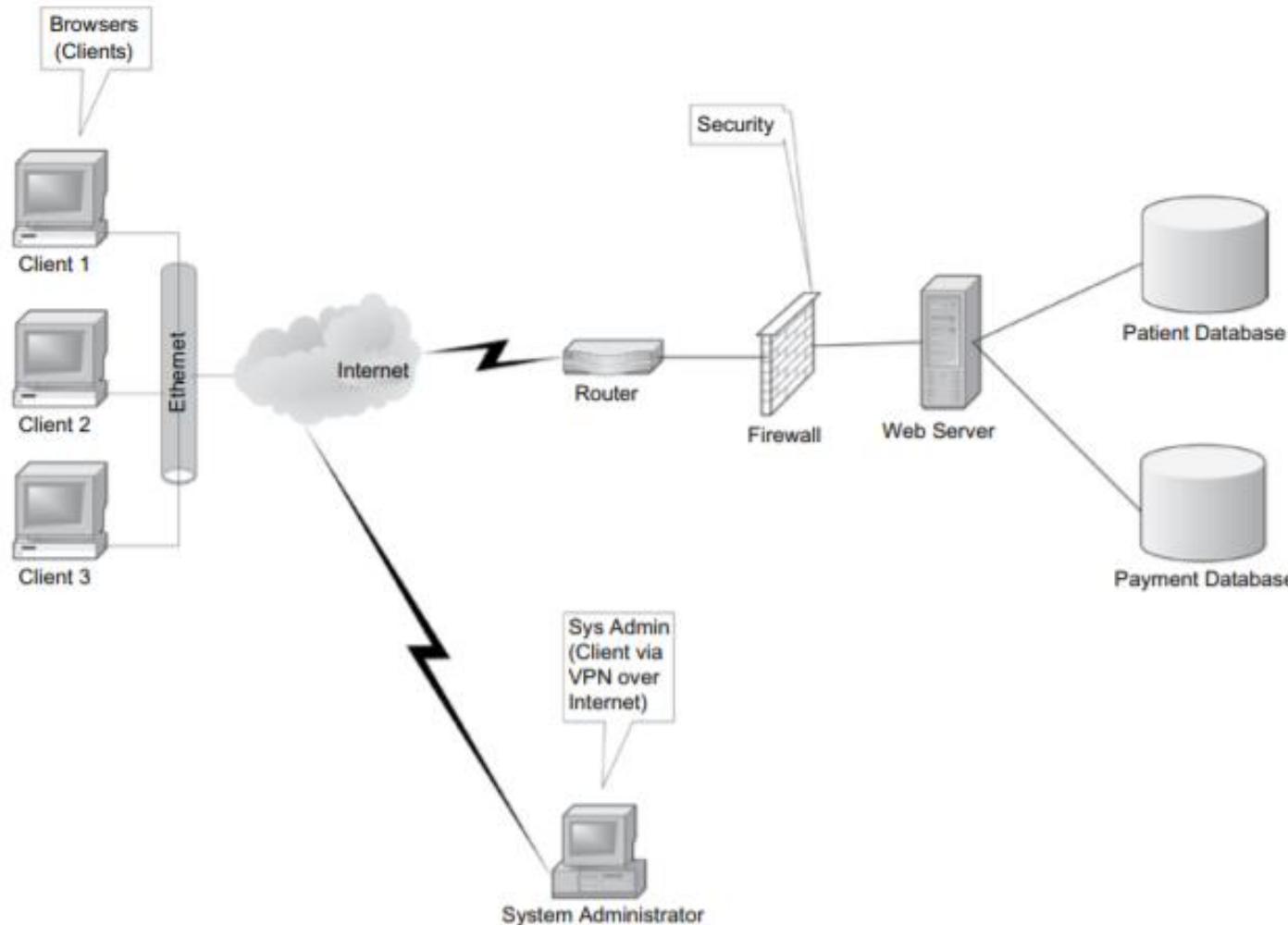
Deployment diagram

- Deployment diagrams can be used to represent **components** that belong to certain nodes by embedding the component symbol within the symbol representing the node.
- There may also be **dependency relationships** between components.



Deployment diagram – another example

- Elements are represented by their actual icons



Deployment diagram example

