

Object Oriented Analysis and Design

LE Thi My Hanh

Faculty of Information and Technology

Da Nang University of Technology

Email : ltmhanh@dut.udn.vn

Mobile: 0905737577

Thông tin học phần

- Tên học phần: Phân tích & Thiết kế hướng đối tượng
- Tên tiếng Anh: Object-Oriented Analysis and Design
- Số tín chỉ: 02 tín chỉ

Mục tiêu học phần

- Học phần nhằm cung cấp cho sinh viên kiến thức cơ bản về các hoạt động phân tích và thiết kế phần mềm hướng đối tượng.
- Học phần giúp sinh viên có thể áp dụng phân tích và thiết kế các hệ thống phần mềm thực tế.
- Học phần Phân tích và thiết kế hướng đối tượng thuộc khối kiến thức chung của ngành CNTT và được giảng dạy sau khi sinh viên đã học về Lập trình hướng đối tượng và các học phần cơ sở khác của ngành Công nghệ Thông tin.

Đánh giá

- Đánh giá quá trình: 20%
 - Chuyên cần: điểm danh 30%
 - Bài tập: bài tập cá nhân & bài tập nhóm – 70% (thảo luận, trình bày)
- Đánh giá giữa kỳ: Trắc nghiệm (30-40 câu/10 điểm) - 20%
- Đánh giá cuối kỳ: Trắc nghiệm (40-50 câu/10 điểm) – 60%

Tài liệu tham khảo

Giáo trình

- [1] Dương Anh Đức, *Phân tích & thiết kế hướng đối tượng bằng UML*, NXB Thống kê, 2005.
- [2] Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Third Edition, Addison-Wesley, 2004.

Tài liệu khác

- [1] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Third Edition, Addison Wesley, 2013.
- [2] Sinan Si Alhir, *Learning UML*, Addison Wesley, 2003.
- [3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley Pub Co, 1995.
- [4] Nguyễn Văn Ba, *Phát triển hướng đối tượng với UML 2.0 và C++*, NXB Đại học Quốc gia Hà nội, 2005.
- [5] www.uml-diagrams.org

Kế hoạch học tập (1)

- Introduction to object-oriented concepts
 - Objects, classes
 - Encapsulation, inheritance, polymorphism
- An overview of UML
 - Modelling concept
 - Object-oriented modelling methods
 - History of UML
 - Basic elements of UML
- UML and Software development processes
- Requirements modelling
 - Use-case diagrams
- Static structure modelling
 - Class diagram
 - Object diagram

Kế hoạch học tập (2)

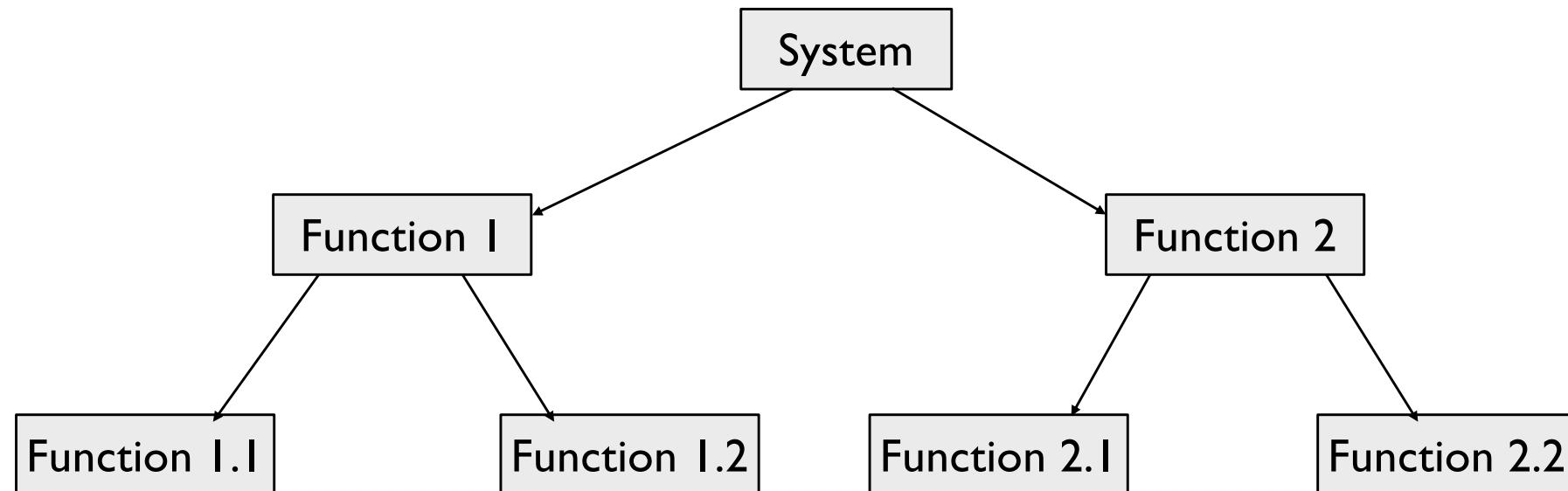
- Dynamic behaviour modelling
 - Activities diagrams
 - State diagrams
 - Interaction diagrams
- Architecture modelling
 - Package diagrams
 - Component diagrams
 - Deployment diagrams
- Design principles
 - GRASP: assignment of responsibilities
- Implementation
 - Remind of object-oriented programming
- Case study
- Conclusions

Introduction to object-oriented concepts

- Functional approach
- Object-oriented approach
- Object-oriented concepts
 - Objects
 - Classes
 - Encapsulation
 - Inheritance
 - Polymorphism
 - Abstraction

Functional/procedural approach

- Based on specified functions of the system
 - A system consists of several functions
- Decomposition of functions into sub-functions
 - A system consists of sub-systems
 - A sub-system is divided into smaller subsystems



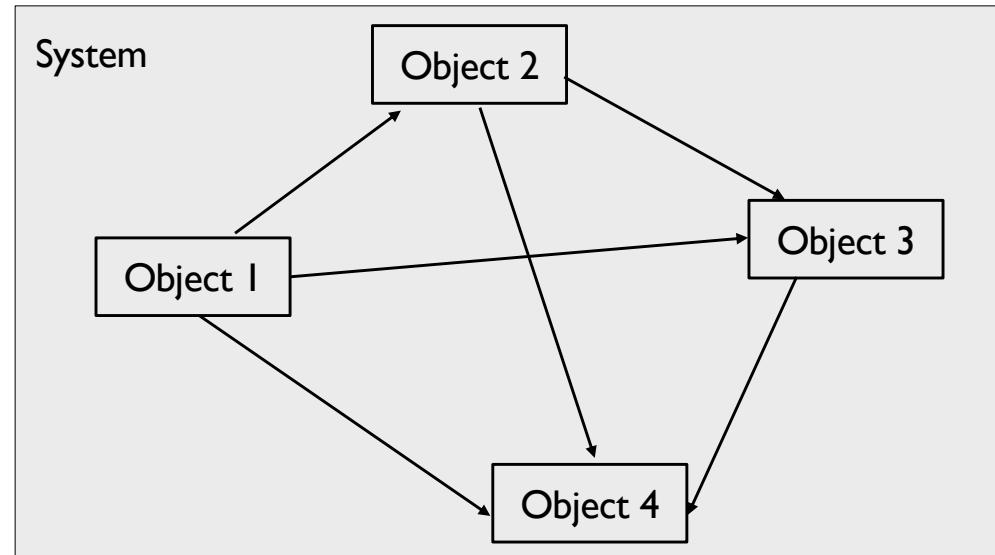
- Functions communicate using shared data or transfer of parameters

Functional approach

- Advantages
 - Easy to apply
 - Work well when data are simple
 - Help to reduce complexity
 - Obtain expected results
- Disadvantages
 - Functions are separated from data
 - Structure of the system is defined based on the functions, therefore a change of functions will cause difficulties in change of the structure
 - The system is weakly open
 - Difficult to re-use
 - A significant maintenance cost

Object-oriented approaches

- The solution of a problem is organized around the concept of objects
- The object is an abstraction of data also containing functions
- A system consists of objects and relationships between them
- Objects communicate by exchanging messages to perform a task
- No global variables
- Encapsulation
- Inheritance

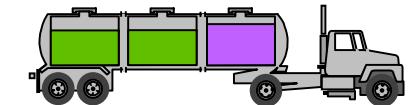


Object-oriented approaches

- Advantages
 - Very close to the real world
 - Easy to reuse
 - Hide information (encapsulation)
 - Lower development cost (inheritance)
 - Suitable for complex systems
- Functional approach v.s. object-oriented approach
 - Functional approach
 - System = algorithms + data structures
 - Object-oriented approaches
 - System = Σ objects
 - Object = algorithms + data structures

Objects

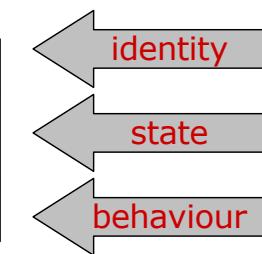
- Object is the concept describing an entity in the real world
- There are relationships between the objects
- Example
 - The Student "Micheal" is an object
 - The Student can't be an object !
- Object = state + behaviour + identity
 - State (data) describes the characteristics of an object at a given time, and is saved in the variables
 - The behaviour is expressed by the functions of the object
 - Each object has a unique identity
- Example



Xe tải



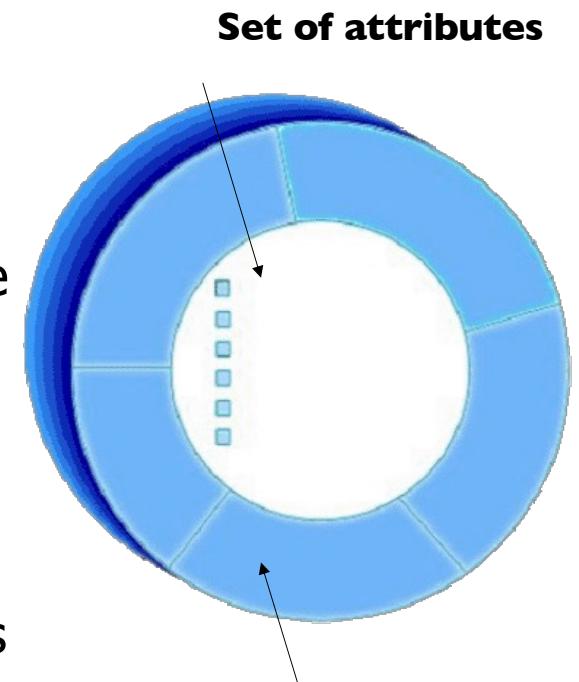
aRectangle
length = 2
width = 4
origin = aPoint
area()



Objects

- **State = Set of attributes**

- An attribute describes one property of the object
- At every moment, an attribute has a value in a specific set of attributes area
- Example
 - The car has properties: color, length, width, weight, number of kilometres, ...
 - A Renault 207 weighs 1300 pounds, it is red, ...



- **Behaviour = Set of functions**

- A function/method is the ability of the object to perform a task
- The behaviour depends on state
 - Example: A car can start the engine then run, ...



name: michael jackson
birth date: august 29, 1958
death date: june 25, 2009
place of birth: GARY, INDIANA
place of death: LOS ANGELES,
CALIFORNIA

Objects

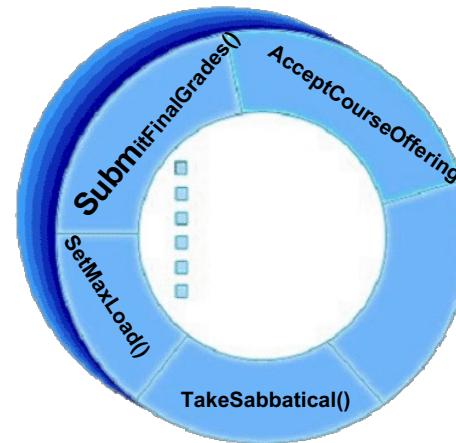


Name: J Clark
Employee ID: 567138
Date Hired: July 25, 1991
Status: Tenured
Discipline: Finance
Maximum Course Load: 3 classes

Giáo sư Clark

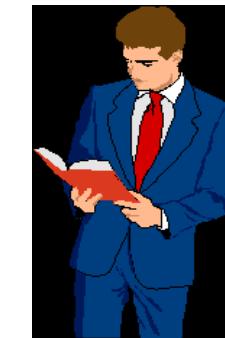


Giáo sư “J Clark”
dạy Sinh học



Các hành vi của giáo sư Clark
Submit Final Grades
Accept Course Offering
Take Sabbatical
Set Max Load

Giáo sư Clark



Giáo sư “J Clark”
dạy Sinh học

Objects

- Links
 - Between objects, there may be links
 - Example



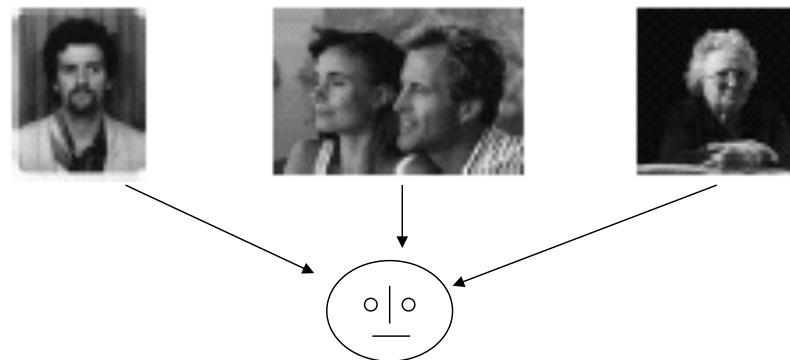
- Communication between objects
 - Send messages



- Message types
 - constructor
 - destructor
 - getter
 - setter
 - others

Classes

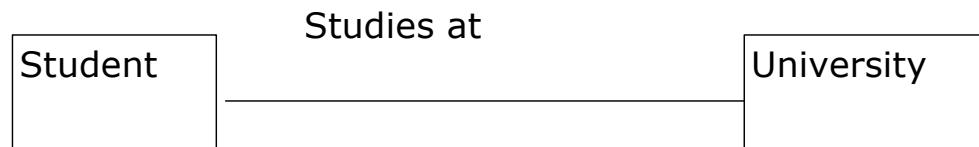
- A class is an abstract description of a set of objects having
 - similar properties
 - common behaviour
 - common relationship with other objects
- Class is an abstraction
 - Abstraction: search for common aspects and omit the differences



- Reduce the complexity

Class

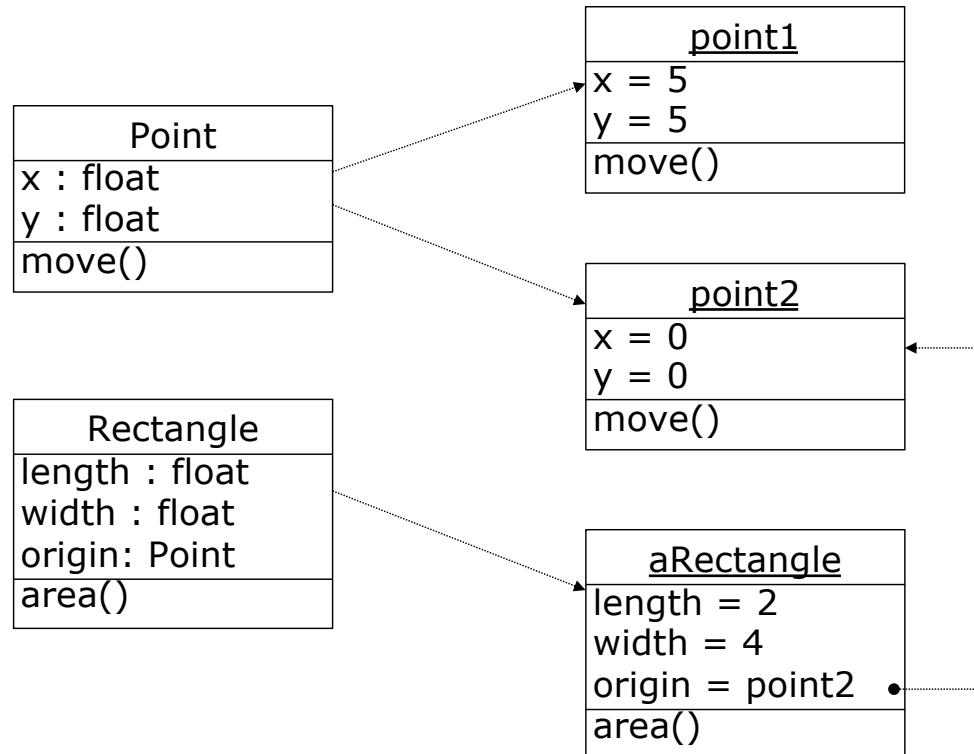
- Relationship
 - There may be relationship between classes
 - A relationship between classes is the set of links between their objects



- Class/Object
 - An object is an instance of a class
 - A value is an instance of an attribute
 - A link between objects is an instance of the relationship between classes

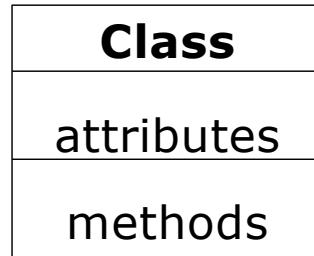
Classes

- Example: Class / Object



Encapsulation

- Data + Processing of data = Object
- Attributes + Methods = Class



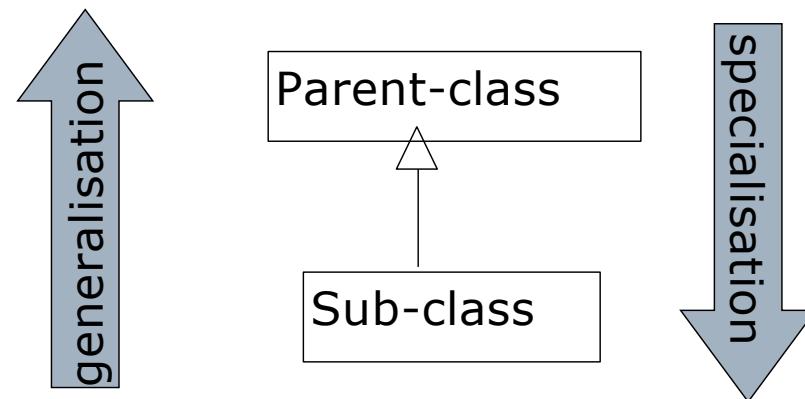
- The state of object is encapsulated by a set of attributes
- The behaviour is encapsulated by a set of methods
 - Users of an object know the messages that the object can receive (public methods)
 - The implementations of methods are hidden from external users

Encapsulation

- Advantages
 - Hide the information
 - Restrict access to the information from the exterior
 - Avoid the global changes in the whole system: the internal implementation can be modified without affecting the external users
 - Facilitate the modularity
 - Easy to reuse
 - Easy to maintain

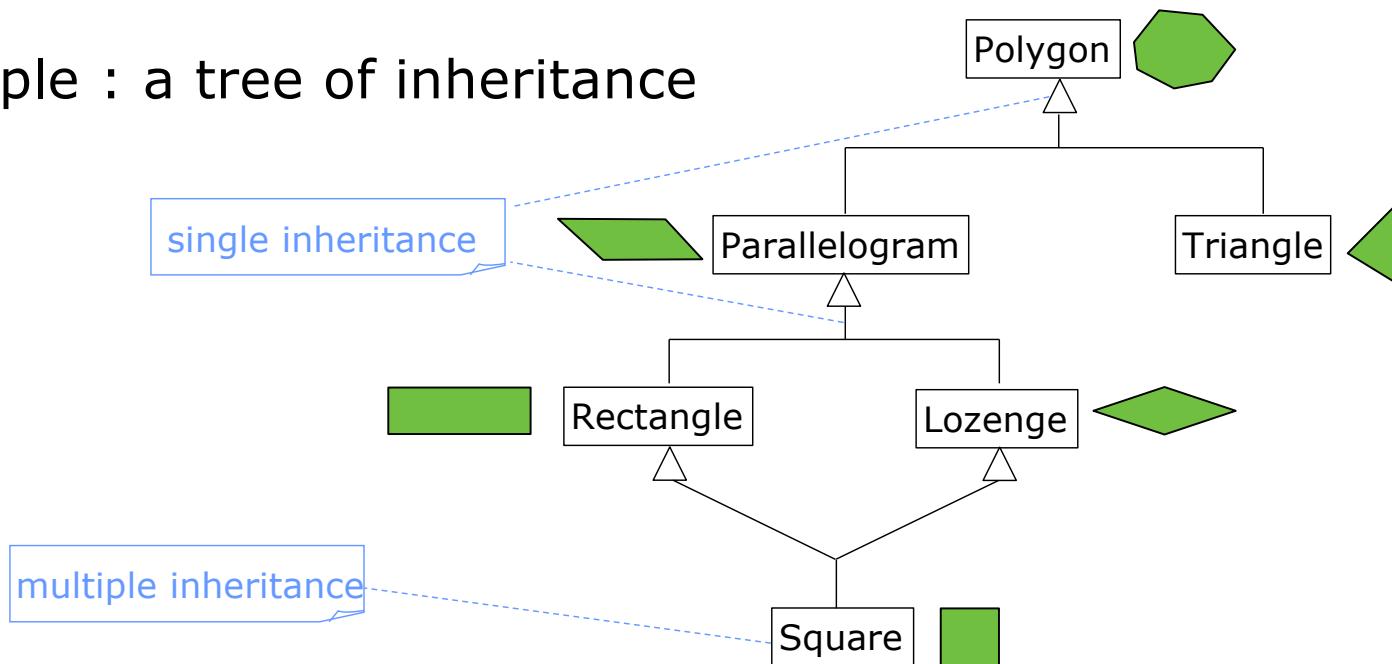
Inheritance

- Inheritance allows the reuse of the state and the behaviour of a class by other classes
- A class is derived from one or more classes by sharing attributes and methods
- Subclass inherits attributes and methods of parent-class
- Generalisation / Specialisation
 - Generalisation: common attributes of sub-classes are used to construct the parent-class
 - Specialisation: sub-classes are constructed from the parent-class by adding other attributes that are unique to them



Inheritance

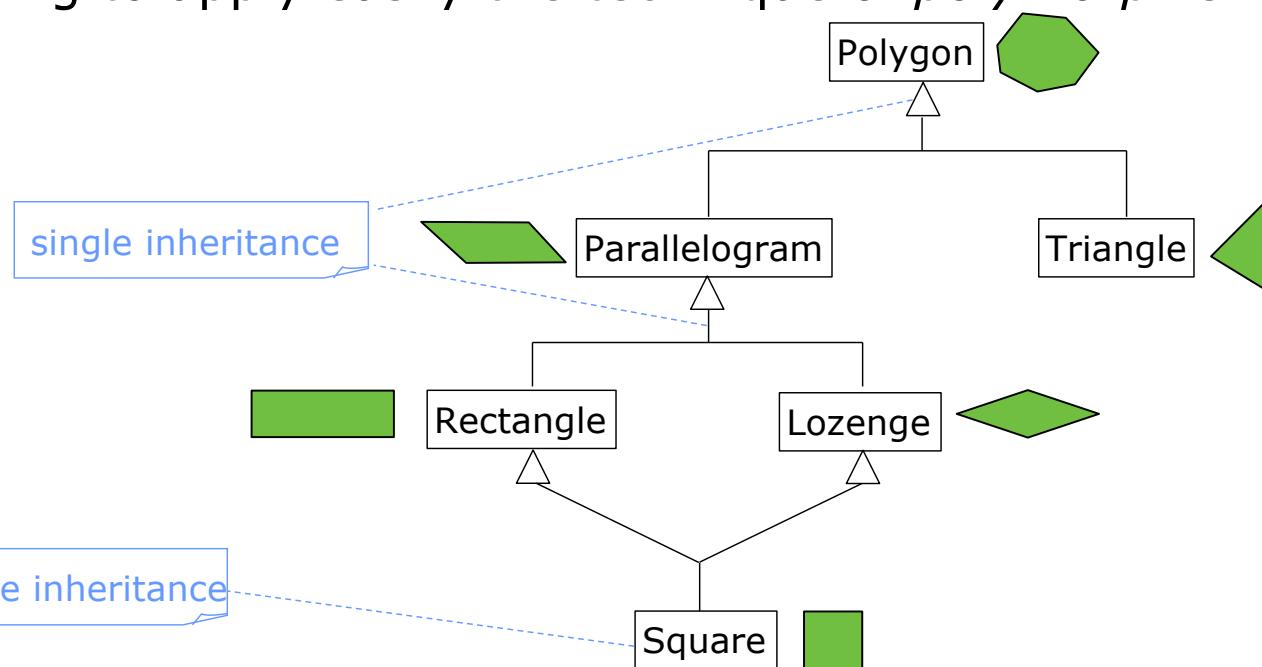
- **Single inheritance:** a sub-class inherits from only one parent-class
- **Multiple inheritance:** a sub-class inherits from multiple parent-classes
- Example : a tree of inheritance



- What is the difficulty of multiple inheritance?

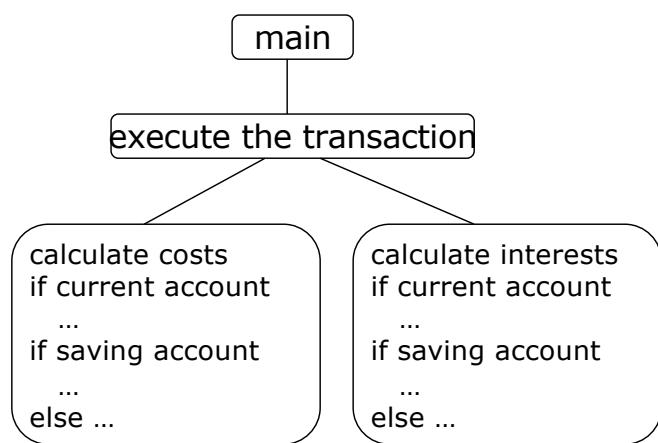
Inheritance

- Advantages
 - Organisation of classes
 - classes are organised hierarchically
 - facilitation of the management of classes
 - Construction of classes
 - sub-classes are constructed from parent-classes
 - Reduction of development cost by avoiding to re-write the code
 - Allowing to apply easily the technique of *polymorphism*

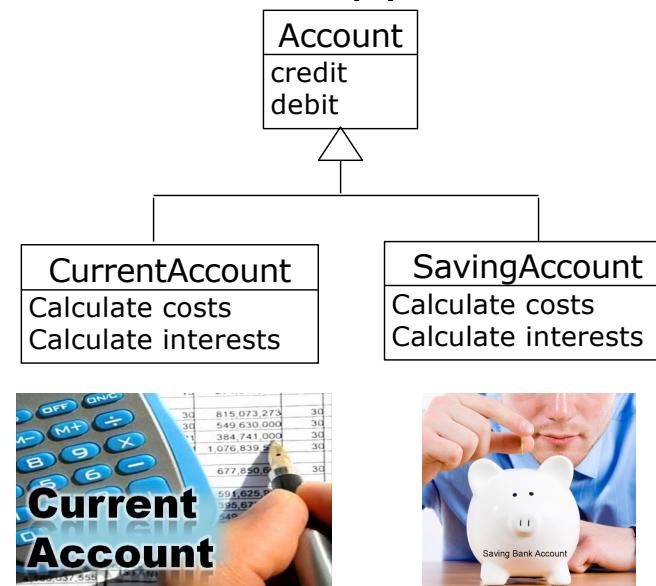


Polymorphism

- Polymorphism of methods
 - Different methods are capable of answering to a request
 - Methods having the same name are defined differently (different behaviours) in different classes
 - Sub-classes inherit the specification of methods from parent-class and these methods can be re-defined appropriately
 - Reducing the use of conditional statements (e.g., if-else, switch)
- Procedural approach *versus* Object-oriented approach

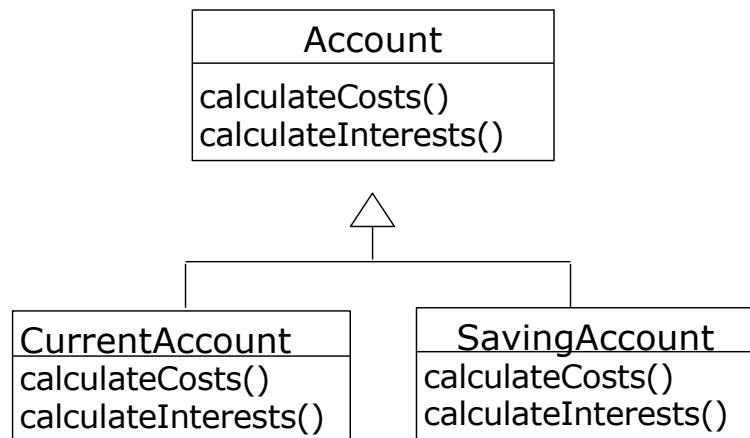


OOAD



Polymorphism: dynamic linking

- The method to be executed by an object depends on the class of the object: dynamic linking
- The dynamic linking is necessary when
 - A variable refers to an object whose class of membership is part of an inheritance tree
 - Several methods exist for the same message (name) in the inheritance tree (polymorphism)

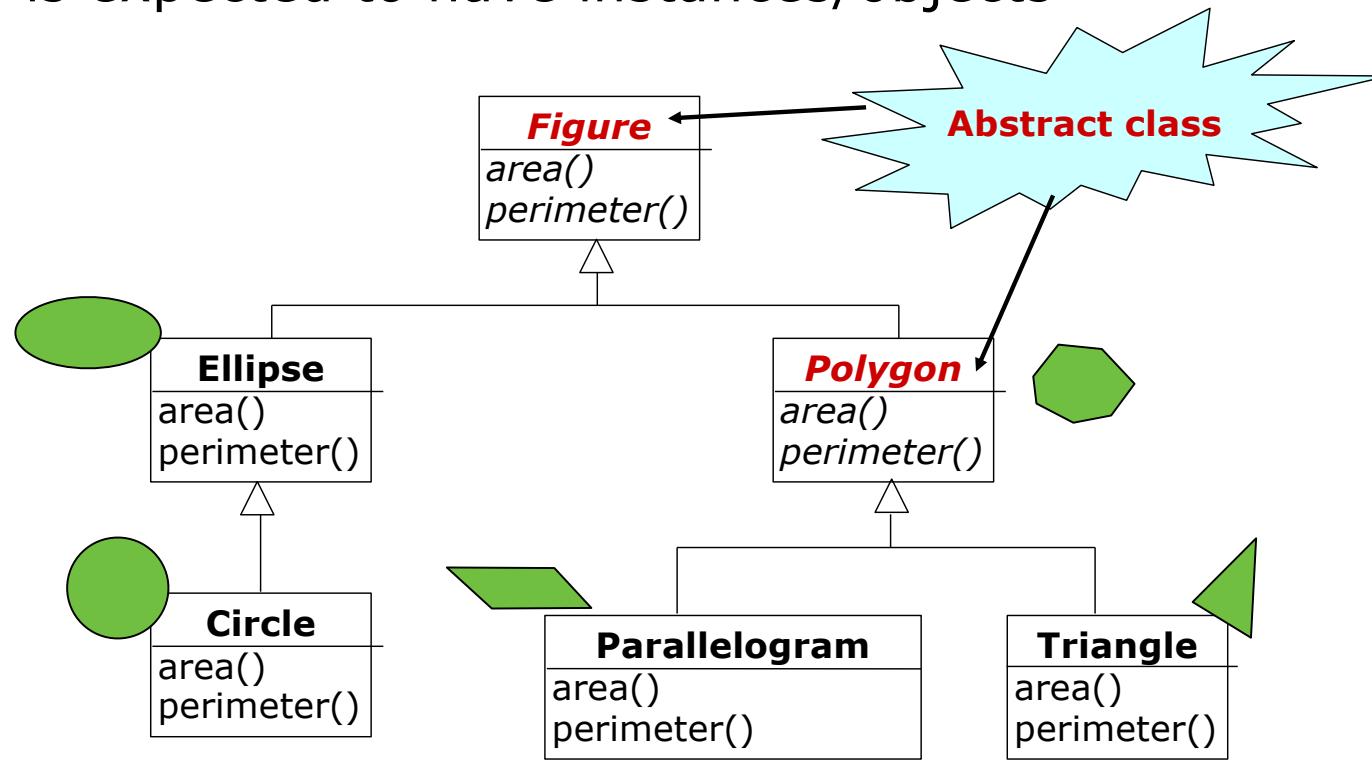


```
int calculateCost(Account accounts)
{
    int s = 0;
    for (int i = 0; i < accounts.length; i++)
        s = s + accounts[i]->calculateCosts();
    return s;
}

void main()
{
    Account accounts = new Account[2];
    accounts[0] = new CurrentAccount();
    accounts[1] = new SavingAccount();
    int s = calculateCost(accounts);
    ...
}
```

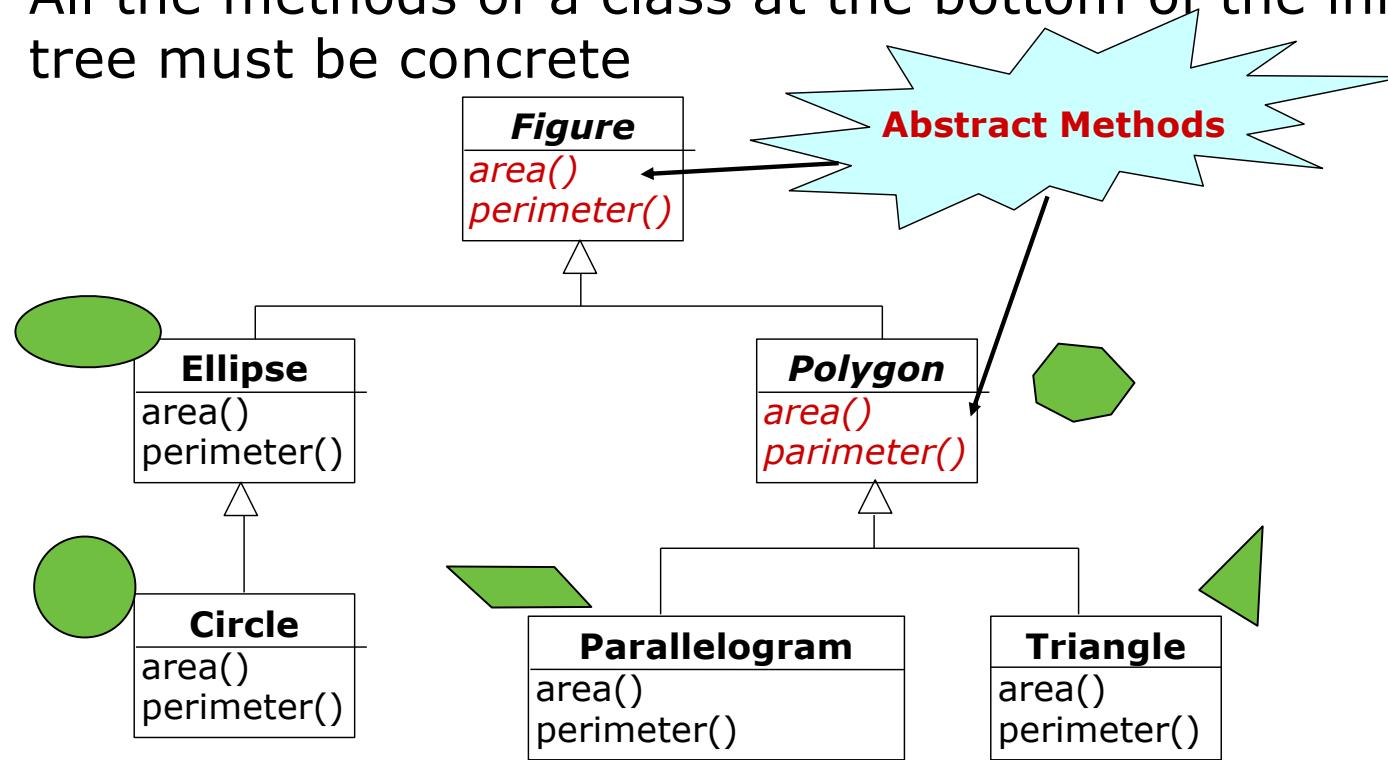
Abstraction: abstract class

- An abstract class
 - indicates the common characteristics of the sub-classes
 - can't have instances/objects
- A concrete class
 - contains a complete characterization of real-world objects
 - is expected to have instances/objects



Abstraction: abstract method

- A method should be defined at the highest possible abstraction level
 - At this level, the method can be abstract (i.e., no implementation)
 - In this case, the class is also abstract
 - If a class has an abstract method, at least one of its subclasses must implement this method
- All the methods of a class at the bottom of the inheritance tree must be concrete

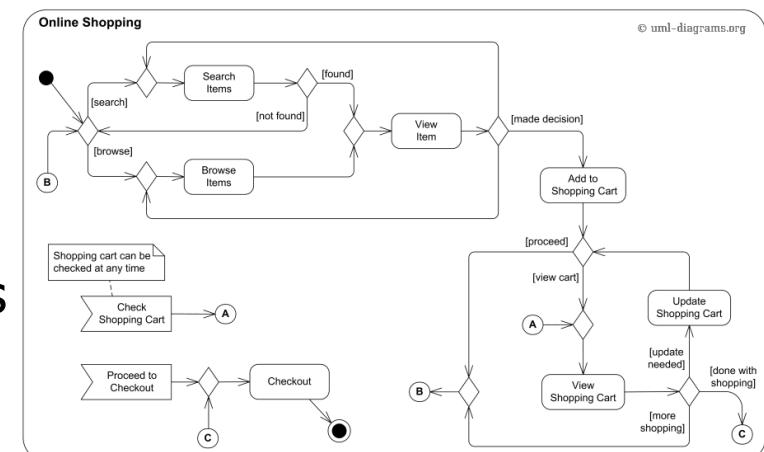


An overview of UML

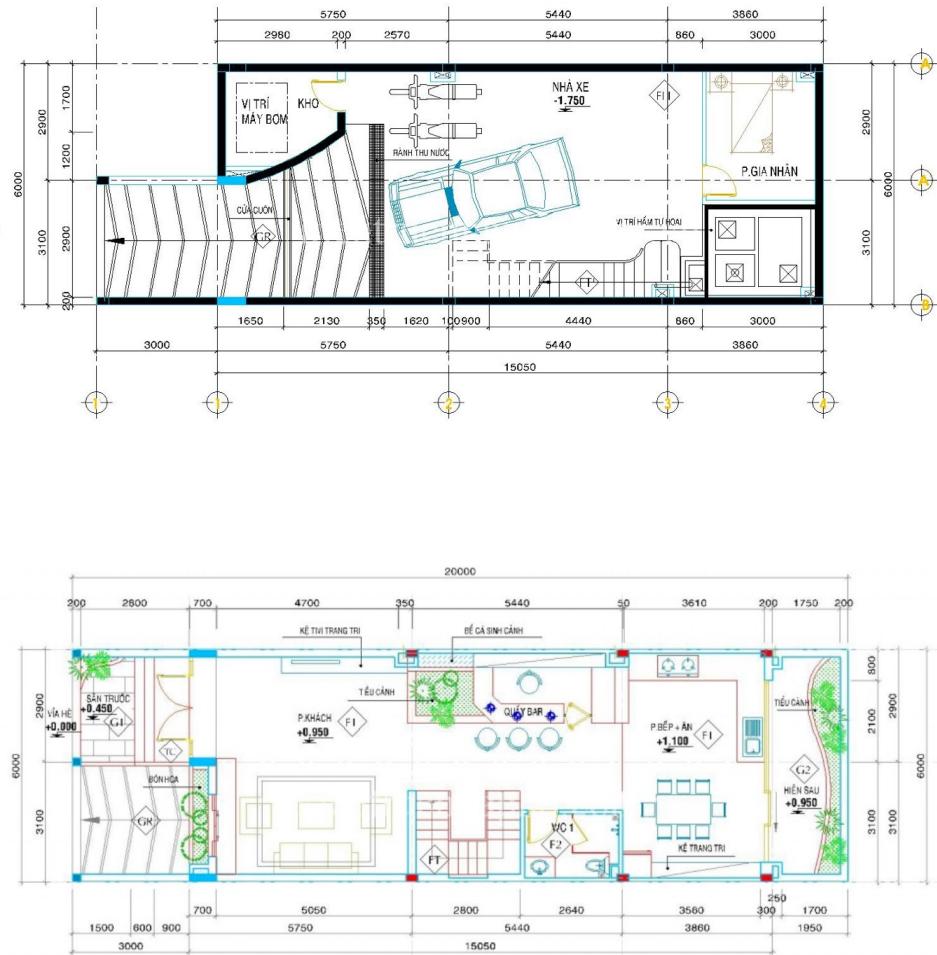
- Modelling
- Object-oriented modelling techniques
- History of UML
- Brief introduction to UML
 - Notions
 - Diagrams
 - Views

Model and Modelling

- A **model** is a simplification of reality.
We build models so we can better understand the system we are developing.
- **Modelling** is the process of building models to represent a system
- Modelling
 - helps us to visualise a system as it is or as we want it to be
 - allows us to specify the structure or behaviour of a system
 - gives us a template that guides us in constructing a system
 - documents the decision we have made

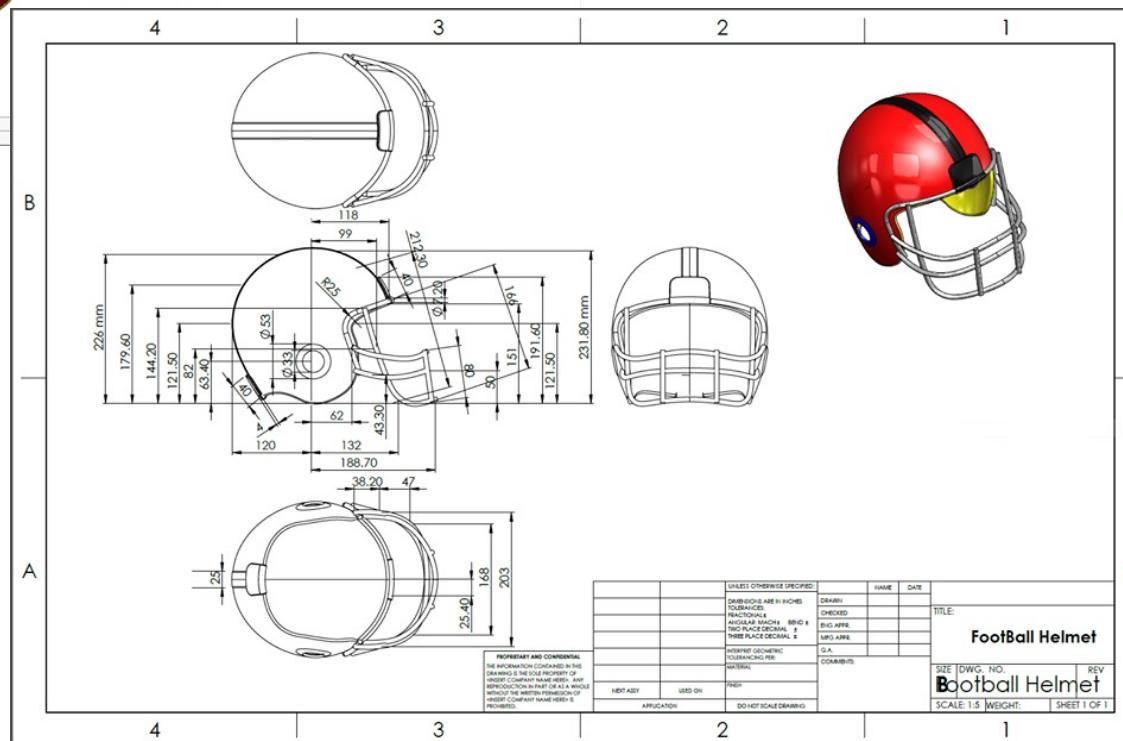
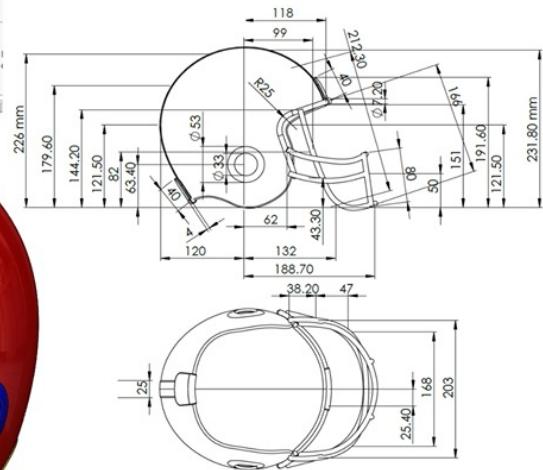
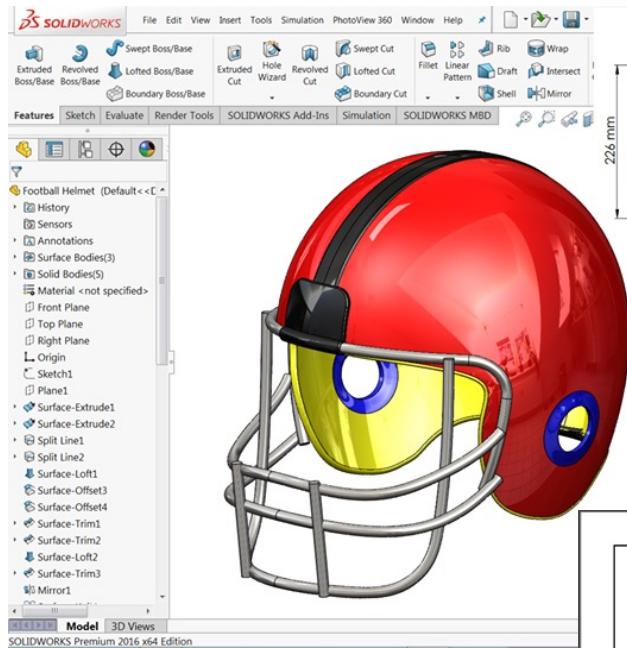


Model and Modelling: Example



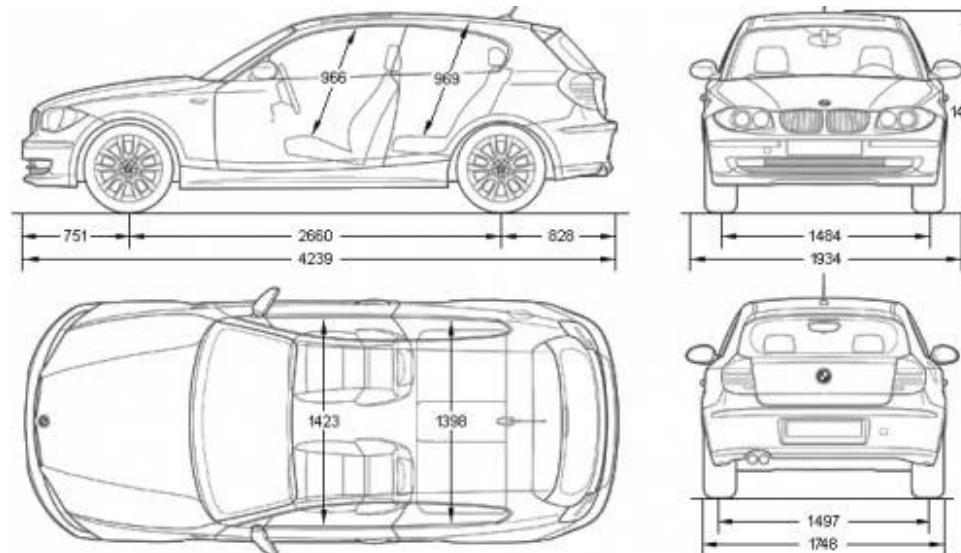
OOAD

Model and Modelling: Example



OOAD

Model and Modelling: Example



Modelling

- A model is a simple representation of a part of the real system with a specific purpose
 - The actual system is complex, then it is necessary to simplify
 - Master the complexity of the system
- A model represents the system
 - at certain abstraction level,
 - according to a viewpoint,
 - by means of description (e.g., text, image, ...)

Modelling: why?

- Better understand the system
 - Facilitate the master the problem
- Ease the communication
 - Supply means of communication between developers
- Better complete the system
 - Ease the recognition of consistency between models and the needs to improve and complete the system
- Specify the structure and the behaviour of the system
- Document the important decisions

Modelling

- Meta-model
 - is a representation of a model
 - can be used to
 - describe the syntax and the semantic of a model
 - manipulate models with tools
 - transform models
 - verify and maintain the coherence between models

Principles of modelling

- The choice of the appropriate model
 - Data view: entity-association model
 - Structural view: algorithm
 - Object-oriented view: classes and relations between them
- The models must represent the system at different levels of abstraction (according to the needs of the users)
- Models must be connected to the real world
 - Constructed models are close to real systems
 - Object approach > Procedural approach
- A system must be modelled by a set of models
 - A model is not sufficient
 - Describe different views of the systems: dynamic, static, installation, use, ...

Modelling

- A **good model** should
 - use a standardized notation
 - be understandable for customers and users
 - allow software developers to understand the system
 - provide an abstract view of the software
 - be visual

Benefits of modelling

- Ease the revision and the evolution of the system
- Reduce errors by allowing to detect errors early in the stages of development
- Reduce development cost
- Reduce time-to-market
- Reduce complexity by mechanism of abstraction

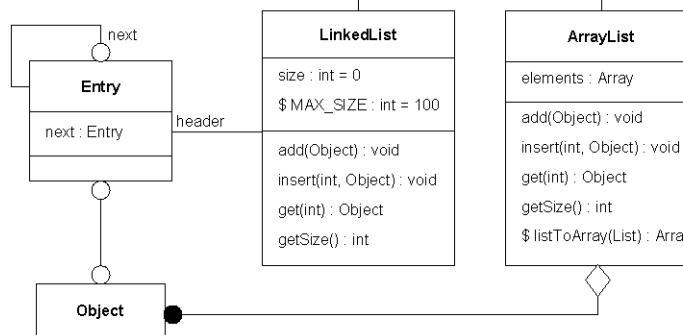
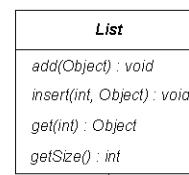
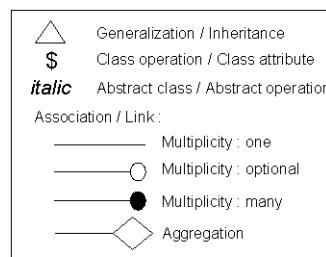
Object-oriented modelling techniques

- **Object-oriented modelling techniques** are processes/methodologies/approaches for software modelling and designing
 - 1975 - 1990: several object-oriented techniques are developed
 - 1990 - 1994: there are more than 50 object-oriented modelling techniques

- Best-known techniques
 - OOD (Object-Oriented Design)
 - OOSE (Object-Oriented Software Engineering)
 - OMT (Object Modelling Technique)

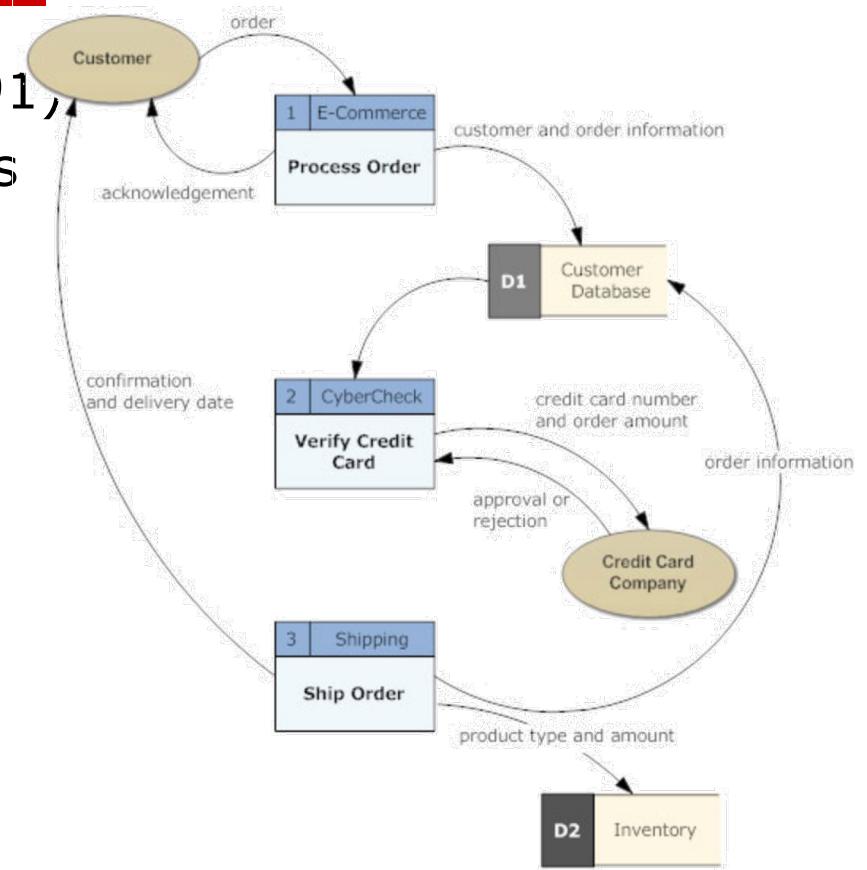
OMT technique

- Developed by Jim Rumbaugh (1991)
- Consists of 3 main types of models
 - Object model: Object diagram
 - Dynamic model: State diagram
 - Functional model: Data flow diagram

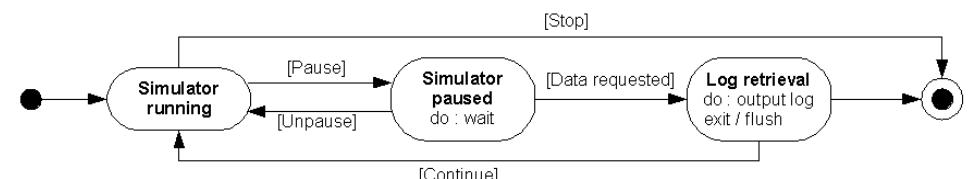


OMT Object Diagram

OOAD



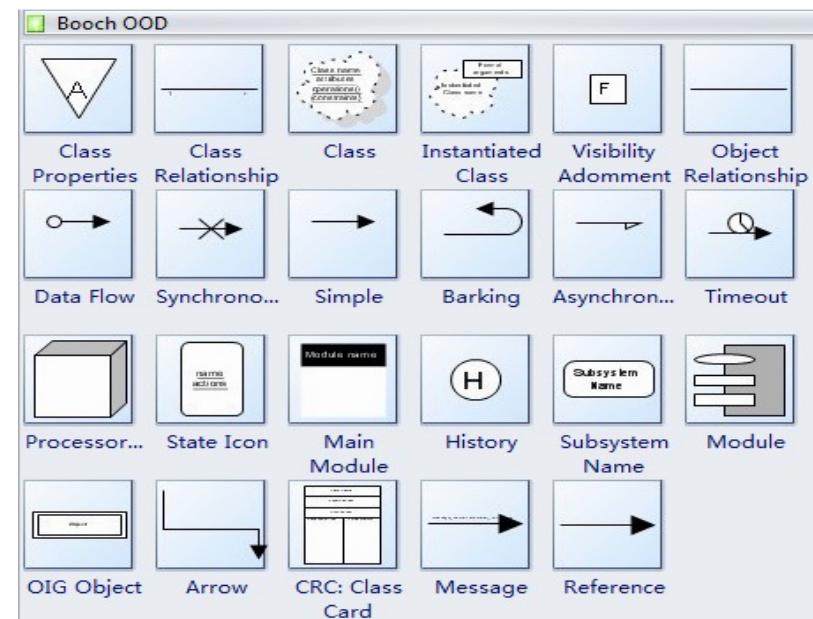
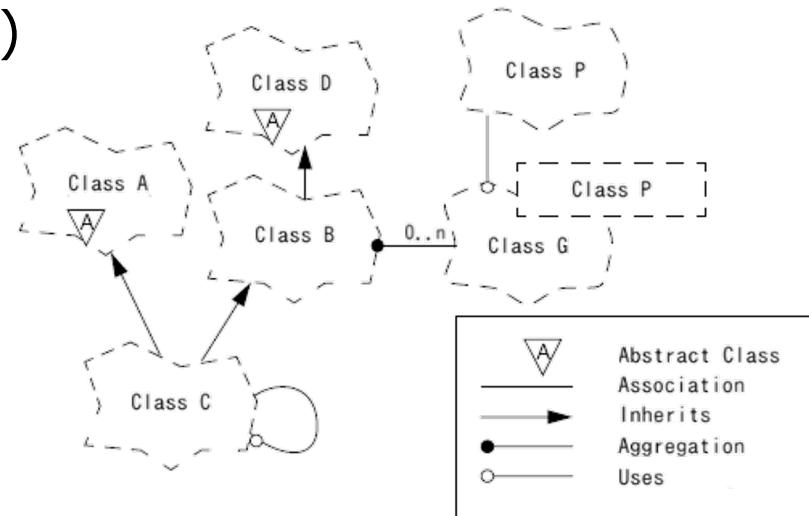
OMT Data flow Diagram



OMT State Diagram

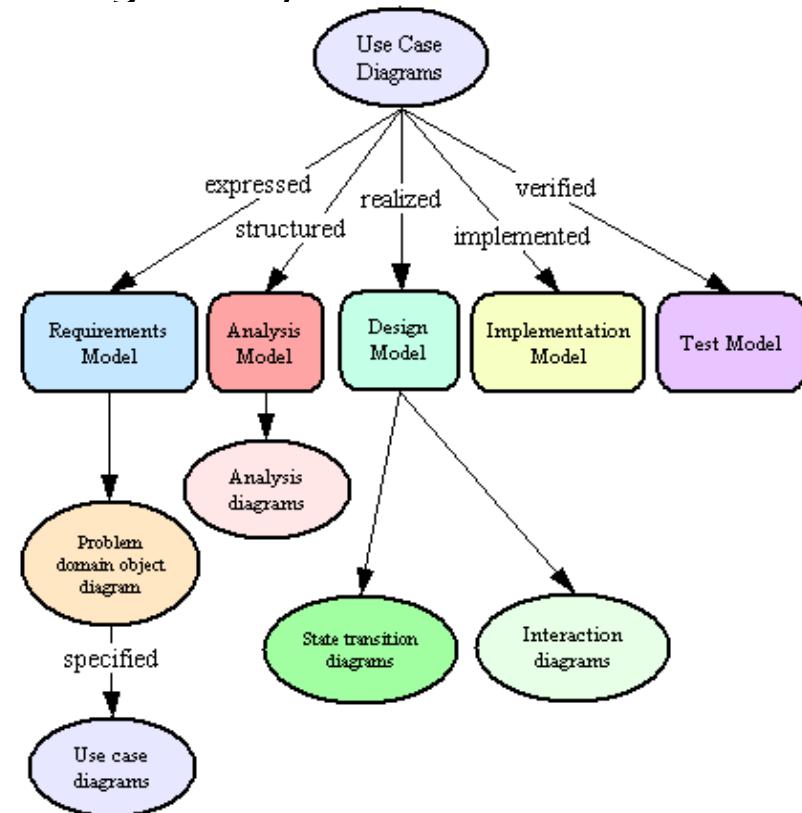
OOD technique

- Developed by Grady Booch (1991)
- Consists of
 - Static view
 - Class diagram
 - Object diagram
 - Module diagram
 - Dynamic view
 - State transition diagram
 - Process diagram
 - Interaction diagram



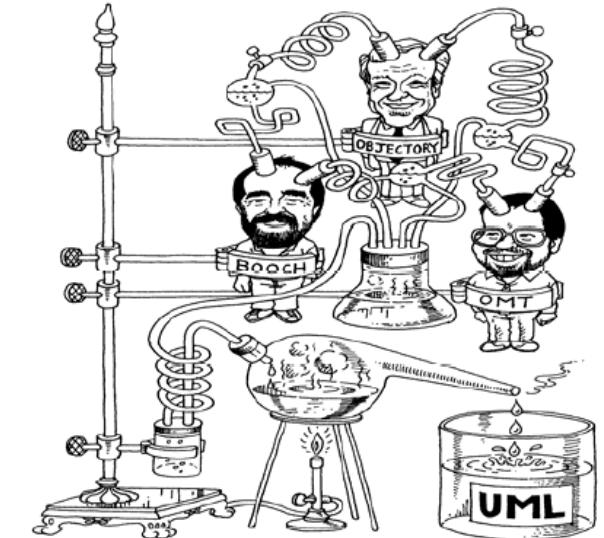
OOSE technique

- Developed by Ivar Jacobson (1992)
- Consists of 5 models
 - Requirements model: Problem domain diagram, Use-case diagram
 - Analysis model: Analysis diagram
 - Design model: State transition diagrams, Interaction diagrams
 - Implementation model
 - Test model

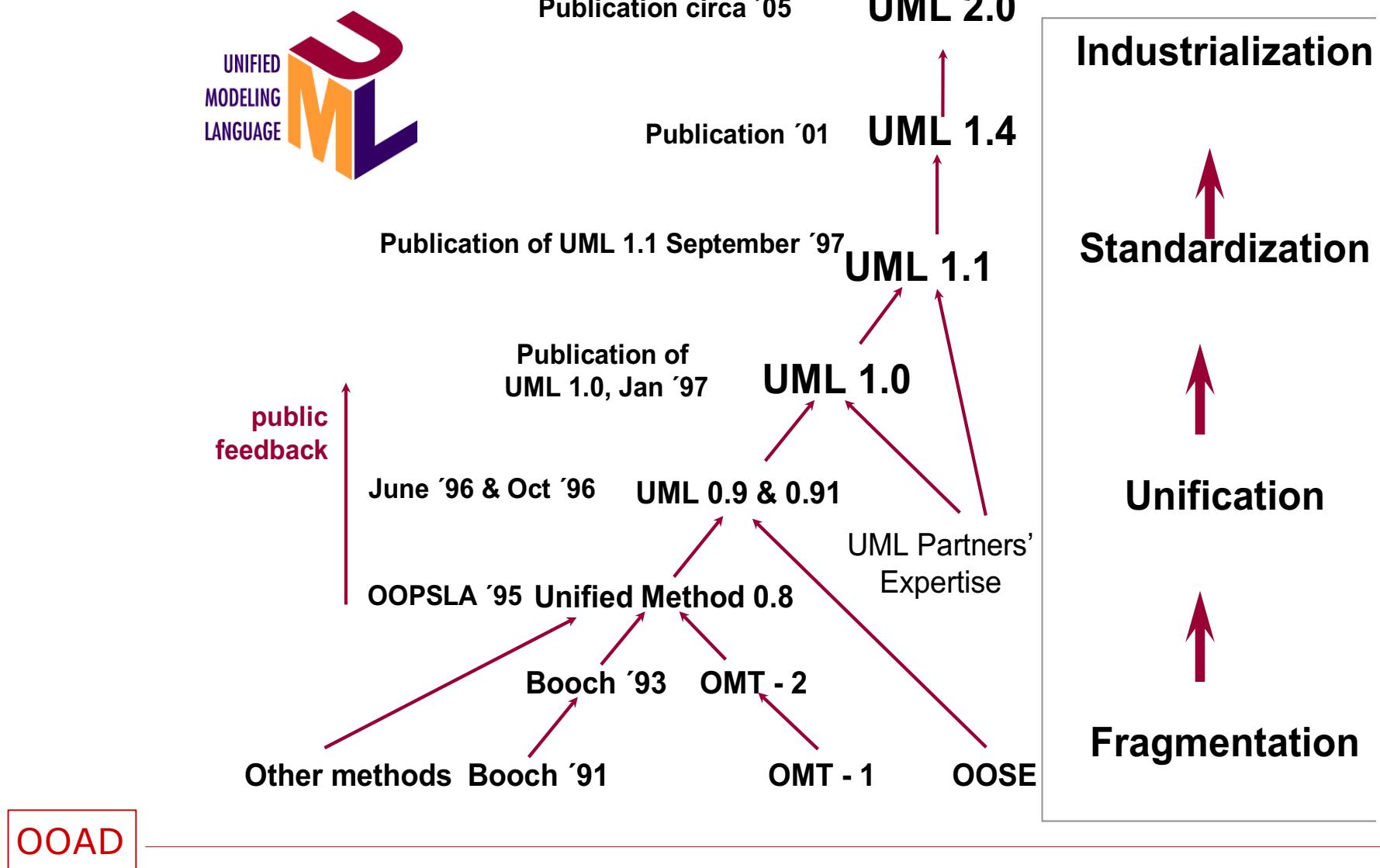


History of UML

- Too many object-oriented modelling techniques
 - Need for standardisation
 - Unification of modelling techniques
- In 1994
 - Rumbaugh (OMT) and Booch (OOD) unified their approaches for the UML project at Rational Software
- In 1995
 - The first version was released under the name "Unified Method" v0.8
- In 1996
 - Jacobson (OOSE) joined the team
- In 1997
 - The birth of UML v0.9 integrating OOSE
 - The first conference of the UML is organized
- In 2005, UML 2.0 is released
 - New diagrams, enhancement of existing diagrams
- In September, 2013, UML v.2.5 RTF - Beta 2

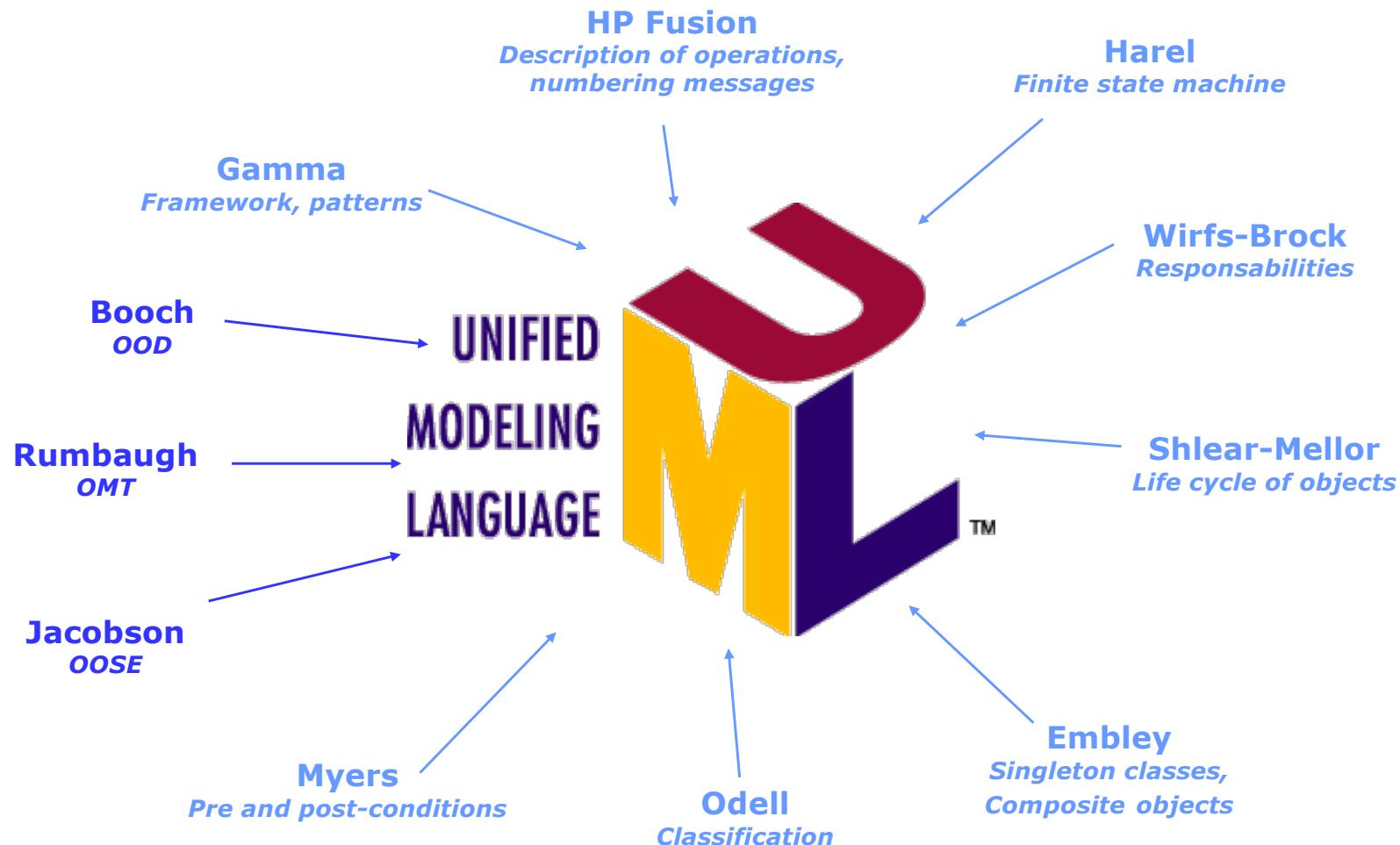


History of UML



History of UML

□ Contributions to UML



Introduction to UML



- **UML** (Unified Modelling Language) is a **modelling language**
 - consisting of the vocabulary, syntax and semantics
 - allowing to represent a system at different levels:
conceptual, physical
 - consisting of vocabulary and rules to describe different
models representing a system

- **UML**
 - is neither a methodology nor a process
 - allows freedom of design
 - can be combined with several development processes

Introduction to UML



- UML is a **language of visualisation**
 - using graphical representations
 - providing a better view of the system (thanks to graphical representations)
- UML is a **language of specification**
 - allowing to specify a system without ambiguity
 - allowing to specify a system at different stages: analysis, design, deployment
- UML is a **language of construction**
 - allowing to simulate the system
 - UML models are easily transformed into source code
- UML is a **language of documentation**
 - allowing to describe all the development stages of the system
 - Built models are complete documents of the system

UML Tools

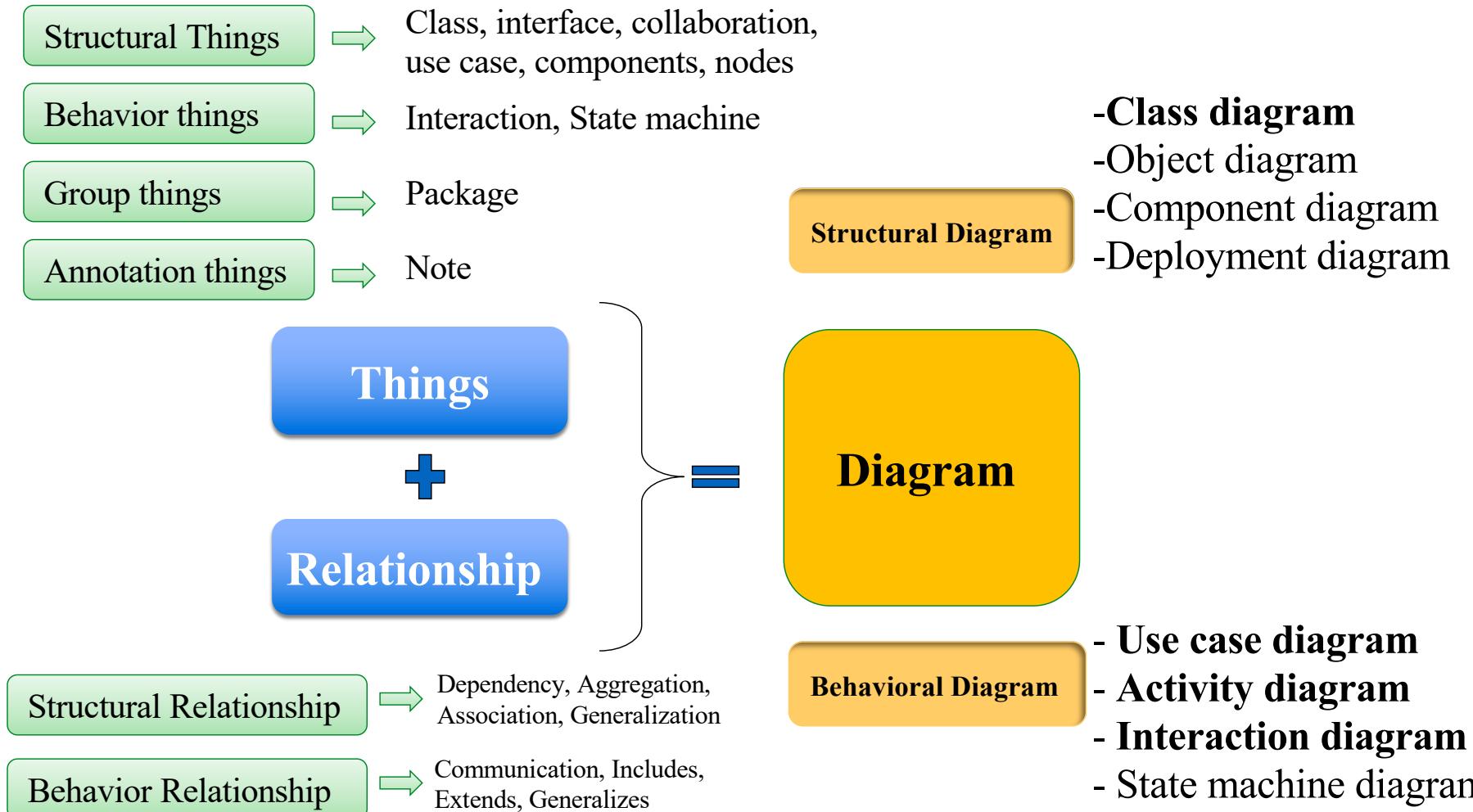
- Tools list
 - http://en.wikipedia.org/wiki/List_of_UML_tools
- Open source tools:
 - EclipseUML
 - UmlDesigner
 - ArgoUML...
- Commercial tools:
 - **Enterprise Architect**
 - IBM Rational Software Architect
 - Microsoft Visio
 - Visual Paradigm for UML
 - SmartDraw...

Introduction to UML: the diagrams

- Consisting of 10 main diagrams
 - **Requirements** modelling
 - Use-case diagrams
 - **Static structure** modelling
 - Class diagrams
 - Object diagrams
 - **Dynamic behaviour** modelling
 - Interaction diagrams
 - Sequence diagrams
 - Collaboration diagrams
 - Activity diagrams
 - State diagrams
 - **Architectural** modelling
 - Package diagrams
 - Component diagrams
 - Deployment diagrams

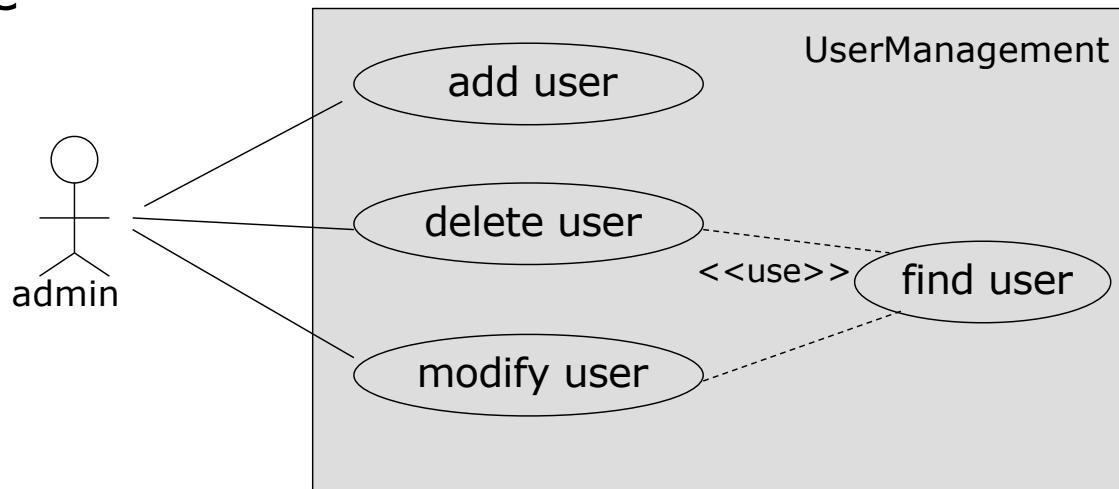


UML Overview



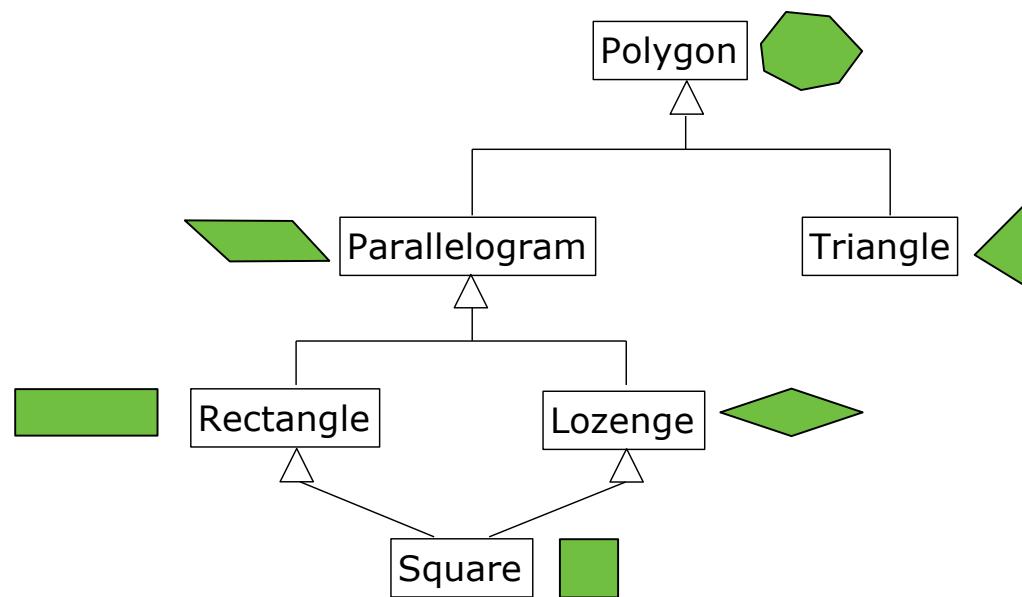
Introduction to UML: Use-case diagram

- Showing the possible uses of a system
 - Describing **the static view** of the system according to users perspective
 - Being very important to understand the functions of the system
-
- Example



Introduction to UML: class diagram

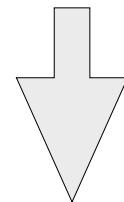
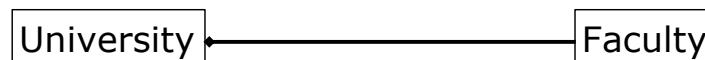
- Describing the classes and their relationship
- Describing **the static view** of the system
- Example



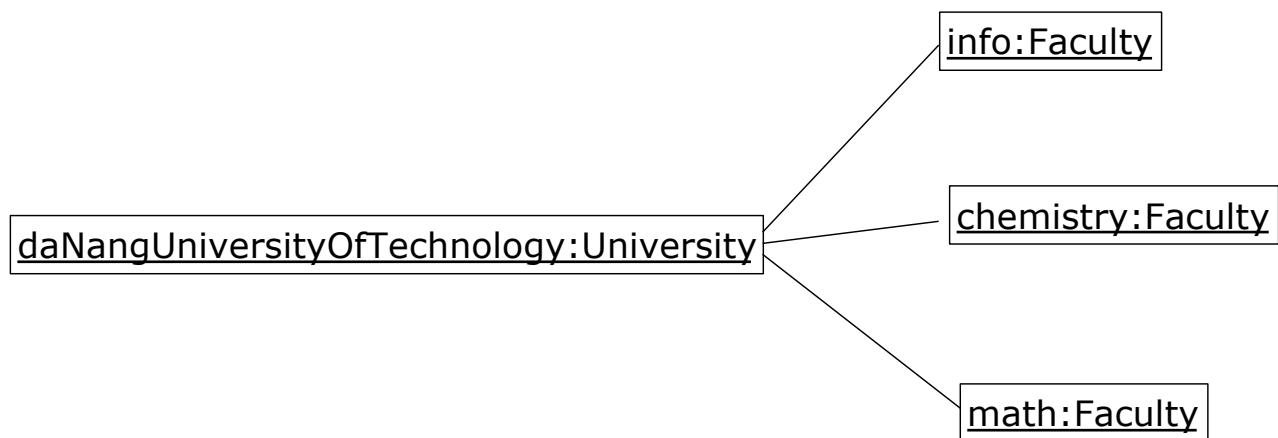
Introduction to UML: object diagram

- Describing a set of objects and their relationship
- An object diagram represents the same information that a class diagram but at the instance level of classes
- Describing **the static view** of the system
- Example

Class diagram



Object diagram

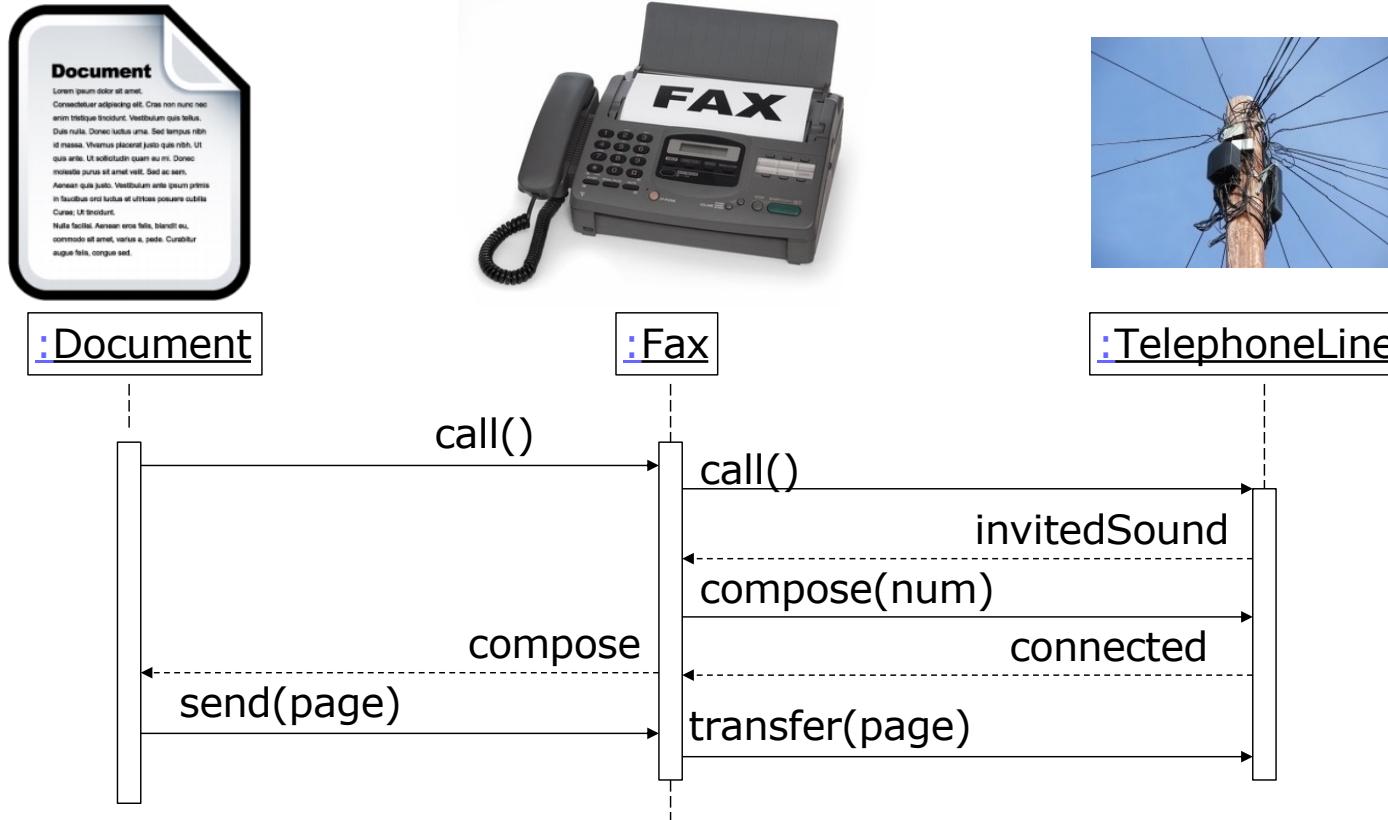


Introduction to UML: interaction diagram

- Describing the behaviours of the system by the interactions between the composing objects
 - Modelling **the dynamic view** of the system
 - The interaction diagram is an extension of the object diagram by describing the interactions between objects
-
- Consisting of two types of diagrams
 - **Sequence Diagram** describes the interactions between objects with the emphasis on sequencing of messages
 - **Collaboration Diagram** describes the interactions between objects with the emphasis on the structure of objects

Introduction to UML: interaction diagram

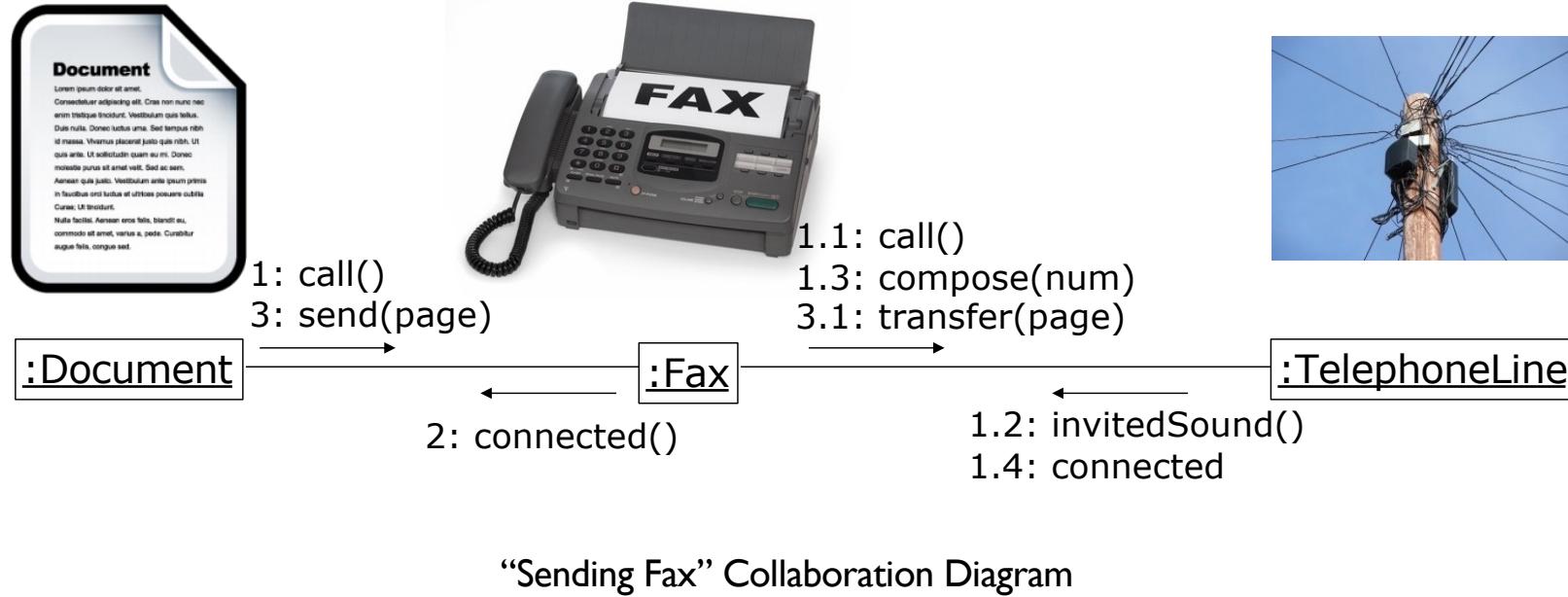
- Sequence Diagram example



“Sending Fax” Sequence Diagram

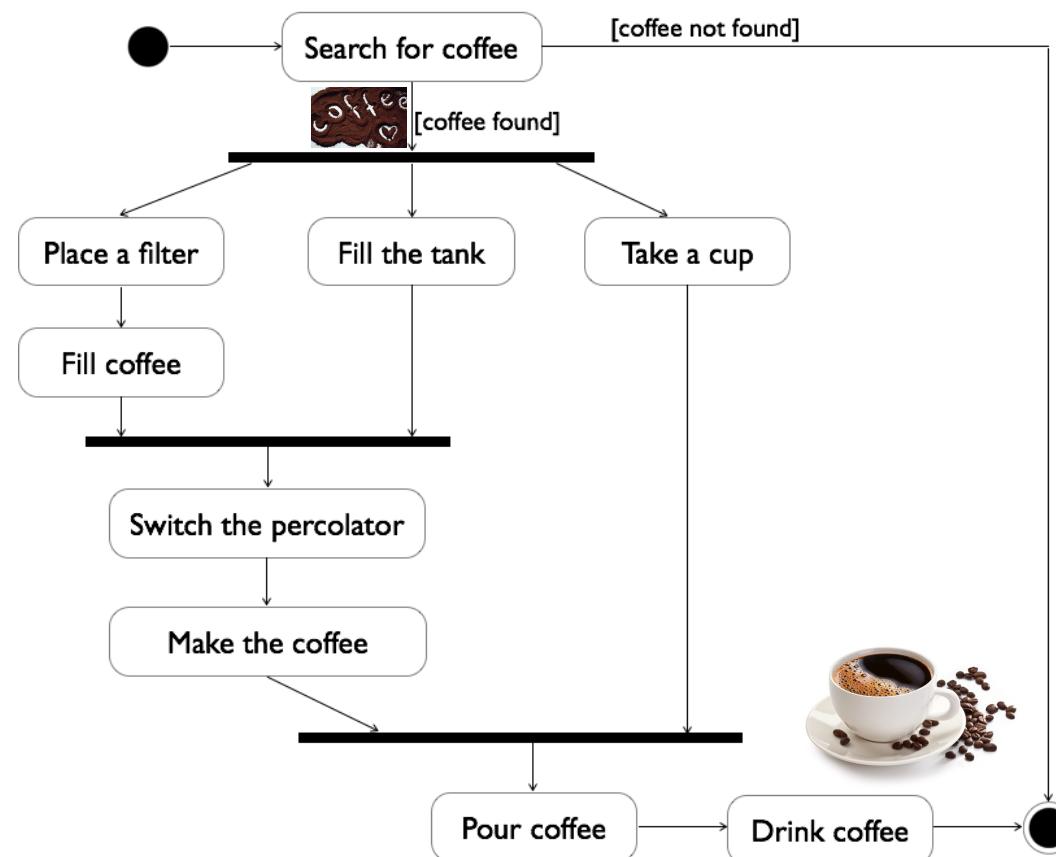
Introduction to UML: interaction diagram

□ Collaboration diagram example



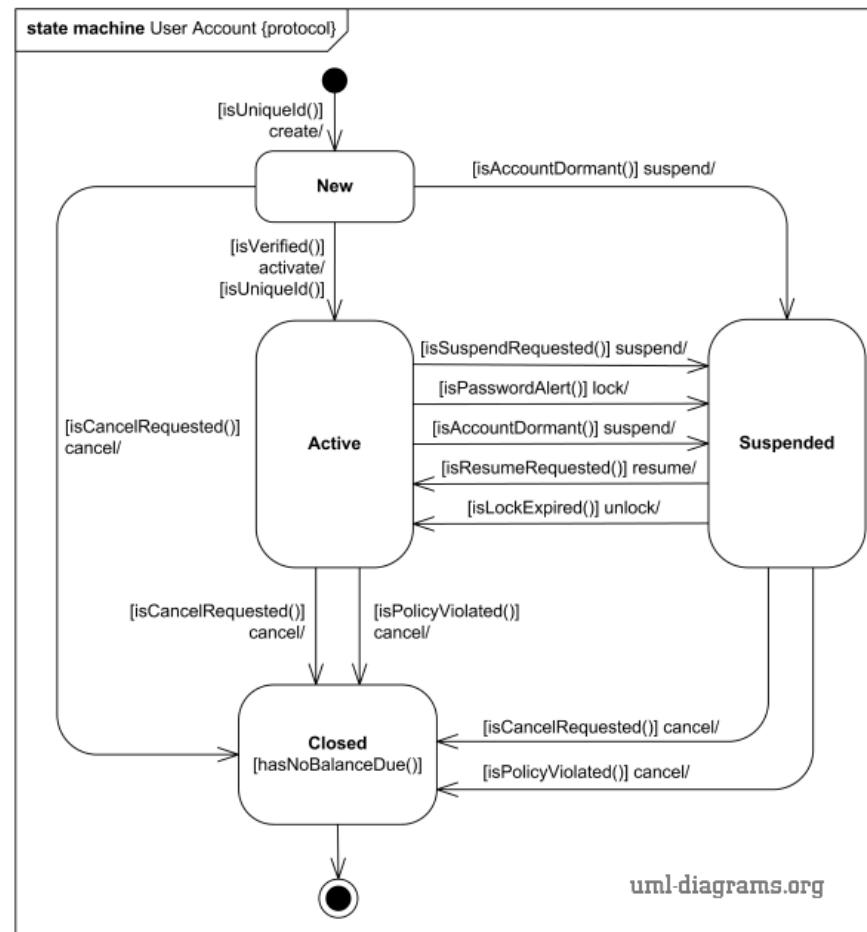
Introduction to UML: activity diagram

- Describing the information flows in the system
- Modelling **the dynamic view** of the system
- Example: Making coffee



Introduction to UML: state diagram

- Describing the internal behaviour of the system
- Modelling the **dynamic view** of the system
- Example



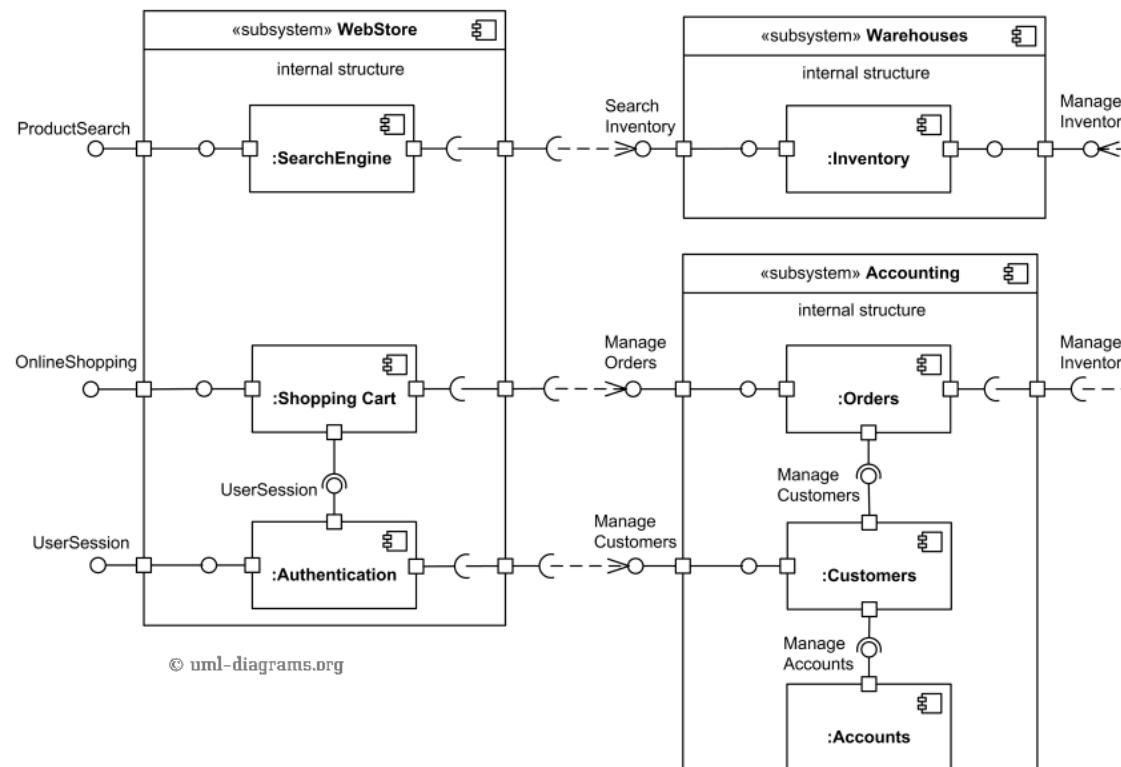
“Online Shopping Account” State Diagram



ebay™
amazon
Alibaba.com™

Introduction to UML: component diagrams

- Describe the organisation of different components of the system
- The **static view** of the organisation of the system
- Example

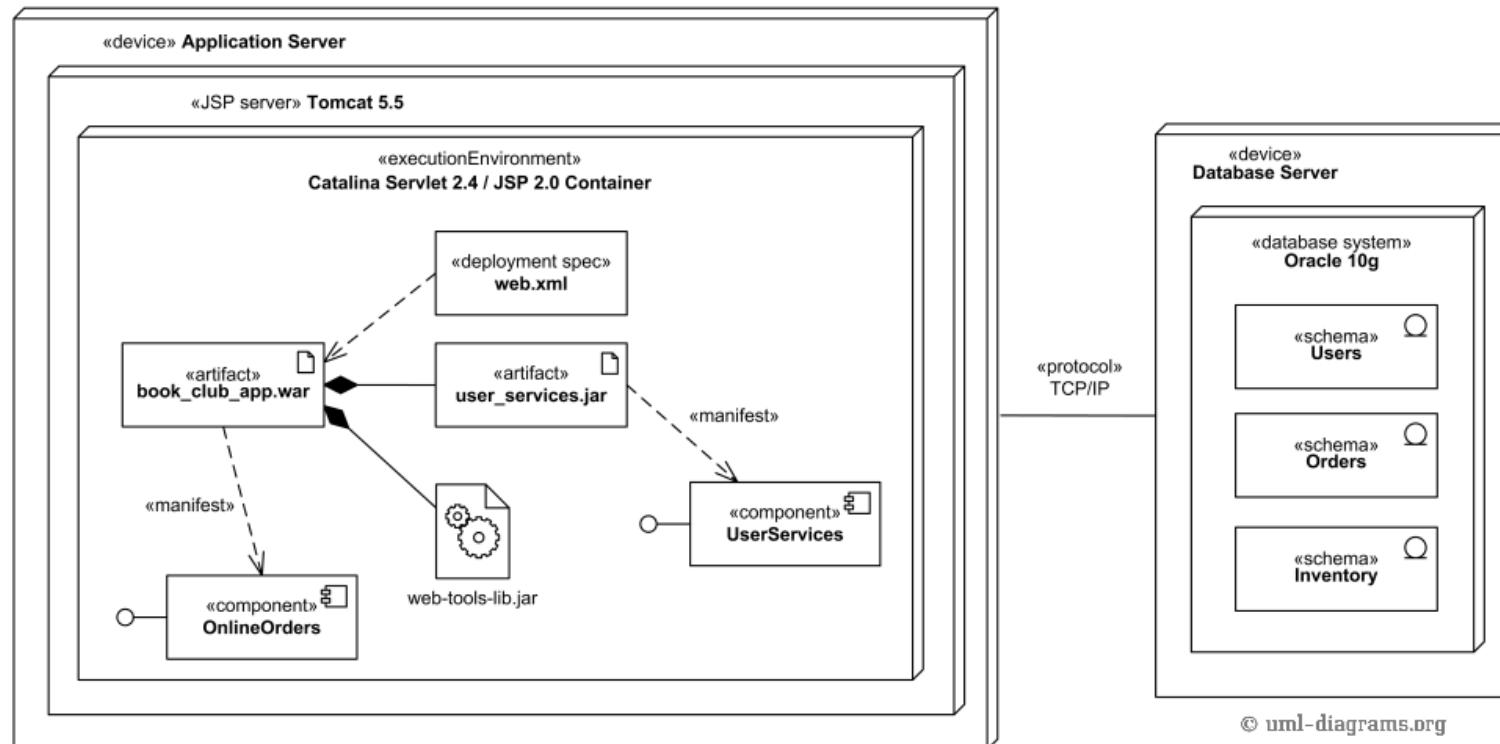


“Online Shopping Website” Component Diagram



Introduction to UML: deployment diagrams

- Describing the physical organisation of different components (machines) of the system (material)



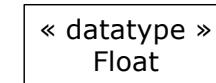
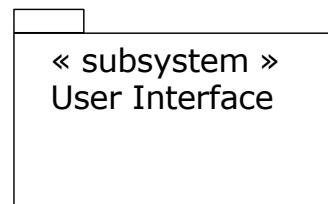
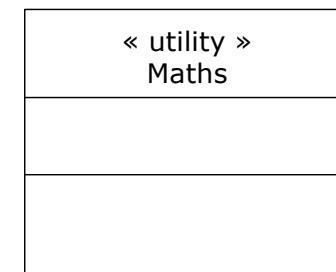
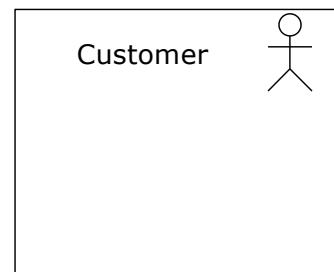
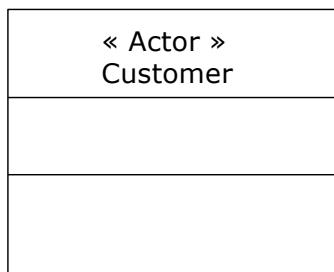
An example of deployment diagram of JEE web application

Introduction to UML: extension mechanism

- Built-in extension mechanism
 - Stereotypes
 - Tagged values
- Notes
- Constraints
 - OCL textual language

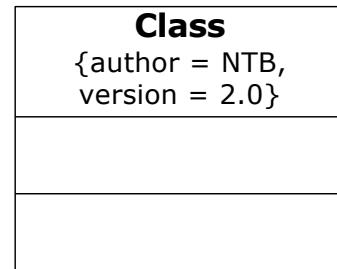
Introduction to UML: general mechanisms

- Stereotype
 - is a built-in extension mechanism
 - expands the vocabulary of UML
 - is used to create new types of UML elements that derive from the existing kinds but which are adapted to a given problem
 - there are predefined stereotypes in UML
 - Notation
 - “name of stereotype”
 - Possibility to introduce an icon



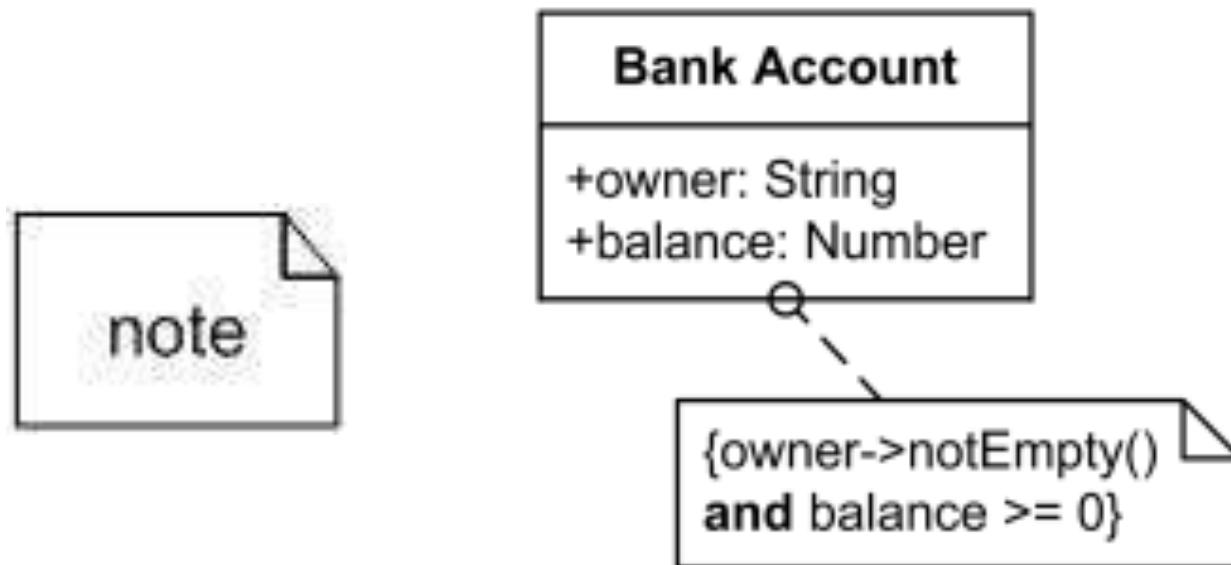
Introduction to UML: general mechanisms

- Tagged values
 - Another extension mechanism
 - Provide additional information on the elements of UML
 - Pairs of type {name = value}
 - Example



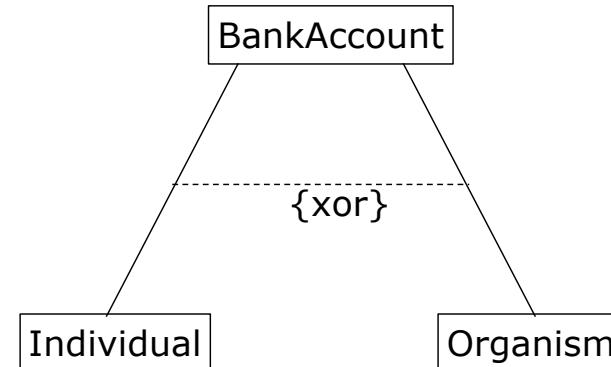
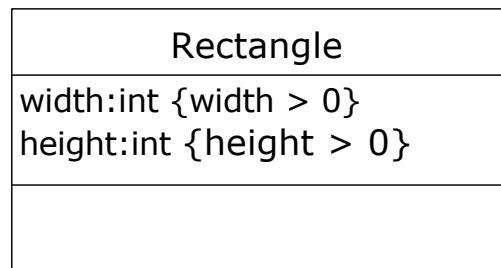
Introduction to UML: general mechanisms

- Notes
 - are comments attached to one or more modelling elements
 - provide additional information on modelling elements
 - belong to the view, not the models



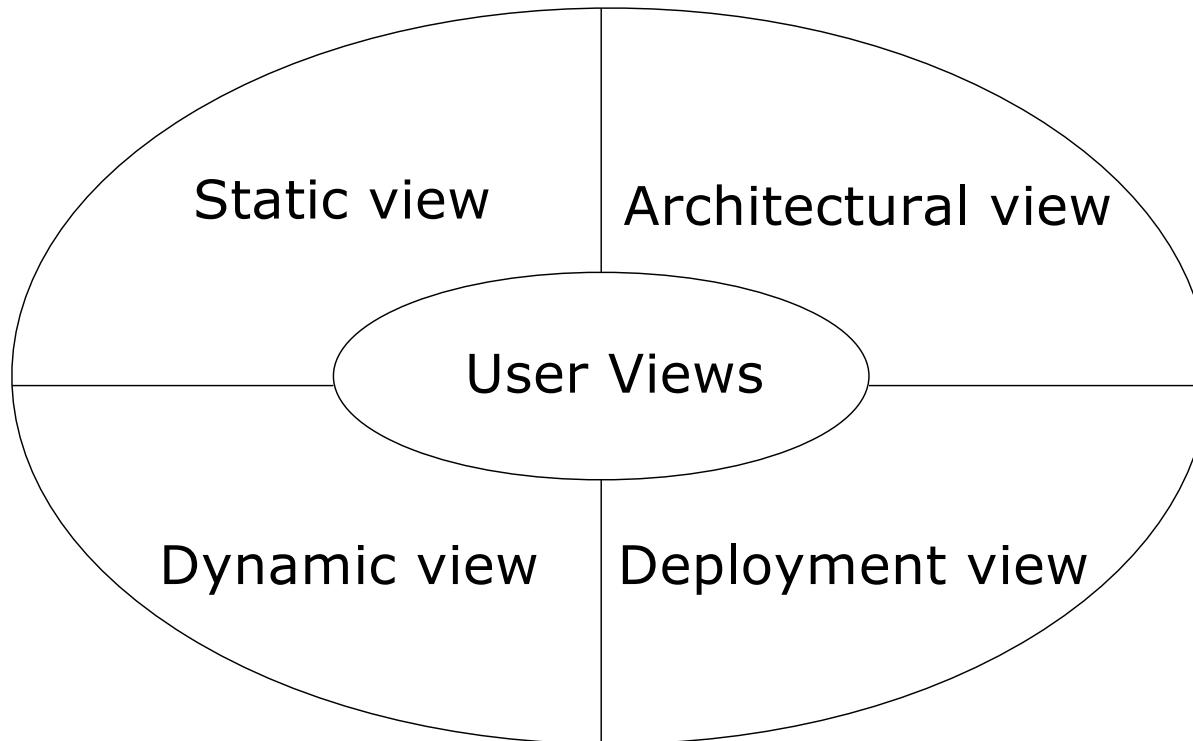
Introduction to UML: general mechanisms

- Constraints
 - are restrictions that limit the use of an element or the element semantic
 - are expressed in natural language
 - are expressed in OCL (Object Constraint Language)
 - Example



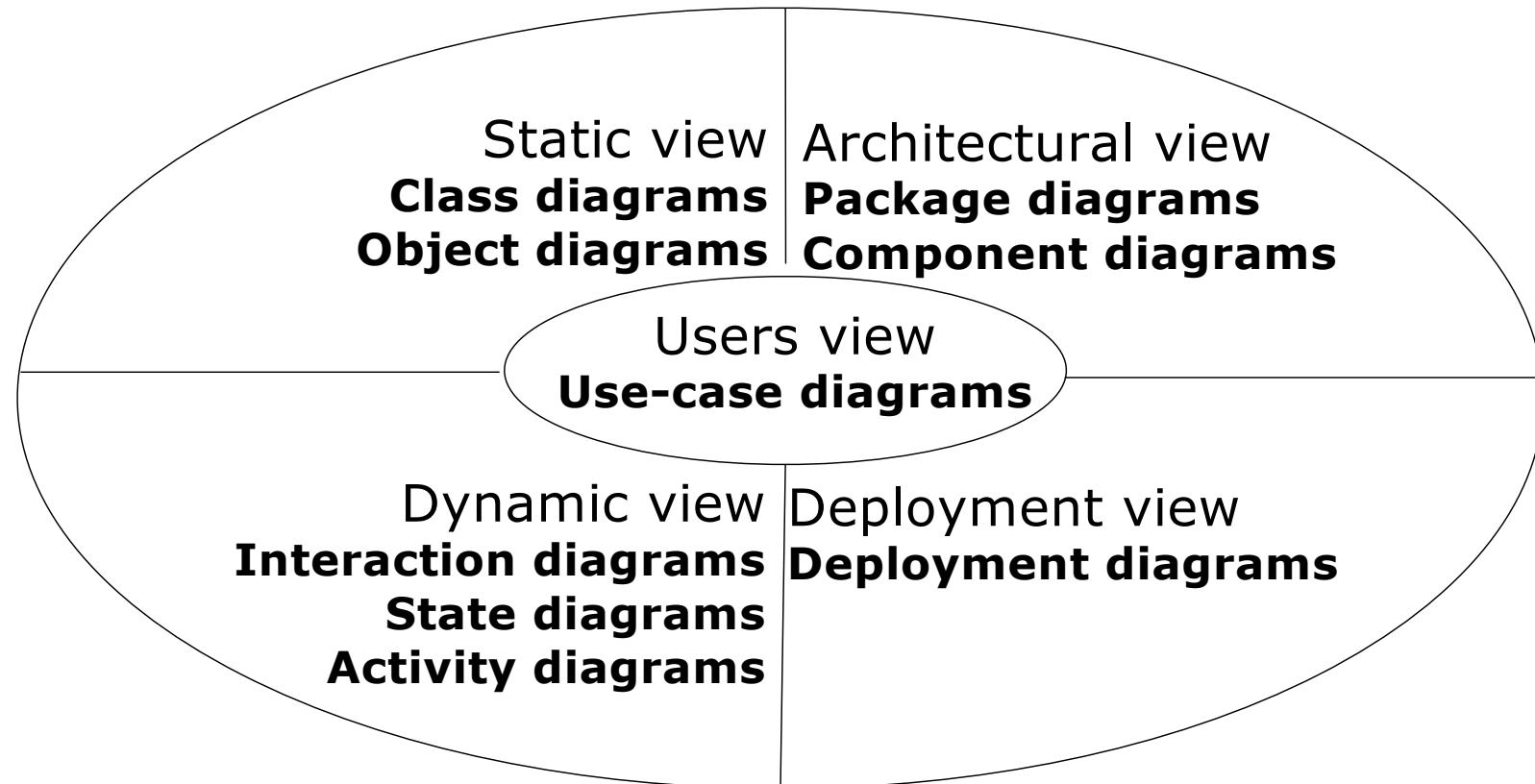
Introduction to UML: views

- A system is modelled by 5 different views in the UML



Introduction to UML: views

□ Diagrams and views



UML and Software Development Process

- Software Development Activities
- Object-Oriented Analysis and Design
- Software Development Processes
- UML and Software Development Processes

Main Software Development Activities

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

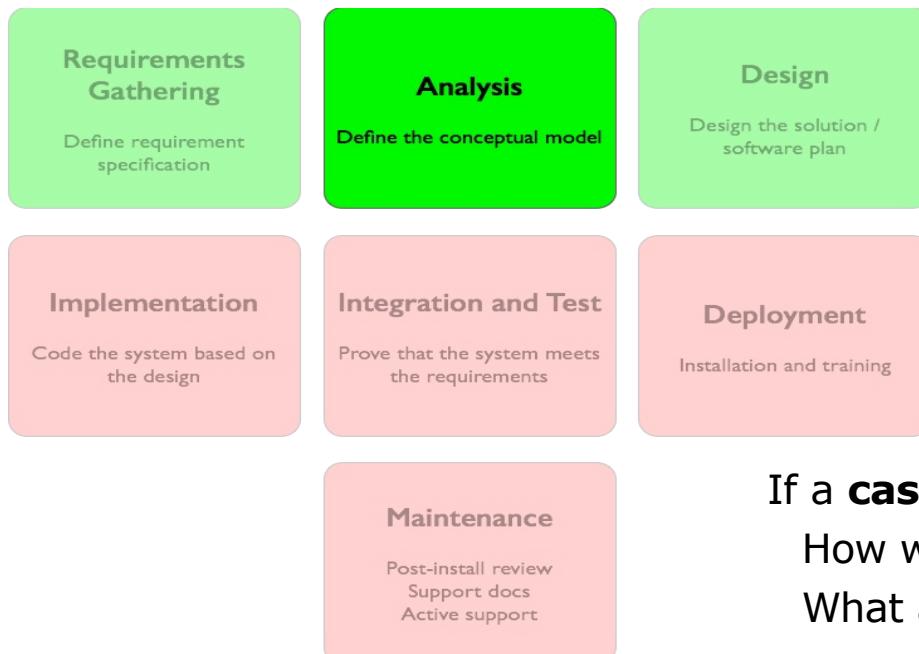
Deployment

Installation and training

Maintenance

Post-install review
Support docs
Active support

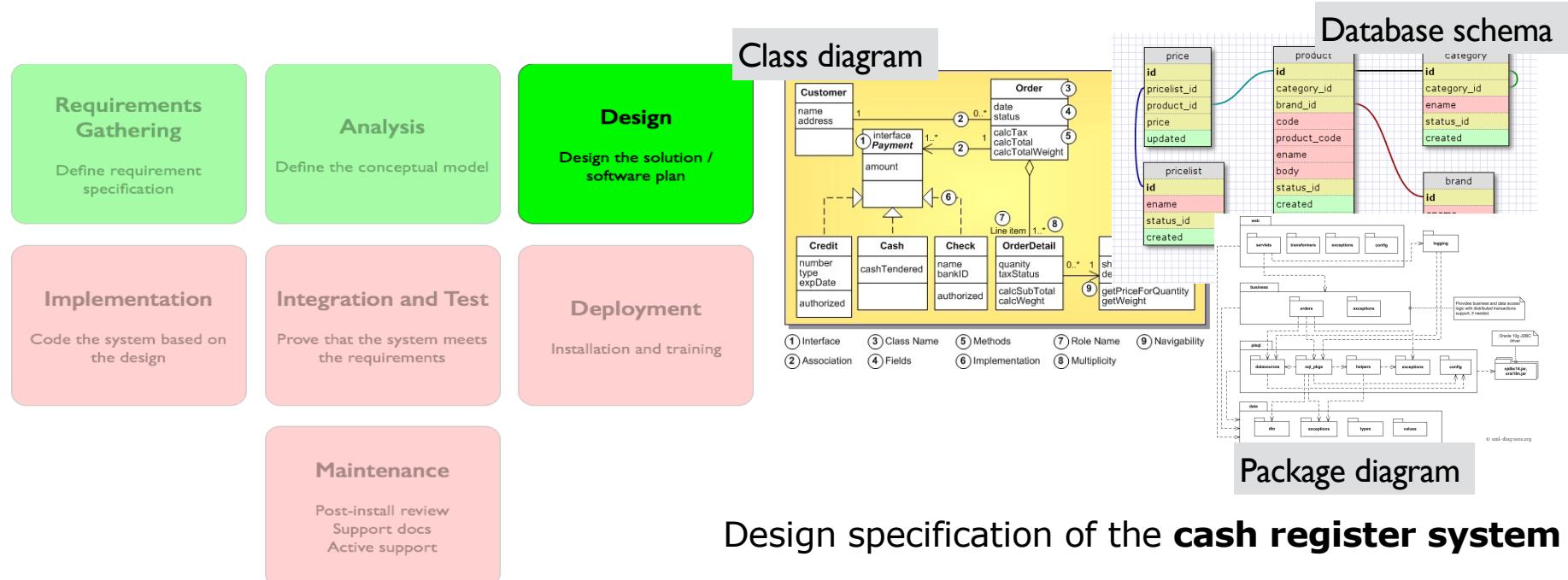
Analysis emphasizes an investigation of the problem and requirements, rather than a solution. During **object-oriented analysis**, there is an emphasis on finding and describing object or concepts in the problem domain.

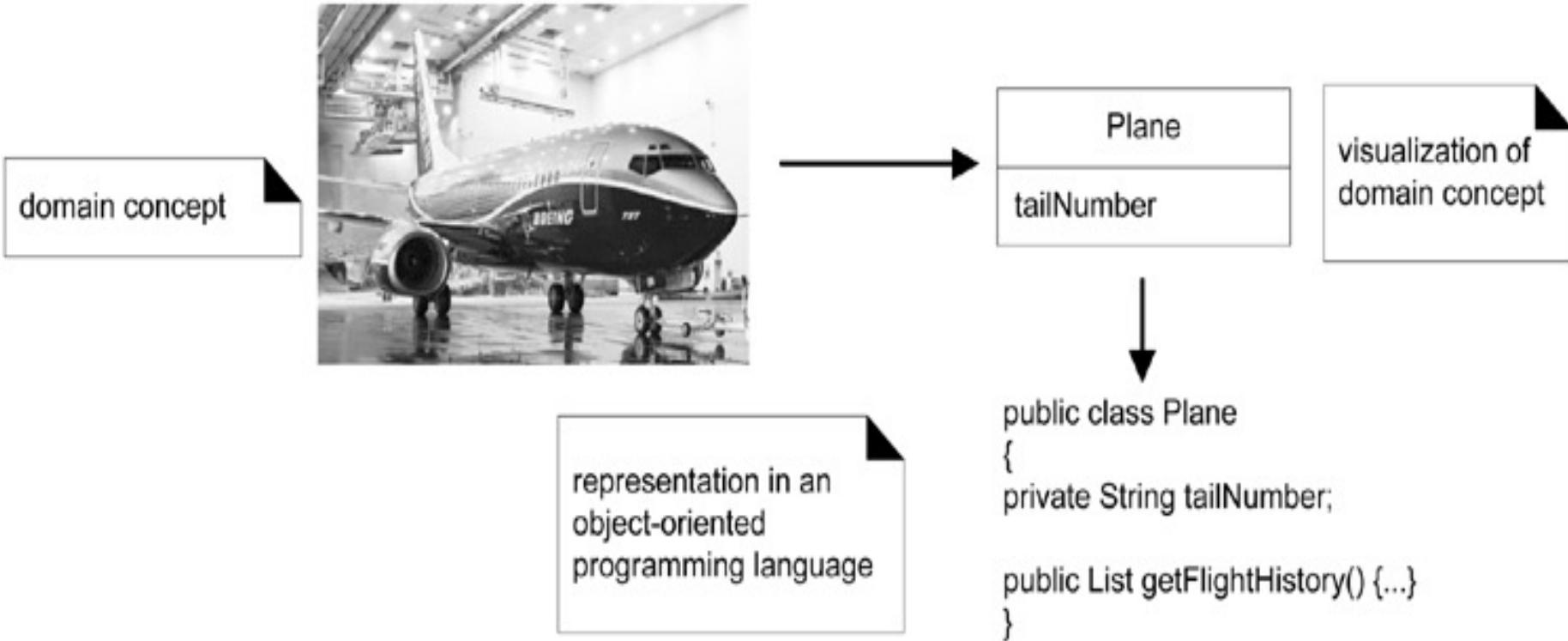


If a **cash register system** at the supermarket is desired
How will it be used?
What are its functions?

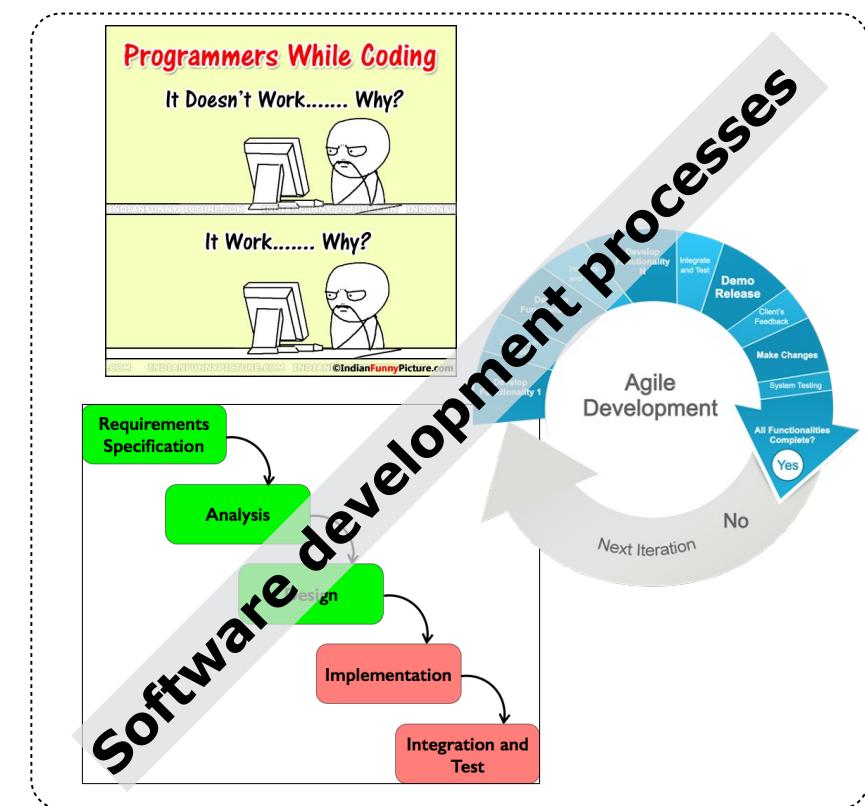
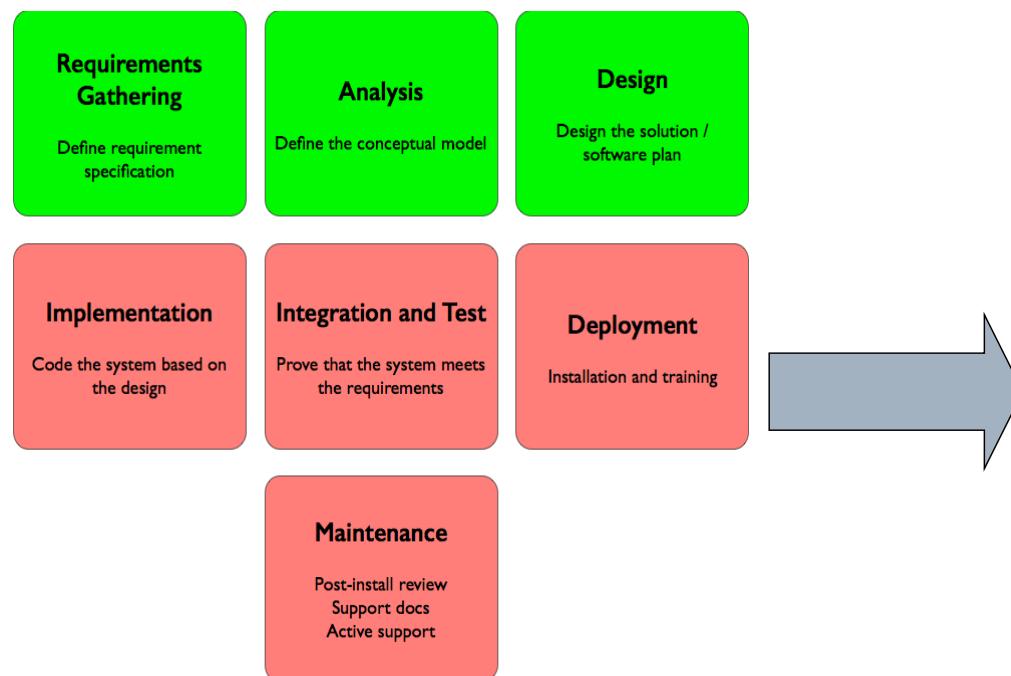
Design emphasizes a conceptual solution in software that fulfils the requirements and “guides” the implementation.

During **object-oriented design**, there is an emphasis on defining software objects and how they collaborate to fulfil the requirements.



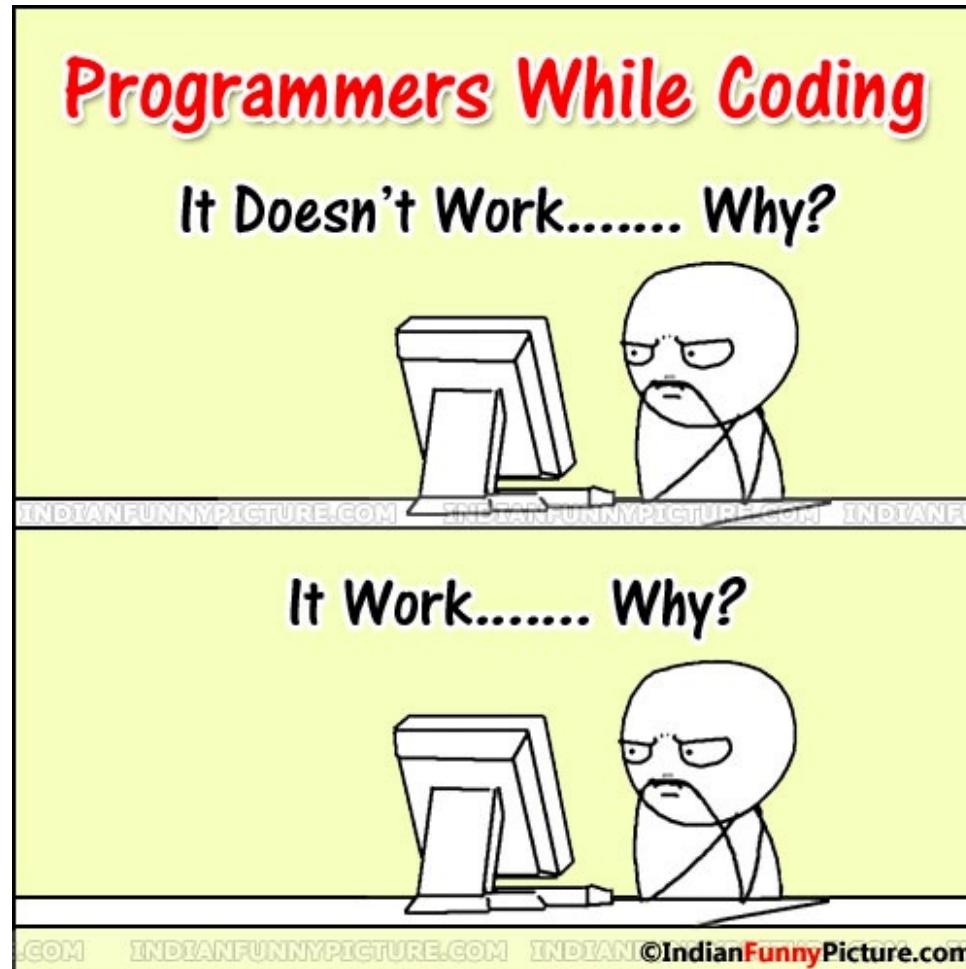


Software development process is a series of software development activities that a software program goes through when developed

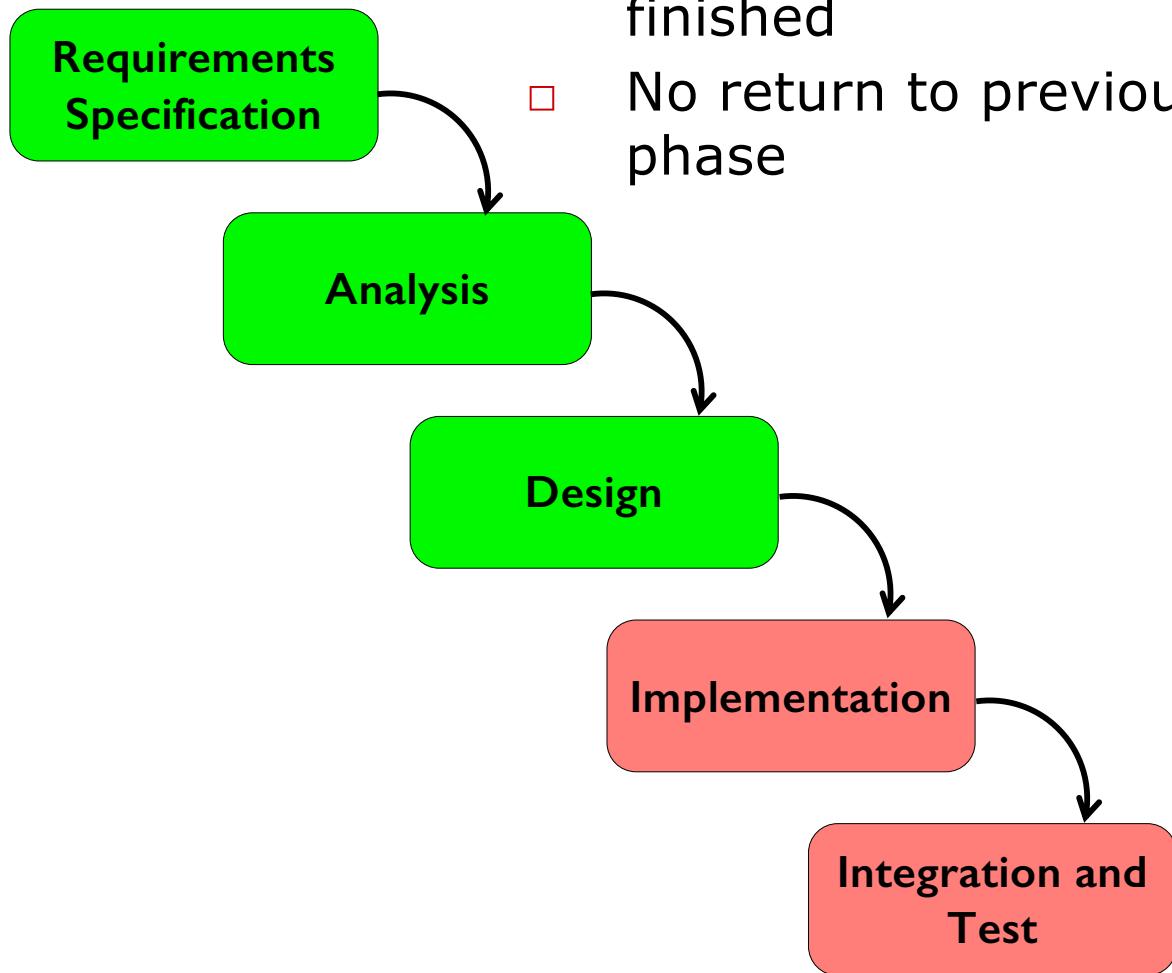


Ad-hoc Coding “process”

- Does not scale to large size project
- Does not scale to large development teams

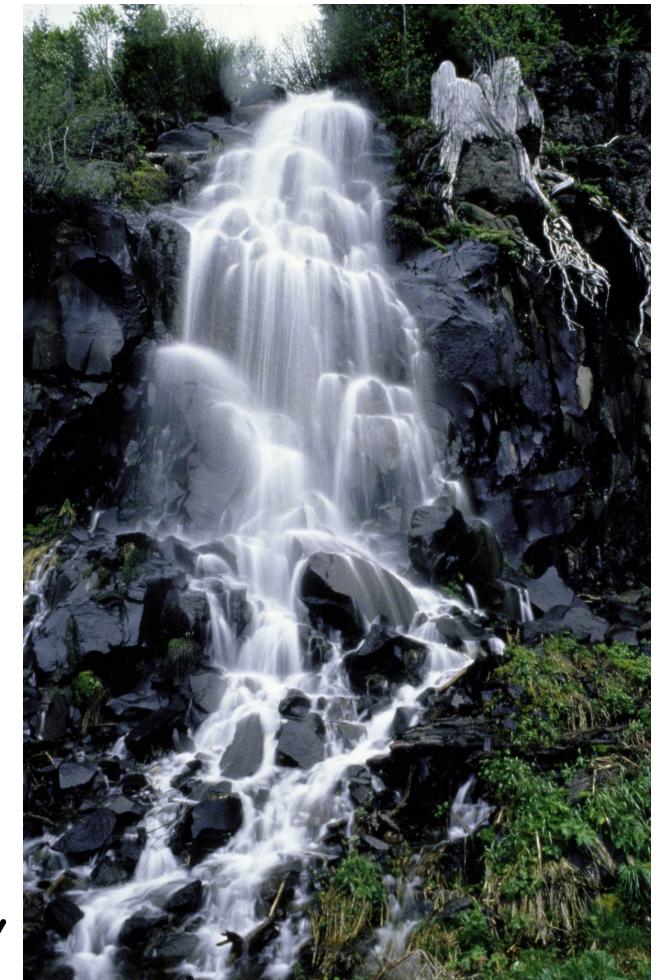


Waterfall process



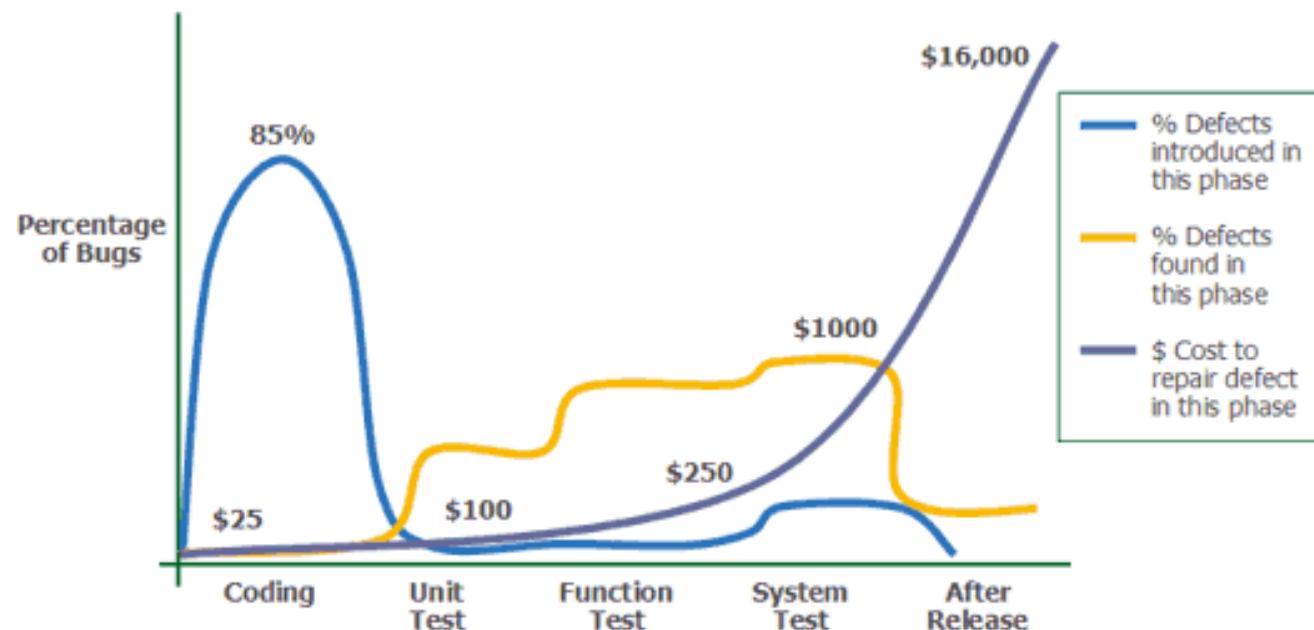
- ❑ An phase is begun only when the previous has finished
- ❑ No return to previous phase

Sequential



Critique of Waterfall process

- Responds poorly to changes and problems
- Substantial upfront document
- Assumes fixed specification - may not be what customer wants
- Fixes come very late - costlier to fix later time



Source: Applied Software Measurement, Capers Jones, 1996.

Iterative and Agile Development Processes

Facts of life

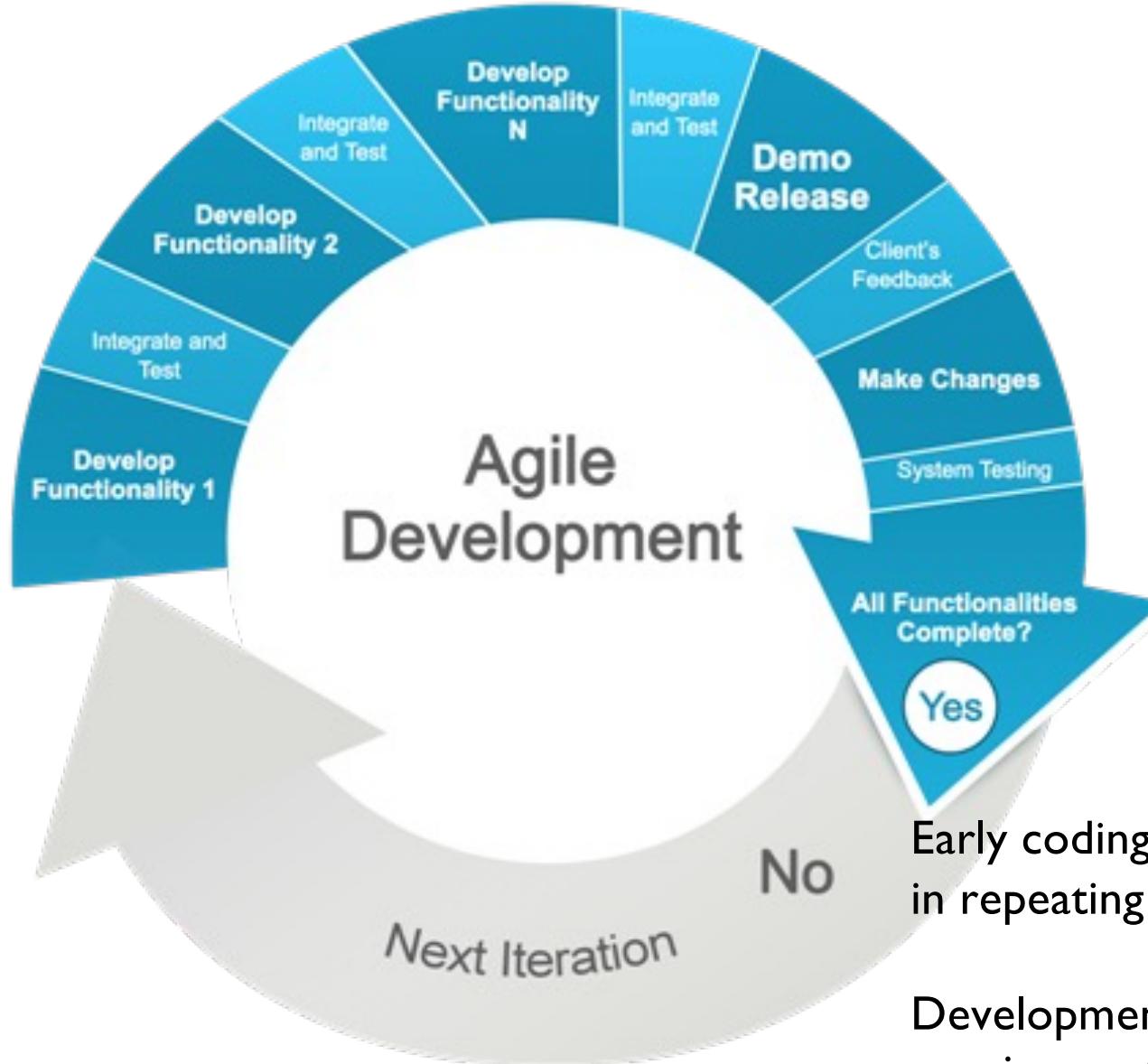
- Requirements change, changes break existing design.
- Coding up a design suggests flaws in design
- Testing identifies flaws in code - which could be design flaws
- Maintenance requires not only fixes but new features

Source: ?

Philosophy

- Embrace change
- Don't do too much, too soon
- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Source: ?



Source: ?

OOAD

Feedback is used to clarify evolving specification.

Early coding, early testing of partial system in repeating cycles.

Development begins before all requirements are defined in detail.

Benefits

- Early rather than late mitigation of high risks
- Early visible progress
- Managed complexity - the team is not overwhelmed by "analysis paralysis" or very long and complex steps

**Less project failure, better productivity,
and lower defect rates**

Early feedback, user engagement, and adaptation, leading to a redefined system that more closely meets the real needs of the stakeholders

- Feedback can also improve development process itself

Agile software development methods

- Adaptive software development (ASD)
- Agile modeling
- Agile Unified Process (AUP)
- Crystal Clear Methods
- Disciplined agile delivery
- Dynamic Systems development method (DSDM)
- Extreme programming
- Feature-driven development (FDD)
- Lean software development
- Kanban
- Scrum

Scrum

The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



Product Owner



The Team



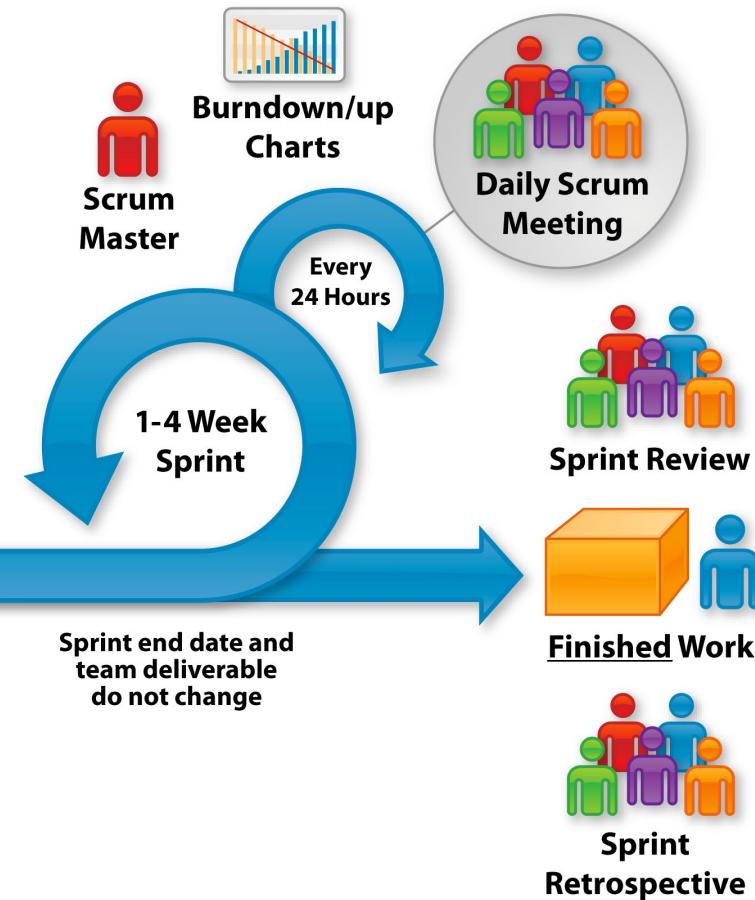
Product
Backlog

Ranked list of what is required: features, stories, ...
Team selects starting at top as much as it can commit to deliver by end of Sprint

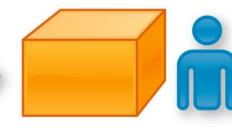
Sprint
Planning
Meeting



Sprint
Backlog



Sprint Review

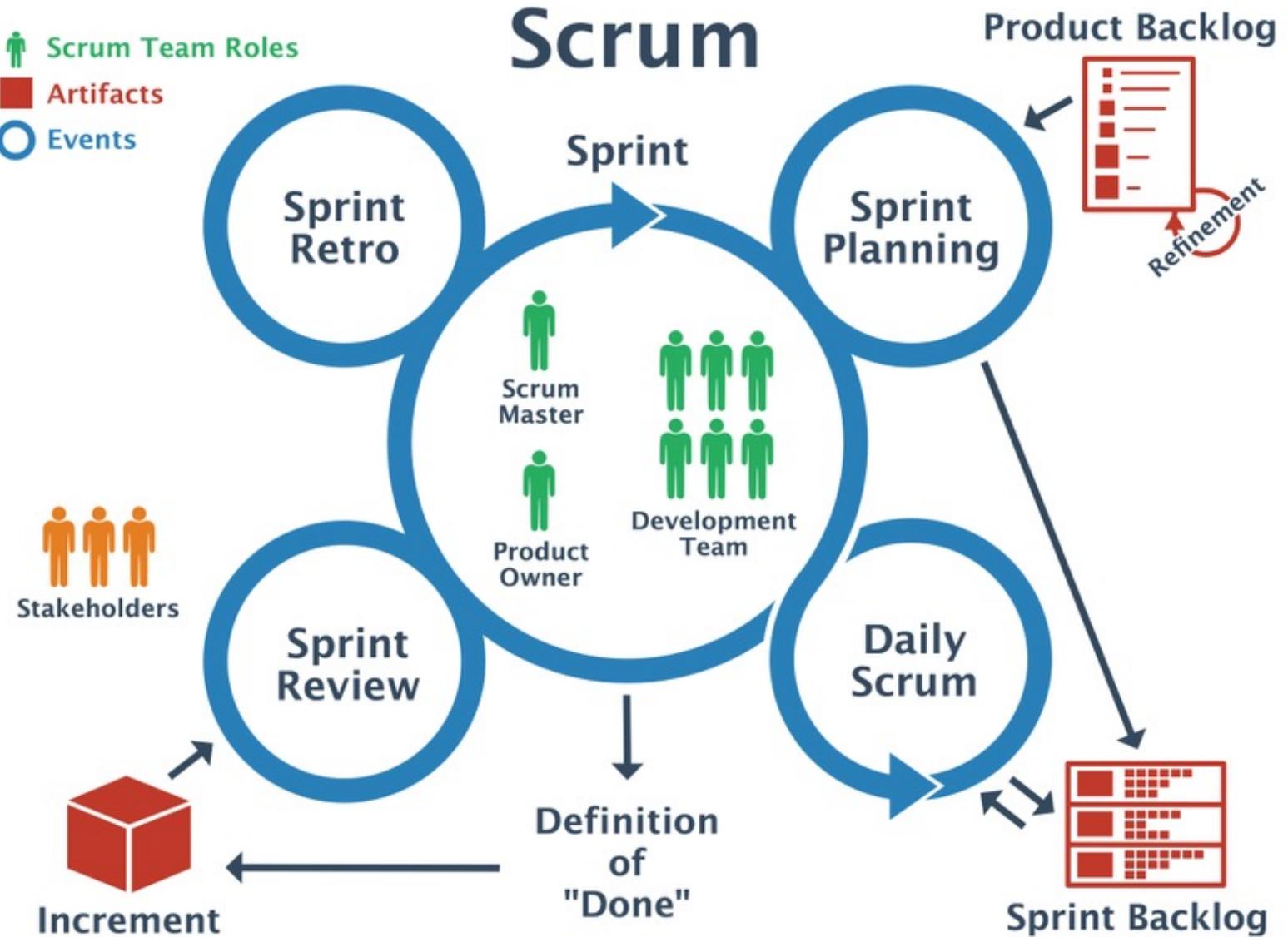


Finished Work



Sprint
Retrospective

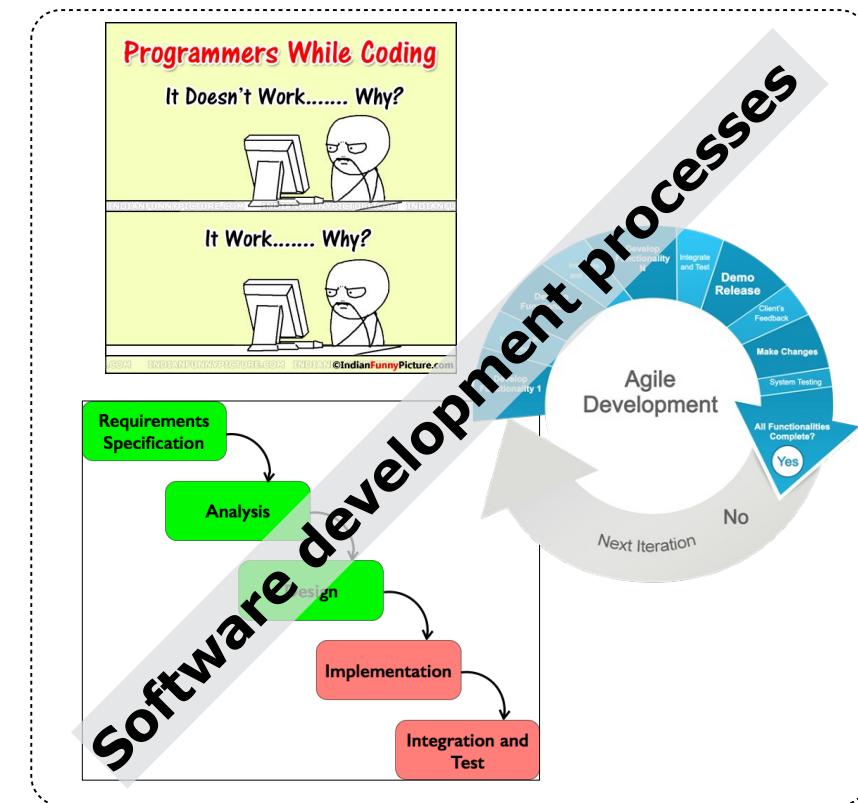
Scrum



According to the July 2016 Scrum Guide™

v2.1 - JordanJob.me

UML can be used in many software development process



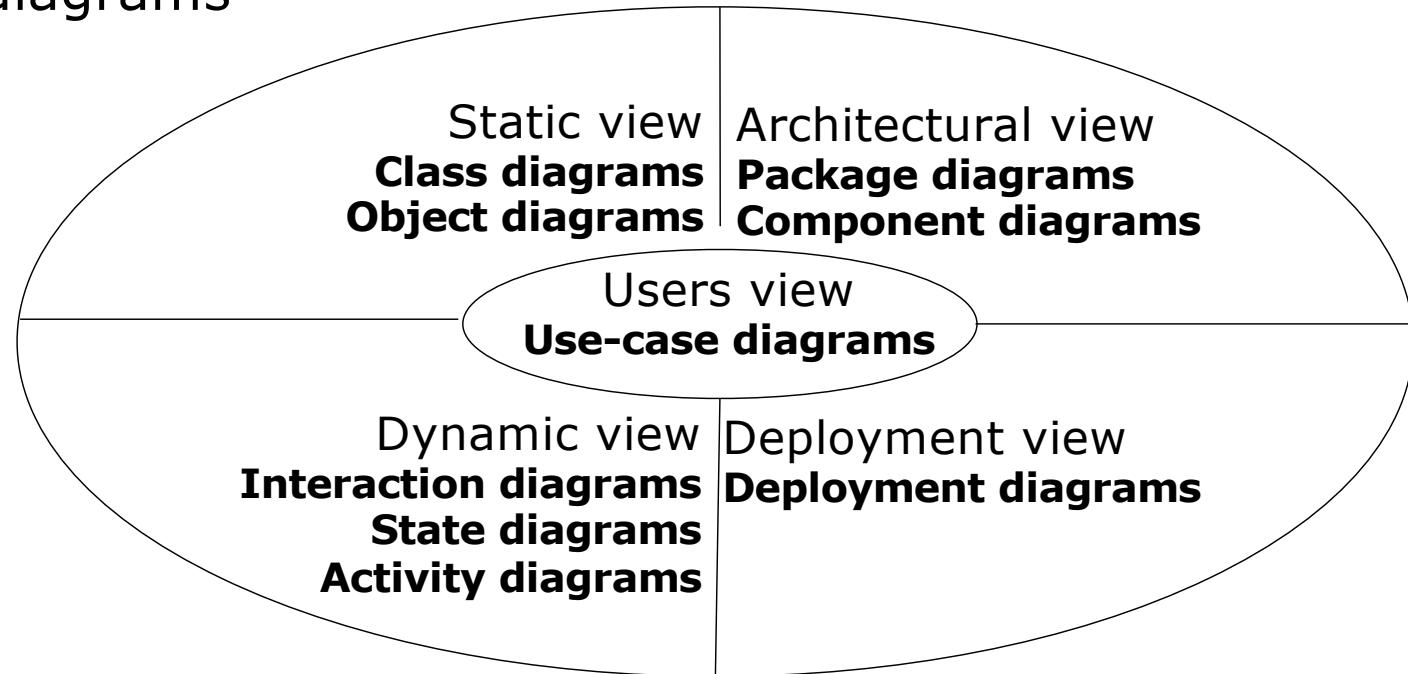
UML diagrams can be applied to several activities

	Requirements	Analysis	Design
Use-case	⊗		
Class, object		⊗	⊗
Activity		⊗	✓
State		⊗	⊗
Interaction		✓	⊗
Component			⊗
Deployment			⊗

✓ : possible usage
⊗ : recommended usage

Requirement modelling

- Use-case diagrams



Software Development Activities

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

Deployment

Installation and training

Maintenance

Post-install review
Support docs
Active support

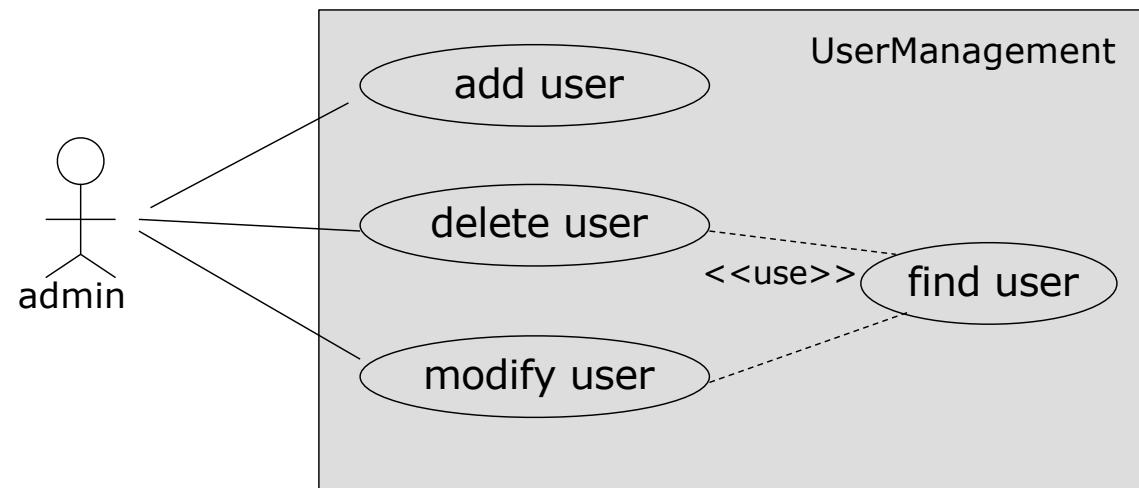
Requirement

- Requirements are capabilities and conditions to which the system - and more broadly, the project - must conform

- Requirement analysis is about describing problems

Use-case diagram

- The first step in requirement analysis is to determine use-cases of the system
- Use-case diagrams
 - allow to **represent the functionalities of the system in the users view**
 - allow to delimit the boundary of the system



Use-case diagram

- With the help of a use case diagram, you can discuss and communicate:
 - The scenarios in which your system or application interacts with people, organizations, or external systems.
 - The goals that it helps those actors achieve.
 - The scope of your system.

User-centred design

- The development of a system should always be centred around the needs of users
 - Understand who are the users
 - Understand the tasks performed by the users
 - Make sure that users are involved in the decision-making process
 - Design the interface well following the needs of the user
 - Users will need to evaluate prototypes and return their comments



Cash register at the supermarket

Interest of user-centred design

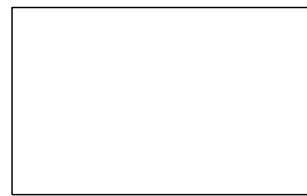
- Meets the actual requirements
- Reduce costs related to changes or maintenance
- Allow to better define the properties in the development
- Reduce learning time
- Reduce training and supporting costs
- Allow efficient use
- Making the system more attractive and better suited to its market

Determining users' characteristics

- Good questions
 - What are their goals?
 - How will they use the software?
 - What is their level of computer literacy?
 - What are their psychological characteristics?
 - What are their habits?

Use-case diagrams

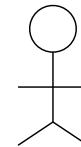
- A use-case diagram consists of three parts
 - The system
 - The use-case
 - The actor
- Graphical representation



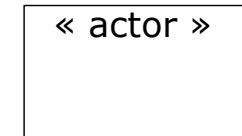
System



Use-case



Actor

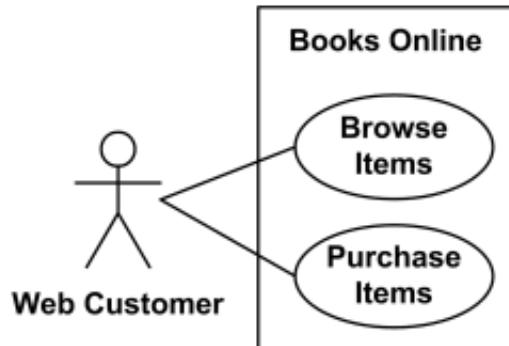


Actor

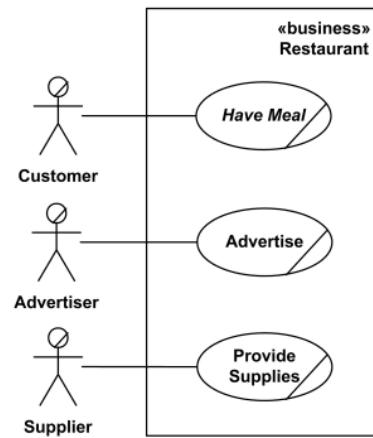
- A use case diagram does not show the detail of the use cases
 - Only summarizes some of the relationships between use cases, actors, and systems.
 - does not show the order in which steps are performed to achieve the goals of each use case

System

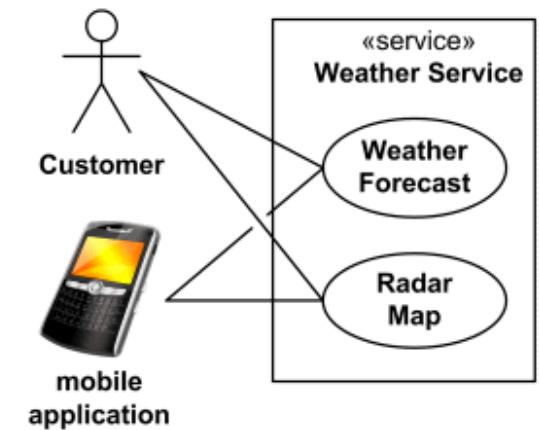
- The system can be any system, not only the software system
- It defines the boundary of the system in a clear and precise manner
 - Not too ambitious
 - Only determine basic functionalities
 - Build a well defined architecture
 - Additional functionality can be added during development



“Books Online” system



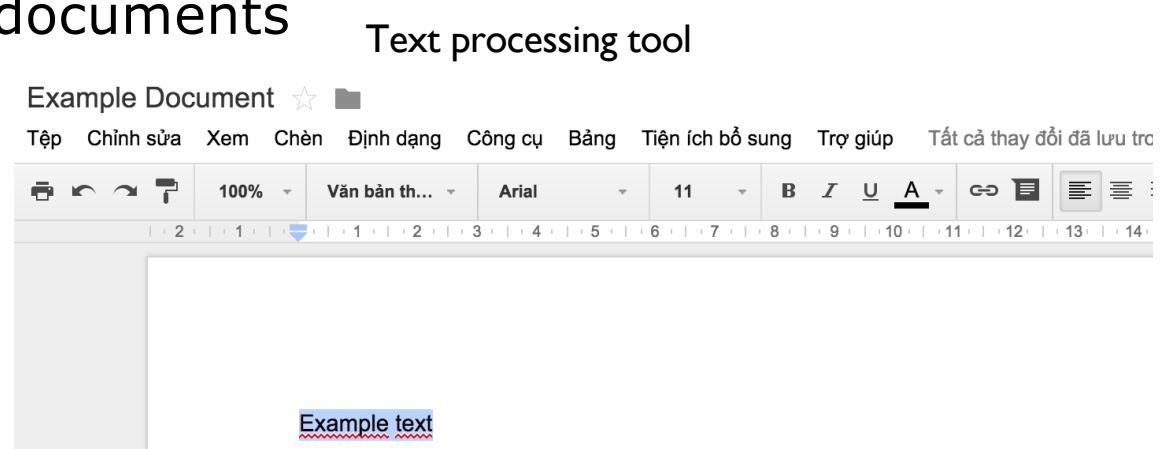
“Restaurant” system



“Weather Service” system

Use-case

- A use-case is a typical interaction or a typical sequence of interactions between the system and its environment
- The objective of a use-case is to model the system
 - according to the perspective of user interacting with the system
 - to accomplish their objectives
- A use-case may can be either large or small
- Example: developing a tool for text processing
 - Some possible use-cases
 - Create a new document
 - Modify an existing documents
 - Delete a document
 - Input new text, ...



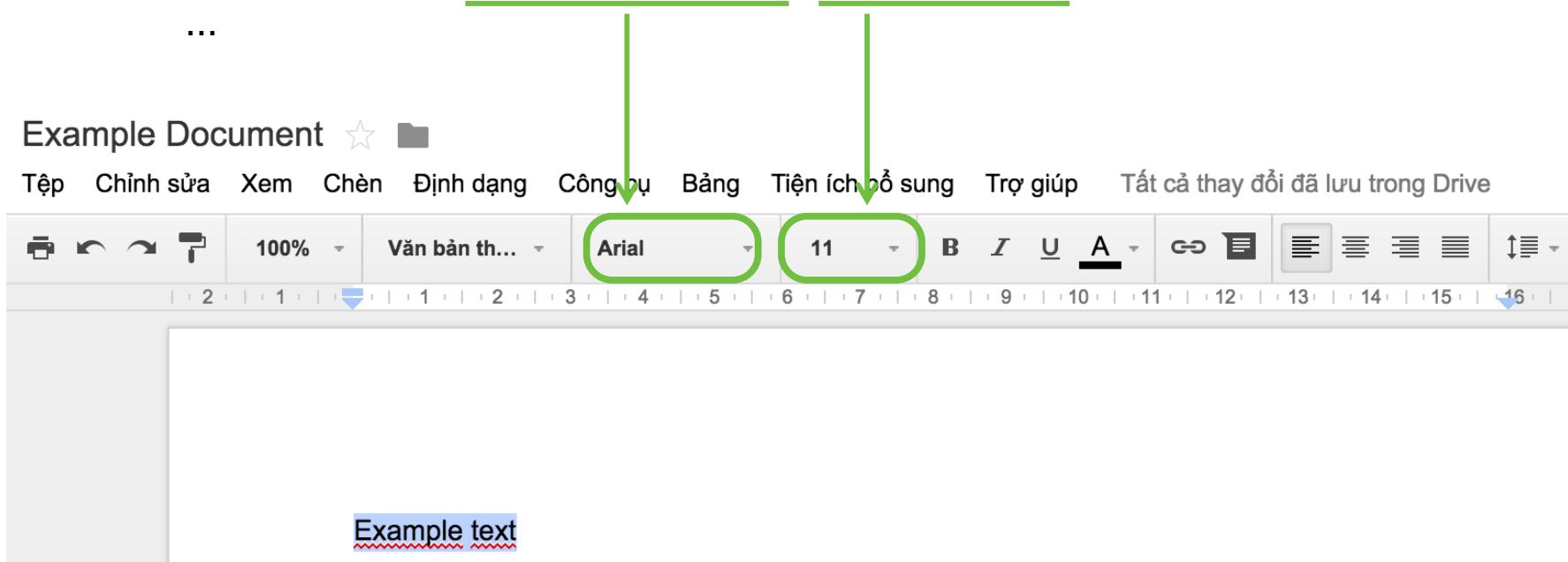
Use-case

- A use-case needs to
 - always correspond to a high level objective
 - describe the interaction between the user and the system, not the operations that the system should perform
 - cover all the steps to follow in performing a given task
 - be written, to the possible extent, independently of the user interface
 - include only the interactions with the system

Use-case

□ Objectives and interactions

- **Objectives** of users: what users expect from the system
- **Interactions** with the system: mechanisms to meet those objectives
- Define the objectives then determine the interactions to achieve objectives
- Example
 - Objective: define the document style
 - Interactions: choose the font, choose sizes, choose the page layout,



Use-case

- Example: developing of an ATM system
- Some interactions in the following scenario
 - Insert the card
 - Enter the PIN code
 - Choose the amount to be withdrawn
 - Confirm the amount
 - Take the card
 - Take the money
 - Take the receipt
- Are all interactions use-cases?



Use-case

- Example (continue)
 - The answer is no
 - Since some interactions such as “confirm the amount” do not meet a goal of the user
 - The goal of the user in this case is to **withdraw money**: this is a use-case



Actors

- An **actor** is a **role** played by the user or an external entity during interaction with the system
- Who or what uses the system
- Actors communicate with the system by sending and receiving messages
- Example
 - Develop a system of cash register at the supermarket
 - Possible actors
 - Client
 - Cashier
 - Manager
 - Inventory manager



Actors

- Distinguishing two notions: **actor** and **user**
 - Multiple users may correspond to a single actor
 - Different cashiers play the same role in the system
 - A user may correspond to several actors
 - A user can simultaneously be a *cashier* and a *manager* in the system



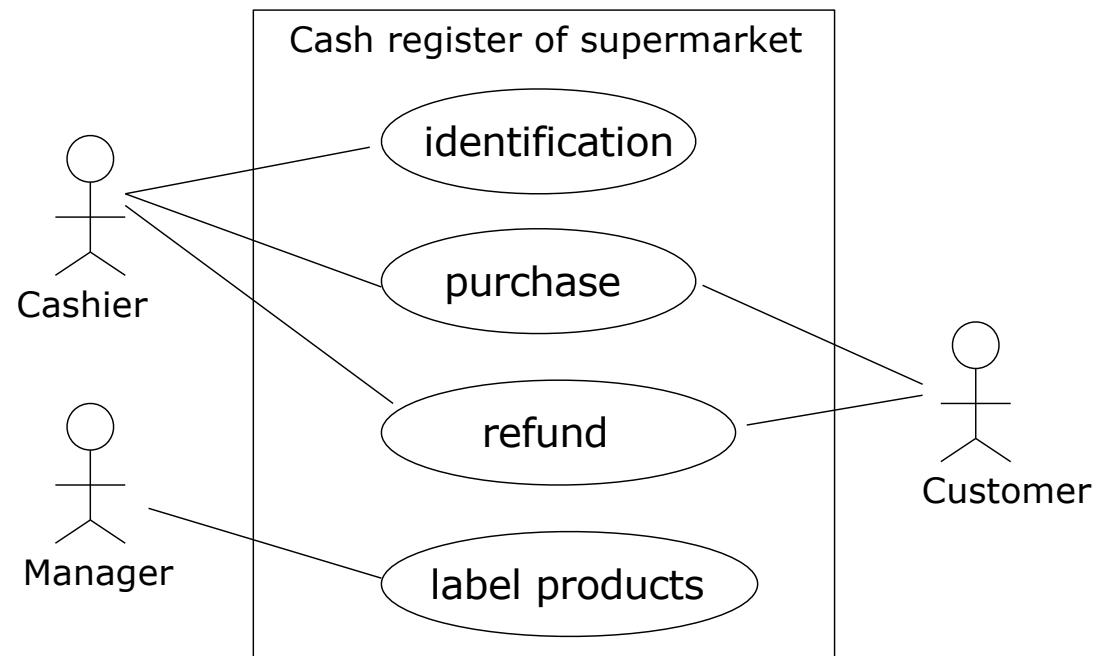
cashier and manager



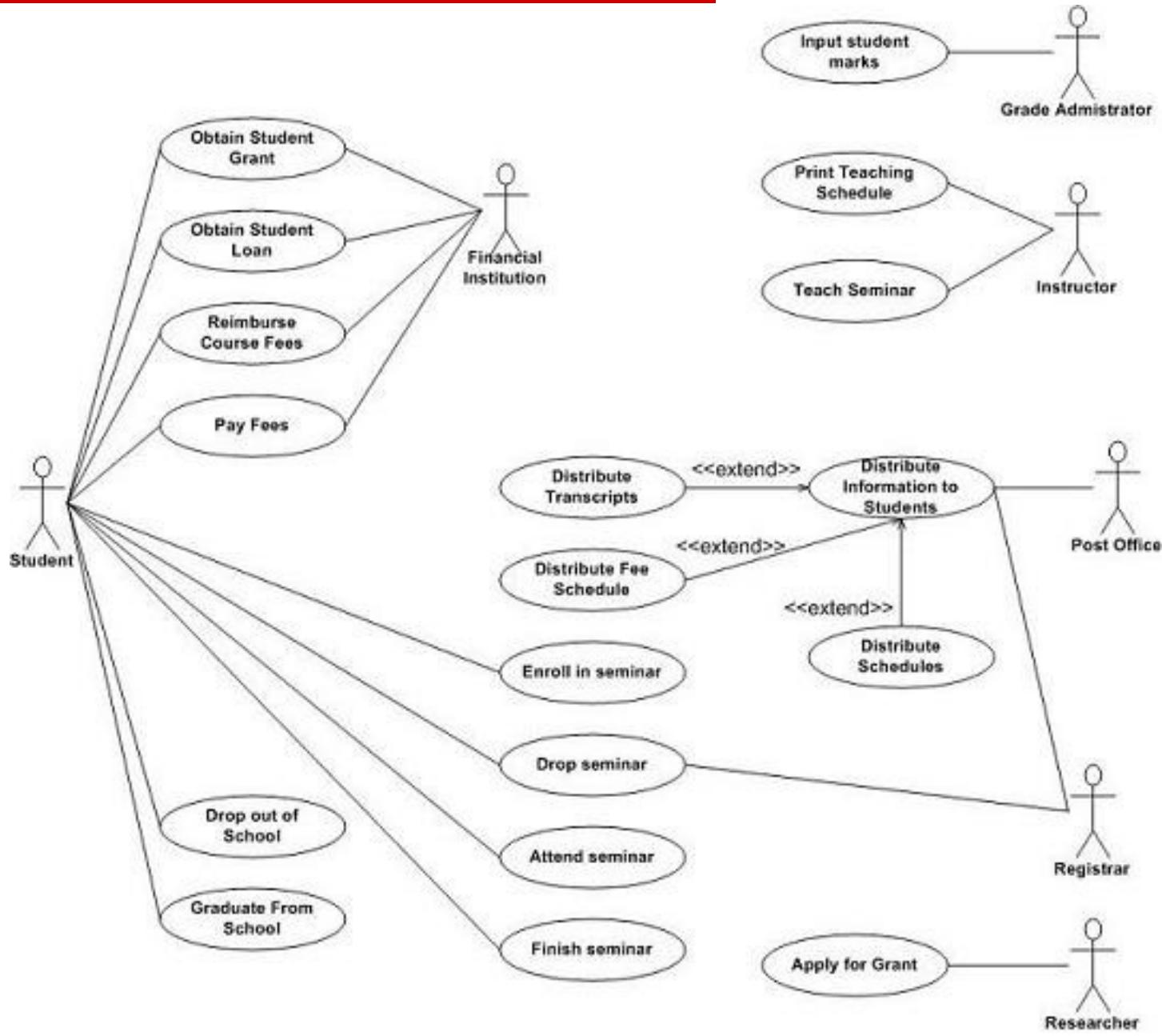
cashier and customer

Actors

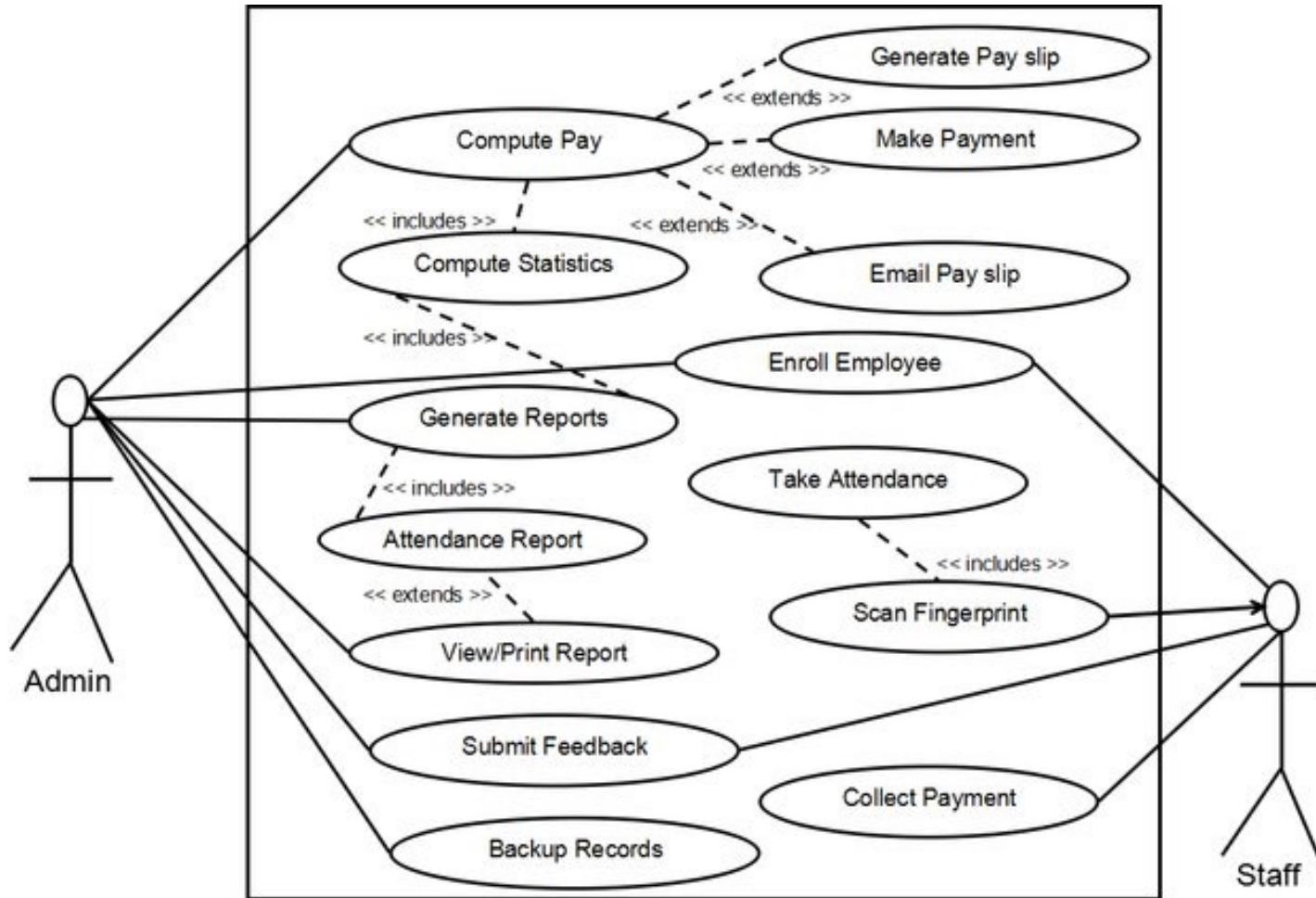
- Questions for identifying the system actors
 - Who will use the main features of the system?
 - Who will need the support of the system to perform its tasks?
 - Who should update, administer and maintain the system?
 - Does the system interacts with other systems?
 - Who or what has interests on the results of the system?



Examples

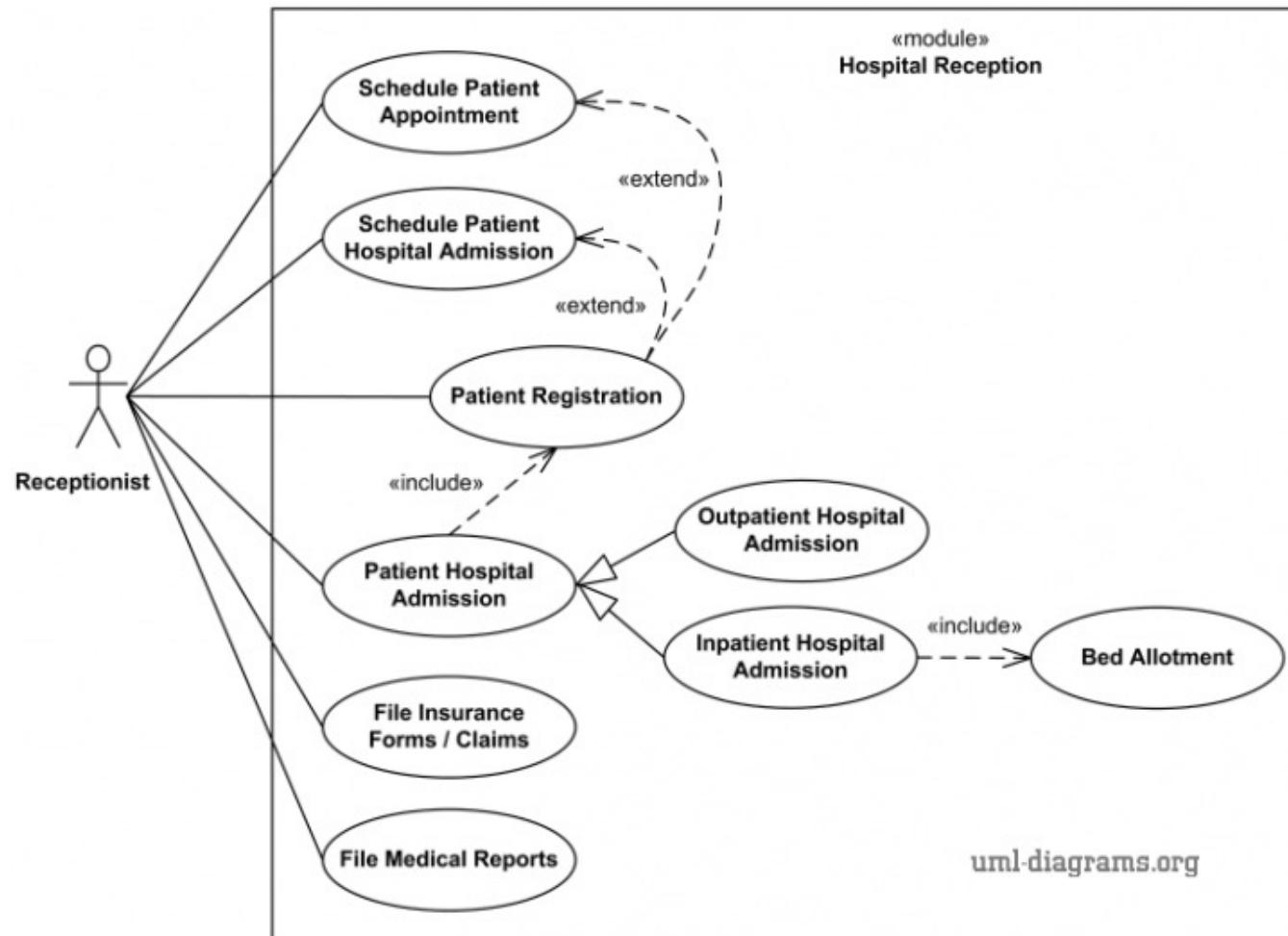


Examples



Use case diagram of the attendance management system

Examples



USE CASE DIAGRAM EXAMPLE FOR HOSPITAL MANAGEMENT

Use-case specification

- Typical specification of a use-case
 - **Use-case**: name of a use-case often begins with a verb
 - **Actors**: list of stakeholders concerning the use-case
 - **Objective**: objective of the use-case
 - **Description**: a brief description of treatment to achieve
- Example
 - **Use-case**: purchase of products
 - **Actors**: Client, Cashier
 - **Objective**: describe a purchase of products by the customer with cash payment
 - **Description**: The clients comes in the box with the selected products. The cashier encodes products, announces the total. The customer pays. The cashier registers the payments.

Use-case specification

- The use-case specification may add
 - the references concerning the specification of the requirement
 - the pre- and post-conditions of the use-case
- Example
 - **Use-case:** purchase of products
 - **Actors:** Client, Cashier
 - **Objective:** describe a purchase of products by the customer with cash payment
 - **References:** R1.2, R3.4
 - **Pre-conditions:** the cashier is identified and authorised
 - **Post-conditions:** the purchase is registered, the payment is made, the receipt is printed
 - **Description:** The client comes in the box with the selected products. The cashier encodes products, announces the total. The customer pays. The cashier registers the payments.

Use-case specification

- A use-case can be specified by adding **scenarios**
- A scenario describes the specific actions of the actors in the system
- A scenario consists of main interactions and exceptional interactions
- The actions can be divided into two flows
 - Flow of actions concerning the actors
 - Flow of actions concerning the systems
- Example
 - A scenario for “purchase products” use-case



Use-case specification

□ Main interactions of “purchase products” scenario

Actions of actor	Actions of system
<ul style="list-style-type: none">• The customer comes to the cash desk with the products to buy• The cashier encodes the identifier of each product If a product has more than one item, the cashier inputs the number of items• After having encoded all of the products, the cashier signals the end of the purchase• The cashier announces the total amount to the customer• The customer pays• The cashier input the amount of money paid by the customer	<ul style="list-style-type: none">• The cash desk displays the description and price of the product This number is displayed• The cash desk calculates and displays the total amount that the customer has to pay• The cash desk displays the balance

Use-case specification

□ Main interactions of “purchase products” scenario (continue)

Actions of actor

- The cashier gives change to the customer and the receipt
- The customer leaves the cash desk with the bought products

Actions of system

- The cash desk prints the receipt
- The cash desk saves the purchase

□ Exceptional interactions of “purchase products” scenario

Actions of actor

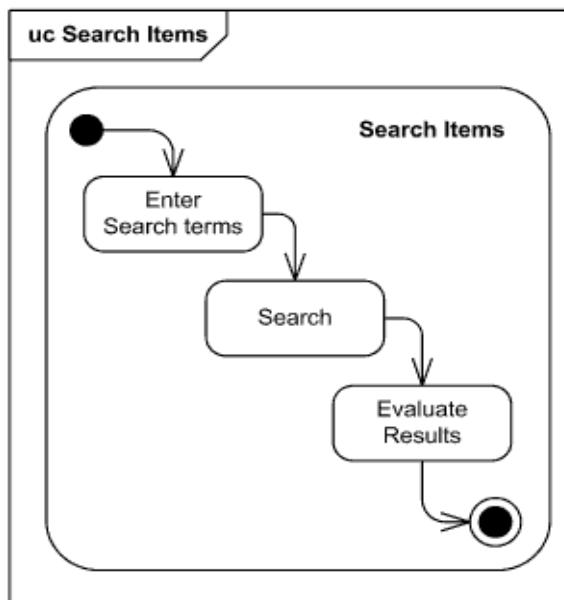
- The customer doesn’t have enough money. The cashier cancel the purchase

Actions of system

- The product identifier is not correct, the system displays the error

Use-case specification

- Remarks
 - The use-case's specification format is only a proposal. Therefore, it is not strict
 - The interactions are described in more detail for important use-cases
 - Use-case's interaction can also be described using activity diagram, state diagram or interaction diagram



Use-case's interactions described in activity diagram

Use-cases identification techniques

- **Software Developer write requirements specification themselves**
 - Lack of human reactions (future users of the system)
- **Interview**



Use-cases identification techniques

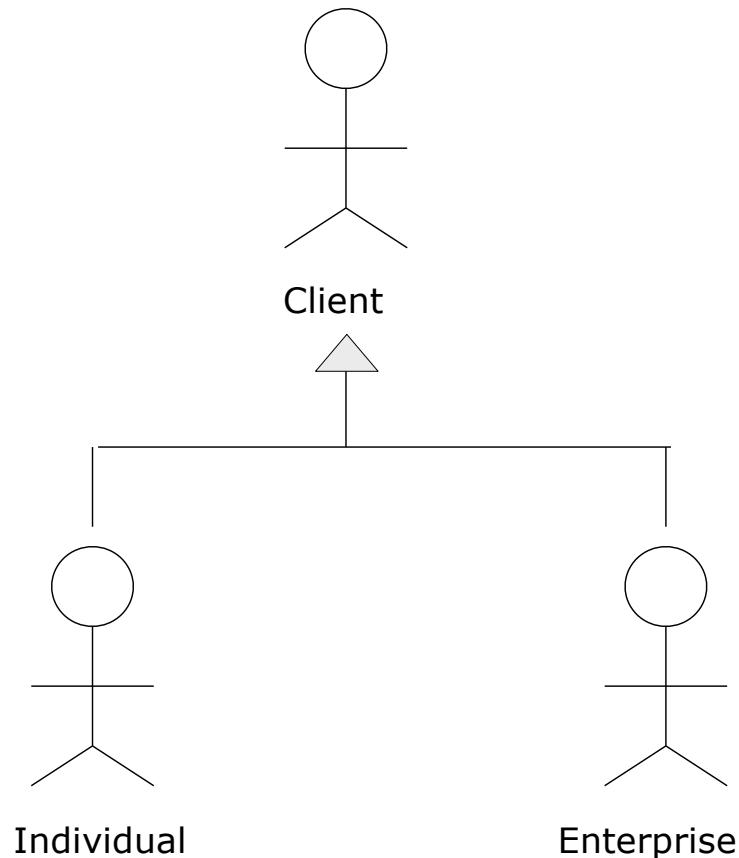
- Workshop (*Organise meetings*)
 - Meeting of all the concerned people of the system to be developed
 - Customers, Users, Software developers
 - Everyone gives their ideas
 - List all the possible actors, use-cases
 - Analyse and describe briefly each use-case
 - Model the use-cases and actors



- Remarks
 - Don't try to search for all the use-cases
 - Other use-cases can appear in the development process

Relations between the actors

- Inheritance between actors



Relationships between use-cases

- Three types of relationship between use-cases
 - Extension
 - Inclusion
 - Generalization
- “extension” relationship
 - Used to specify the **optional interactions**
 - These are **exceptional cases**
 - The case where a use-case is similar to another but it includes **additional actions**
 - The extending use-case must list all the actions in the main use-case and also the supplementary actions

Relationships between use-cases

- “extension” relationship
 - Example: “purchase product with payment by credit card” use-case
 - **Use-case:** purchase products
 - **Actors:** Customer, Cashier
 - **Objective:** describe a purchase of products by the customer with payment by credit card
 - **Description:** The customer comes to checkout with selected products.

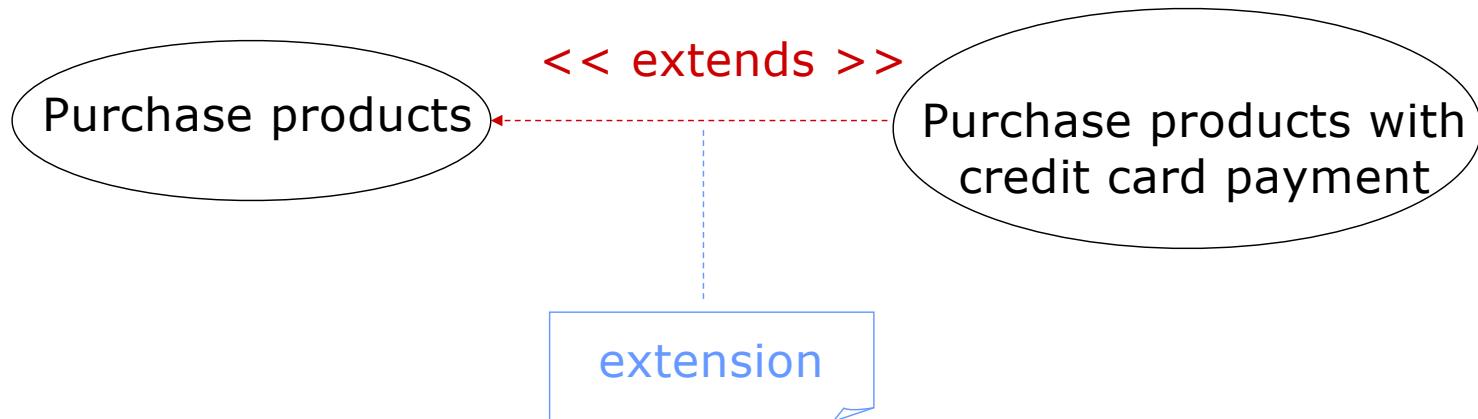


The cashier encodes products, announces the total amount. The customer gives his credit card. The cashier inserts the credit card into the system. The customer types the PIN code. The system verifies the card and then deducts the total of the card.

- This use-case is a variation of the “purchase products” use-case but adds actions relating to the use of credit card.

Relationships between use-cases

- “extension” relationship
 - “Purchase products with credit card payment” use-case is an extension of the “Purchase products” use-case
 - Notation



- **Remarks:** If a use-case is associated with an actor, all extensions are also associated with this actor. This is expressed implicitly in the use-case diagrams.

Relationships between use-cases

- “inclusion” relationship
 - describes a series of joint actions in several cases of different usages
 - if several use-cases share **the same sequence of actions** and this common part is intended to meet a clearly defined goal then the part is described in a separate use-case
 - helps to avoid repeating the same details in different use-cases

Relationships between use-cases

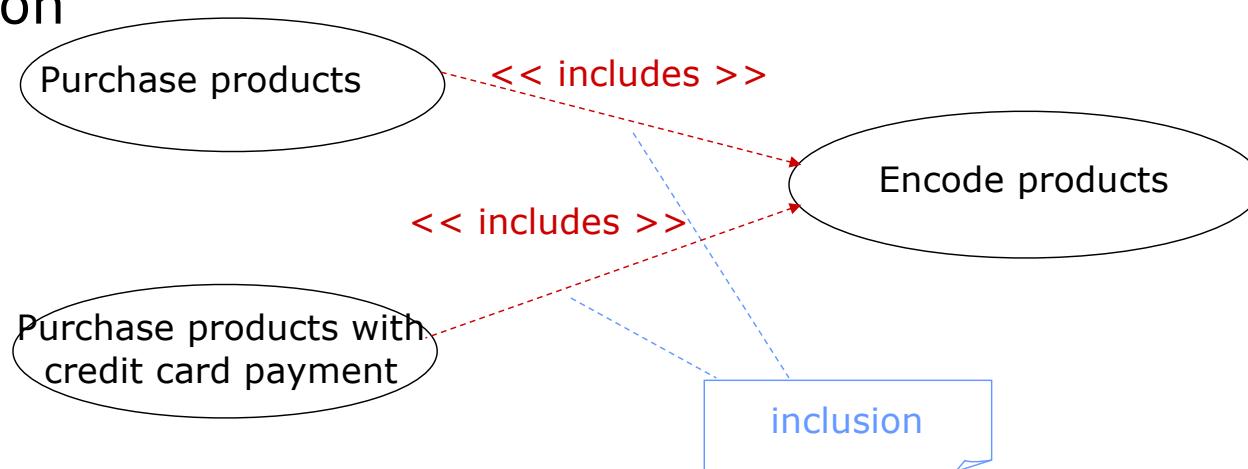
- Example of “inclusion” relationship
 - Suppose we have two use-cases “purchase product” and “purchase products with credit card payment”
 - Both use-cases have the same sequence of actions of encoding products that can be described by the “encode products” use-case

Actions of actor	Actions of system
<ul style="list-style-type: none">• The customer comes to the cash desk with the products to buy• The cashier encodes the identifier of each productIf a product has more than one item, the cashier inputs the number of items• After having encoded all of the products, the cashier signals the end of the purchase• The cashier announces the total amount to the customer	<ul style="list-style-type: none">• The cash desk displays the description and price of the productThis number is displayed• The cash desk calculates and displays the total amount that the customer has to pay

actions of encoding products

Relationships between use-cases

- “inclusion” relationship
 - Example (continue)
 - “encode products” use-case
 - **Use-case:** encode products
 - **Actor:** Customer, Cashier
 - **Objective:** describe the encoding of the products bought by a customer
 - **Description:** The customer comes to checkout with the selected products. The cashier encodes products, announces the total amount to the customer.
 - Notation



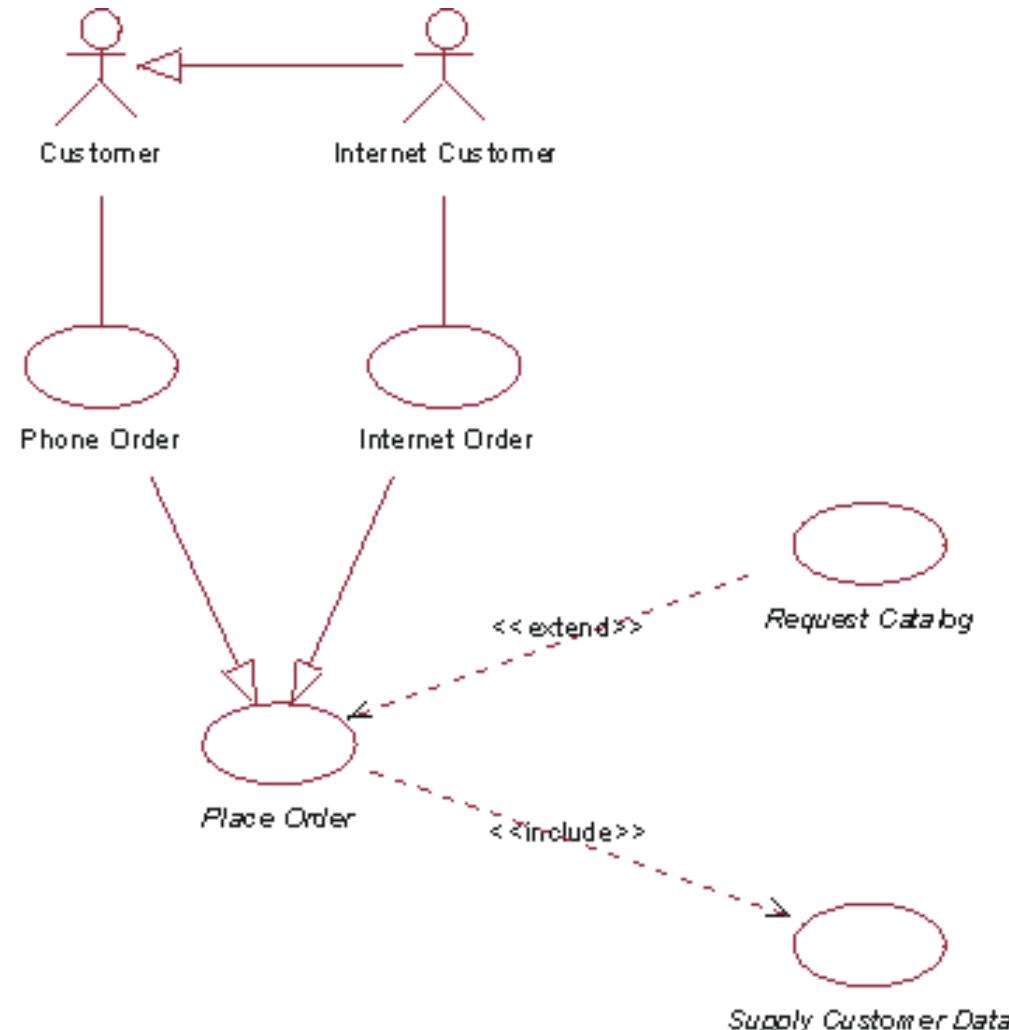
Relationships between use-cases

- Generalization relationship
 - Used when a number of Use Cases all have some subtasks in common, but each one has something different about it that makes it impossible to lump them all in a single use case
 - The generalized and specialized use cases must share the same goal
 - A specialized Use Case may capture an alternative scenario of the generalized Use Case
 - The generalized Use Case must be complete
 - The Specialized use case may interact with new actors.
 - The Specialized use case may add pre-conditions and post-conditions (AND semantics).



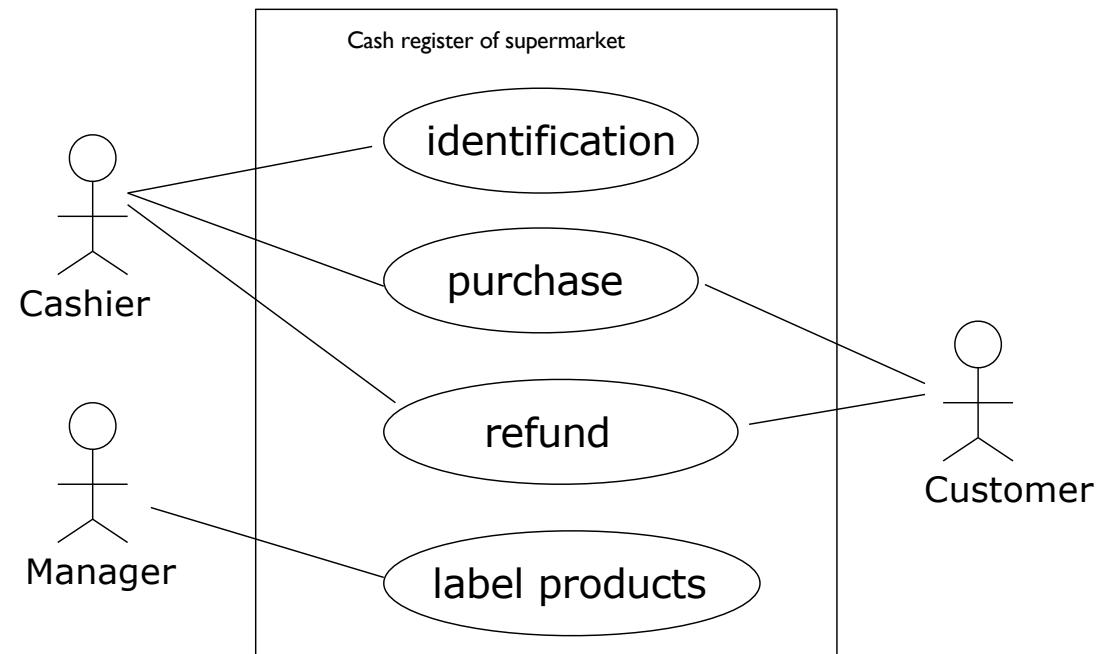
Relationships between use-cases

- Generalization relationship (continue)
 - Example



Building use-case diagrams

- A use-case diagram describes the relationships between the use-cases and actors of the system
- The steps to build a use-case diagram
 - Define the limits of the system
 - Identify the actors
 - Identify the use-cases
 - Define the relationships between use-cases
 - Verify the diagrams



Benefits of Use Cases

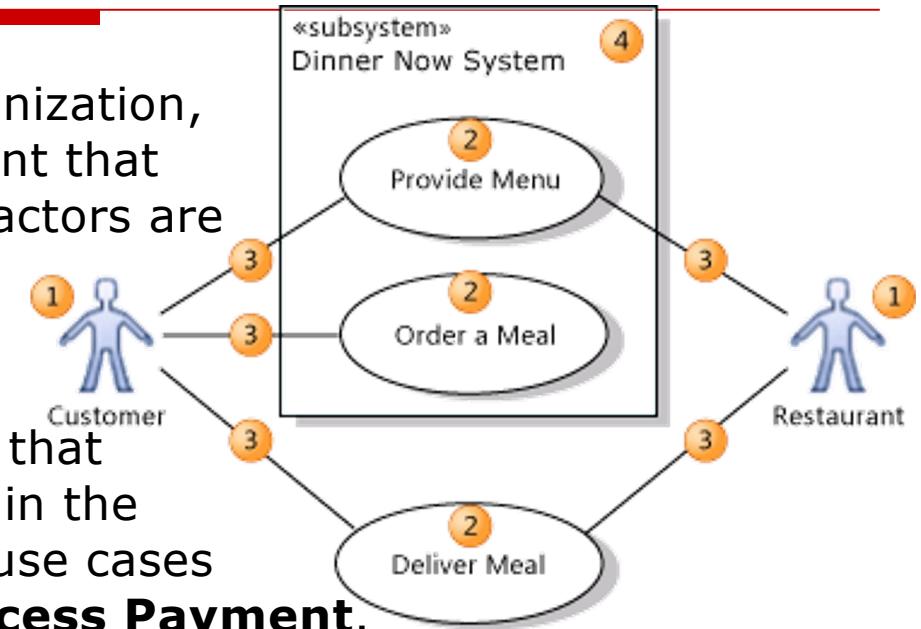
- Captures functional requirements from user's perspective
- Gives a clear and consistent description of what the system should do
- A basis for performing system tests
- Provides the ability to trace functional requirements into actual classes and operations in the system
- Serves as a unit of estimation
- The smallest unit of delivery
 - Each increment that is planned and delivered is described in terms of the Use Cases that will be delivered in that increment

Use Cases vs. Requirements

- A requirements document states what the system shall do. Use Cases describe actions that the users take and the responses that the system generates
- Use Cases are sometimes used as a means to elicit requirements
- Requirements may be effectively documented as Use Cases
 - Better traceability
 - Easier user validation of functional requirements
 - Helps structuring the users manuals
 - A tool for finding classes
- Use-Cases are not well suited for capturing non-functional requirements.
- We will view use-cases as one piece of our software requirements specification

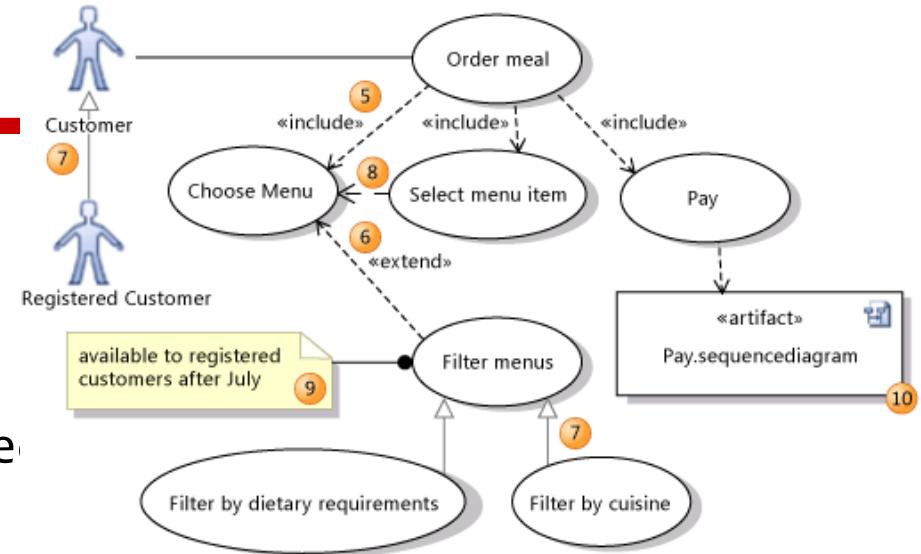
SUMMATIVE Elements

- An *actor* (1) is a class of person, organization, device, or external software component that interacts with your system. Example actors are **Customer, Restaurant, Temperature Sensor, Credit Card Authorizer.**
- A *use case* (2) represents the actions that are performed by one or more actors in the pursuit of a particular goal. Example use cases are **Order Meal, Update Menu, Process Payment.**
- *Association:* On a use case diagram, use cases are associated (3) with the actors that perform them.
- Your *system* (4) is whatever you are developing. It might be a small software component, whose actors are just other software components; or it might be a complete application; or it might be a large distributed suite of applications deployed over many computers and devices. Example subsystems are **Meal Ordering Website, Meal Delivery Business, Website Version 2.**
- A use case diagram can show which use cases are supported by your system or its subsystems



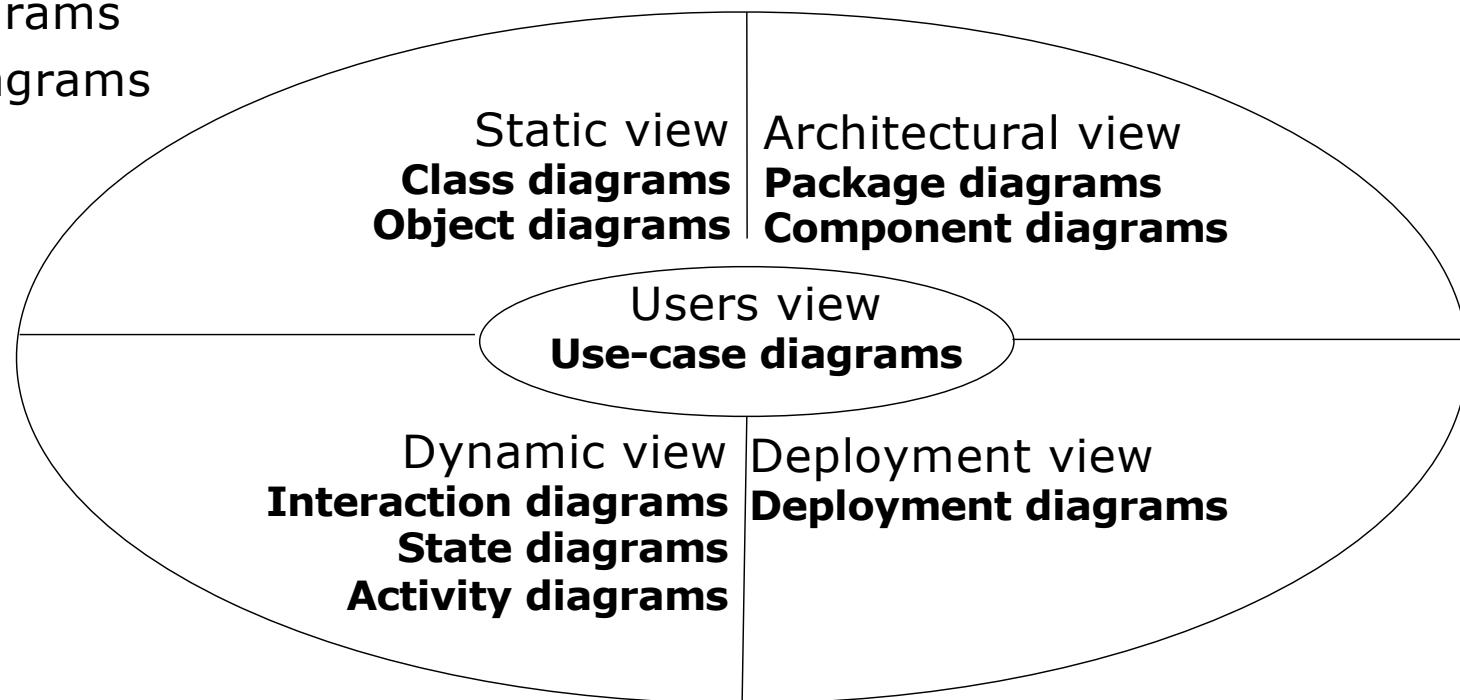
SUMMATIVE Structuring

Include	An including use case calls or invokes the included one.
Extend	An extending use case adds goals and steps to the extended use case.
Inheritance	<p>Relates a specialized and a generalized element. The generalized element is at the arrowhead end.</p> <p>A specialized use case inherits the goals and actors of its generalization, and may add more specific goals and steps for achieving them.</p> <p>A specialized actor inherits the use cases, attributes and associations of its generalization, and may add more.</p>
Dependency	Indicates that the design of the source depends on the design of the target.
Comment	Used to add general notes to the diagram.
Artifact	An artifact provides a link to another diagram or document.



Modelling static structure

- Class diagrams
- Object diagrams



Main Software Development Activities

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

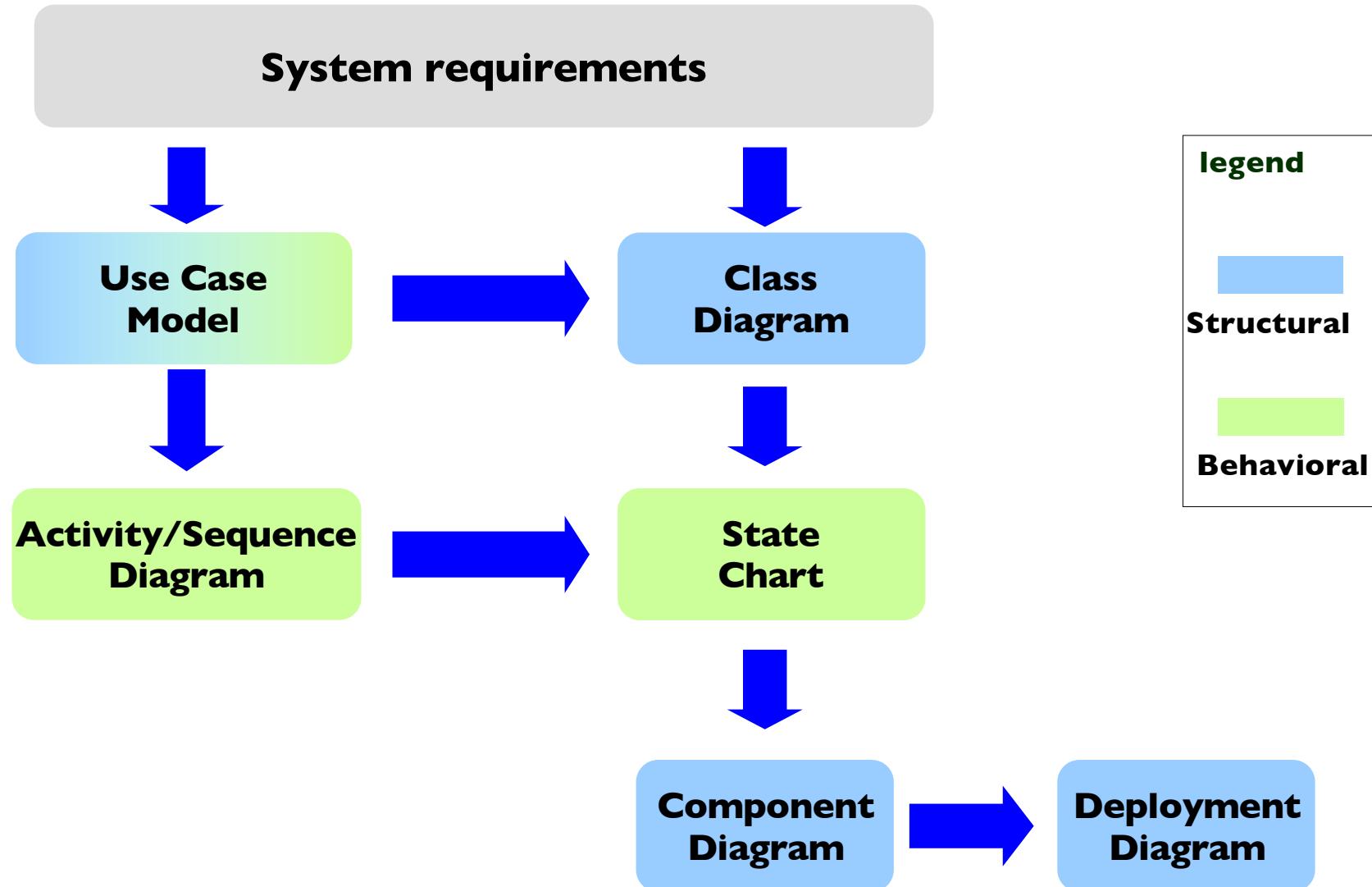
Deployment

Installation and training

Maintenance

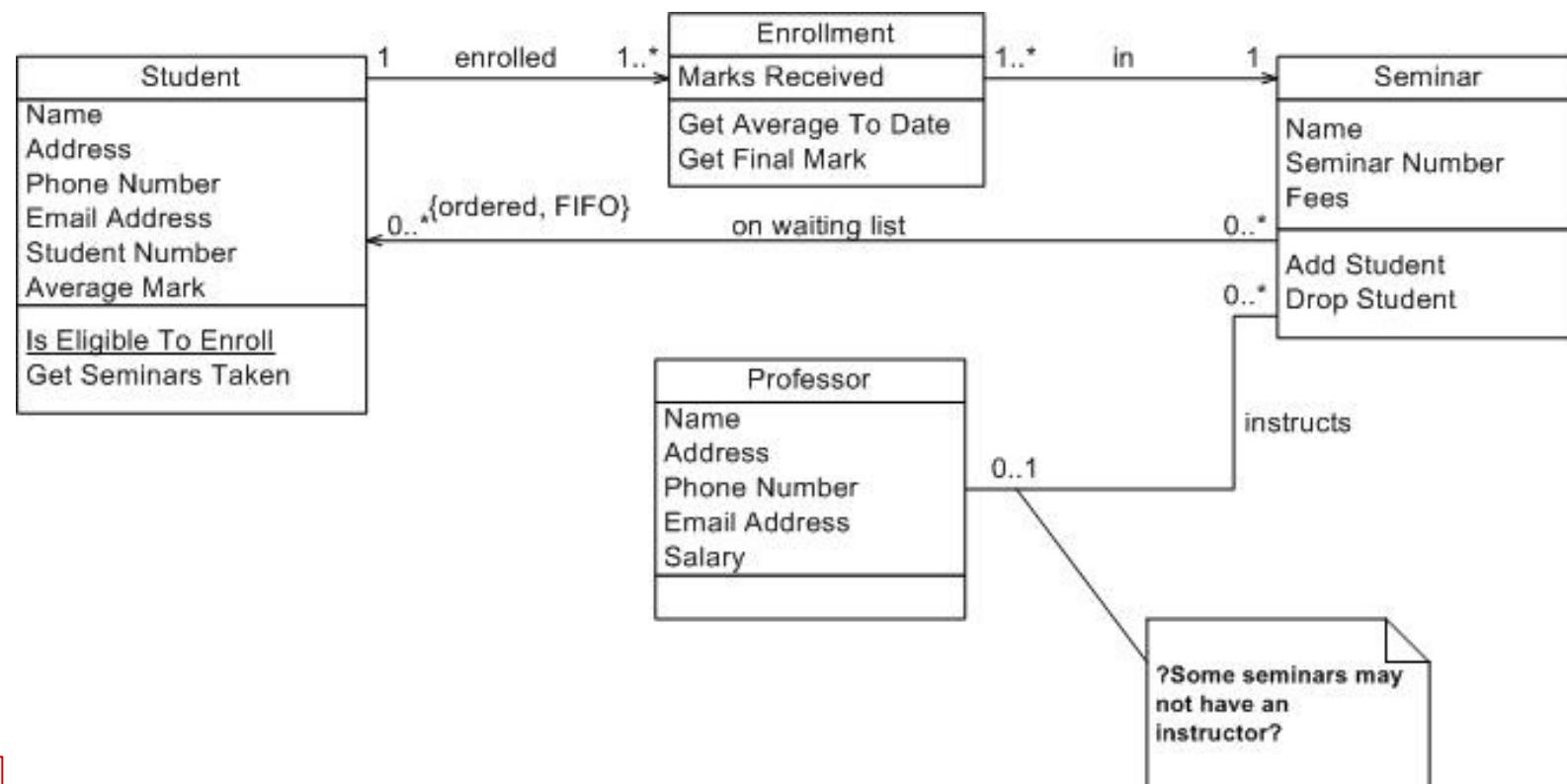
Post-install review
Support docs
Active support

Design Process



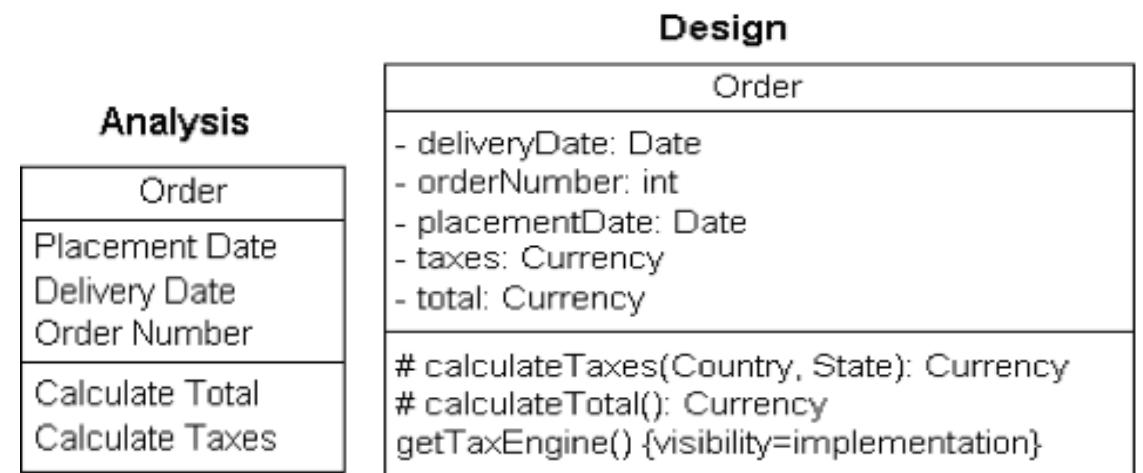
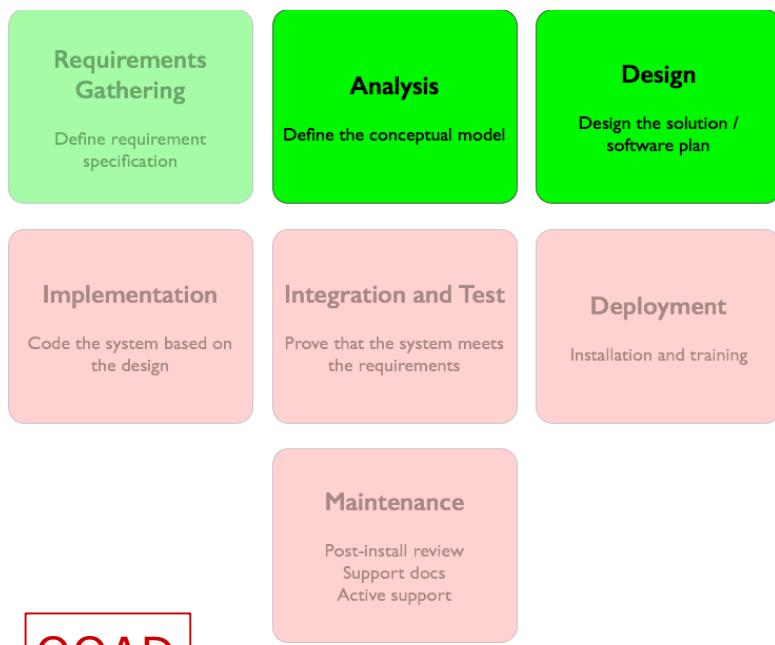
Class diagrams

- Class diagrams
 - consist of a set of classes, interfaces and their relationships
 - represent the **static view** of the system
 - can produce / build the **skeleton** of the system
- Modelling class diagrams is the **essential step** in object-oriented design



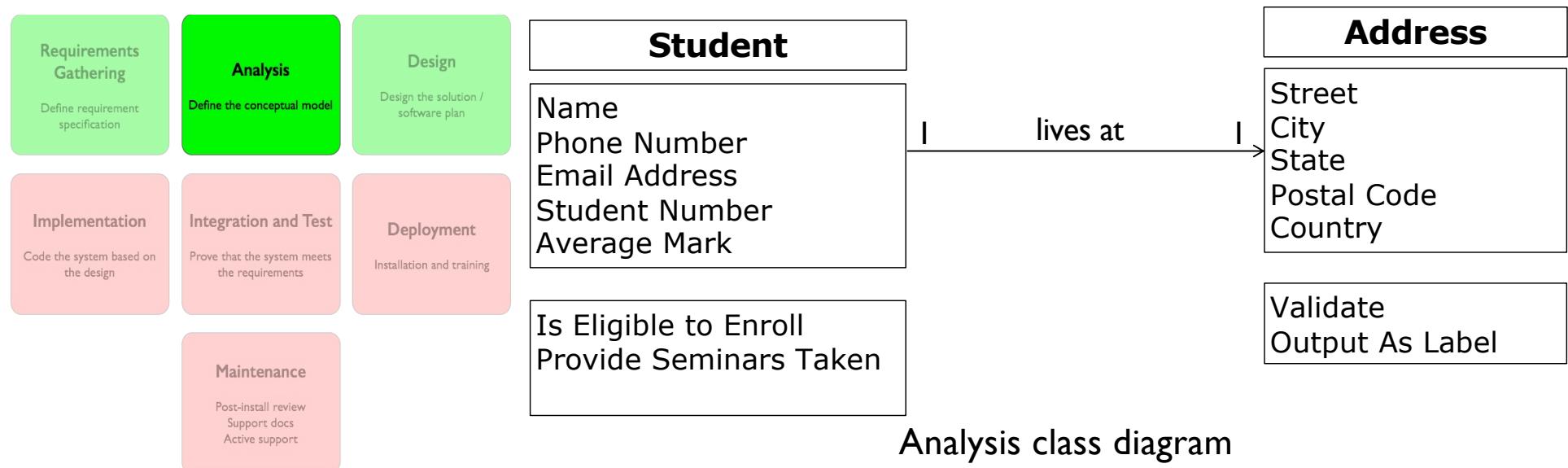
Analysis class diagram v.s Design class diagram

- Two main types of class diagrams
 - Conceptual/Analysis class diagram (domain model)
 - is developed in the analysis phase
 - describes the system from the “user point of view”
 - Design class diagram
 - is developed in the design phase basing
 - describes the system from the “software developer point of view”



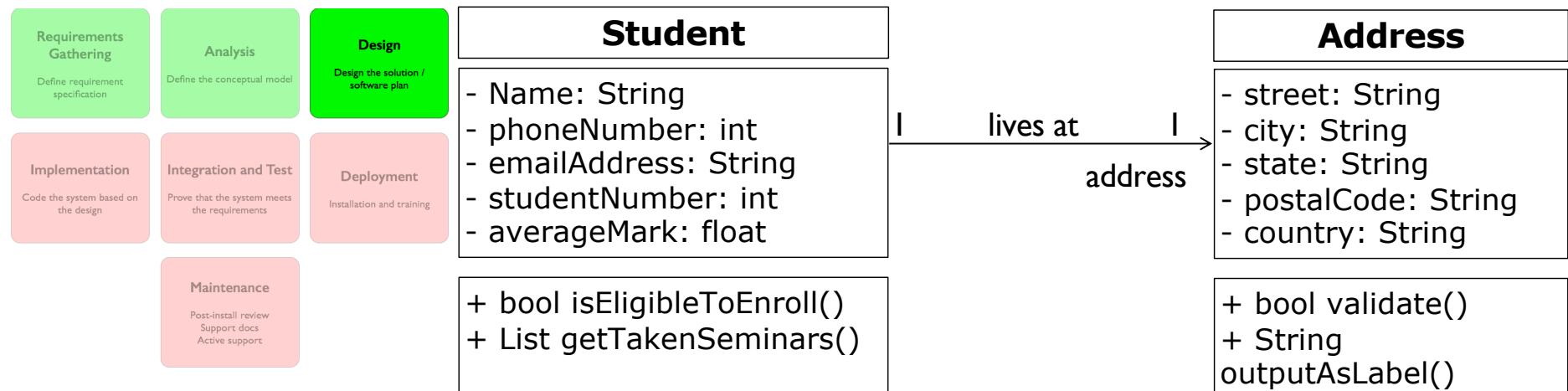
Analysis Class Diagram

- Conceptual/analysis class diagram (domain model)
 - is constructed in the analysis phase
 - captures the concepts recognized by user/customer/stakeholder
 - doesn't contain information of how the software system should be implemented

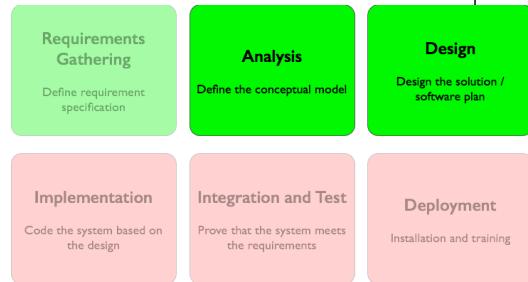
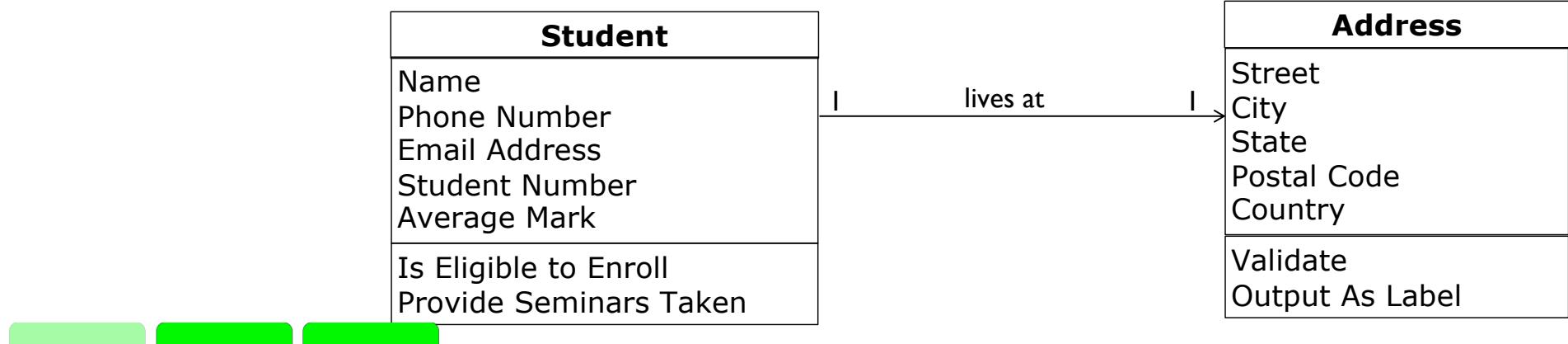


Design Class Diagram

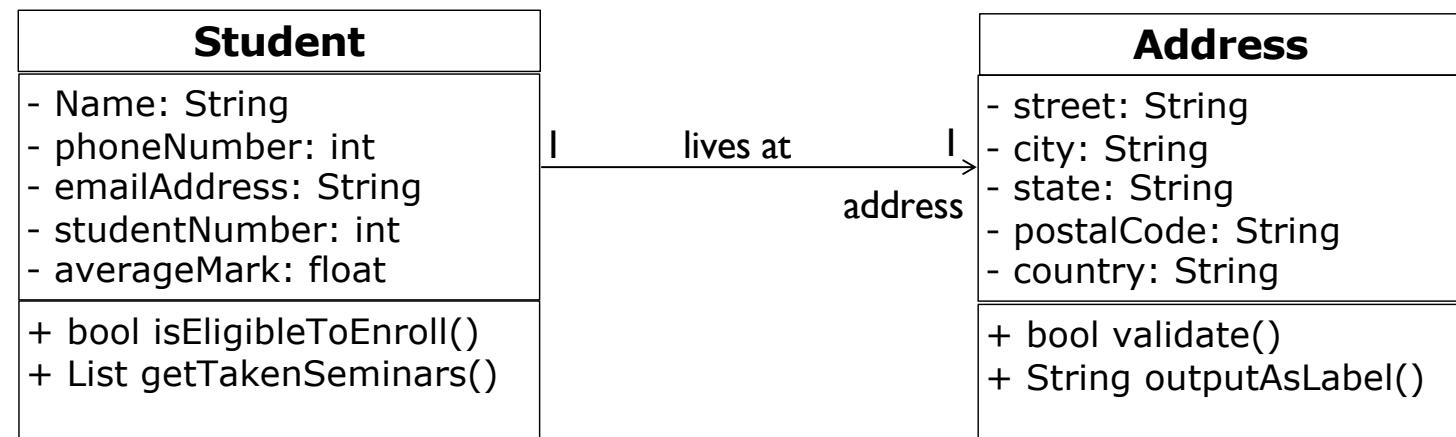
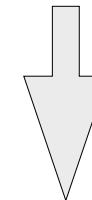
- Design class diagram
 - is construct in the design phase
 - a detail version of the analysis class diagram
 - an analysis class may correspond to several design classes
 - contains information about how the software system should be implemented
 - attributes' and methods' visibility
 - attributes' and methods' name conform to the target programming language



Analysis Class Diagram v.s. Design Class Diagram



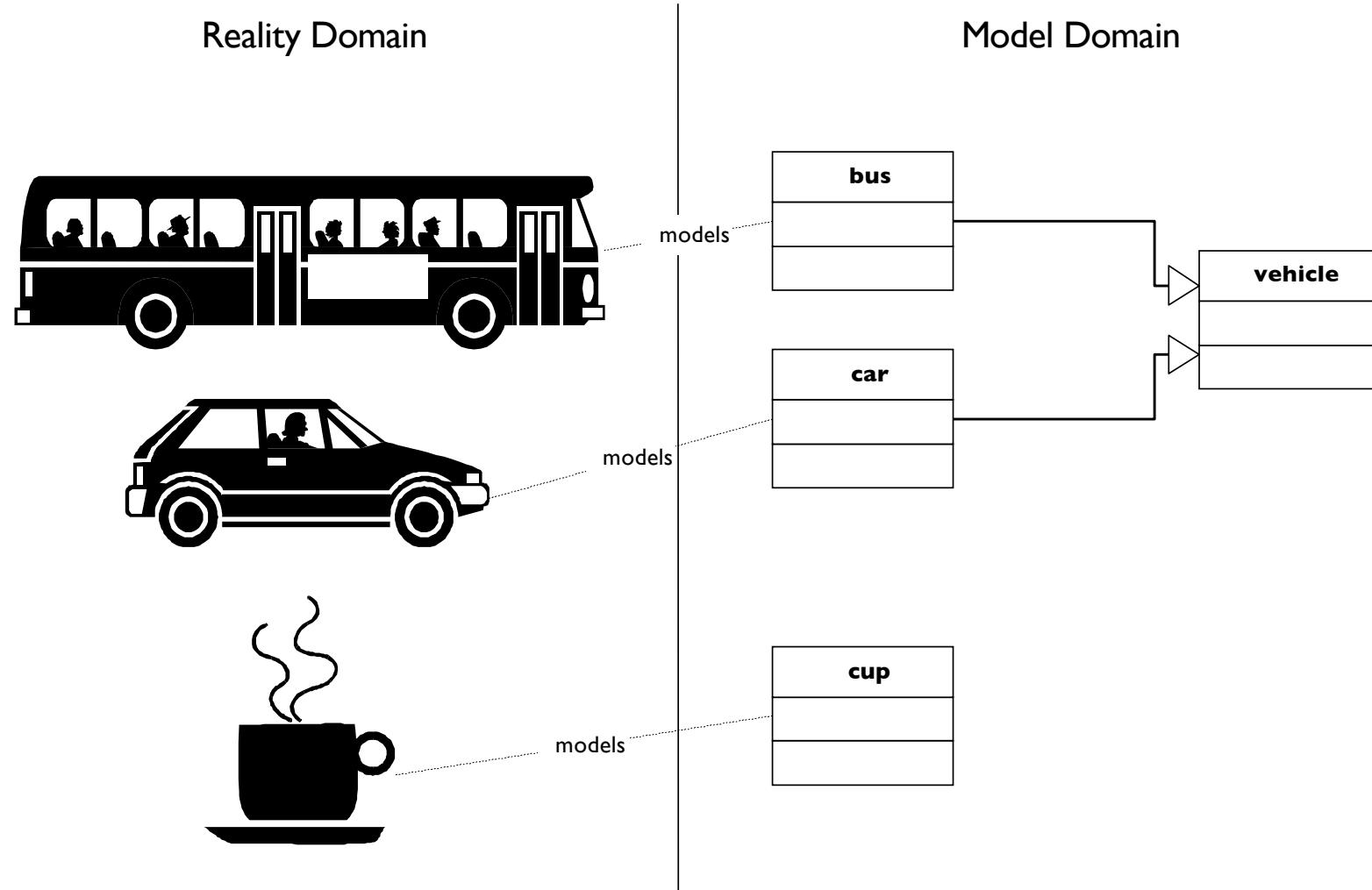
Analysis class diagram



Design class diagram (for Java implementation)

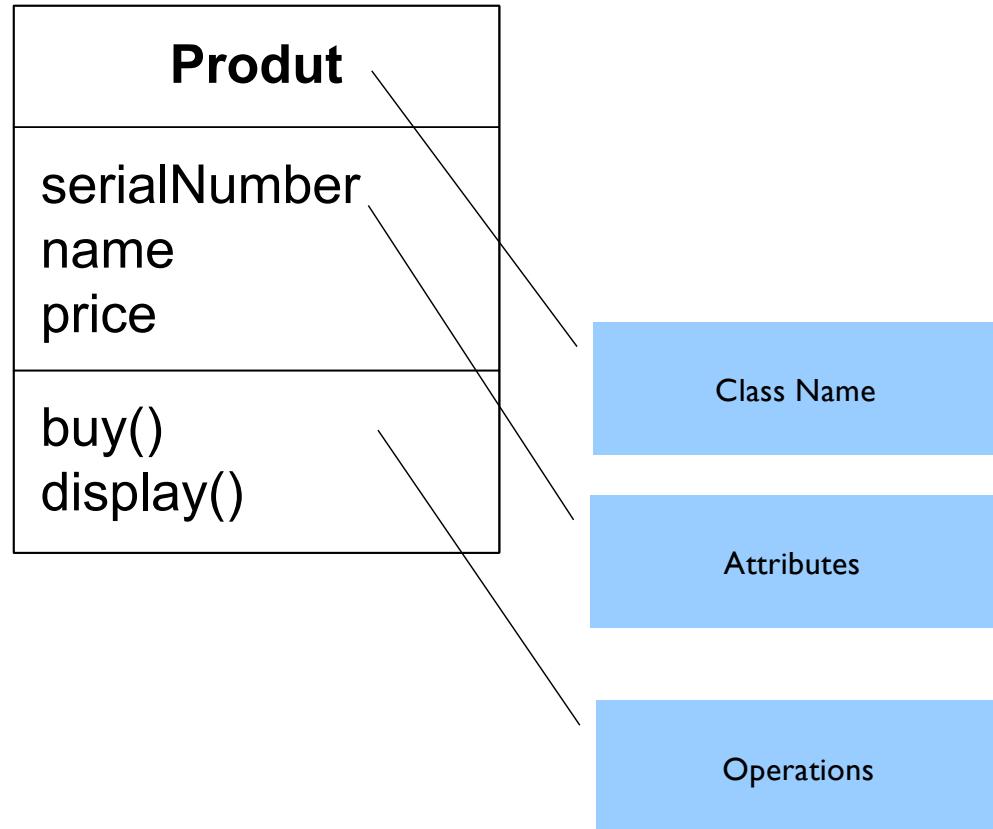
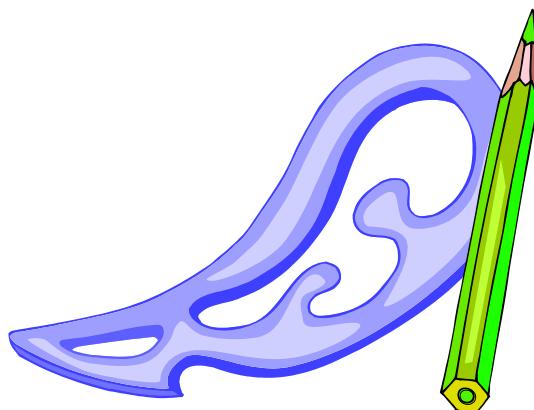
Object-Oriented Approach

- Objects are abstractions of real-world or system entities



Classes

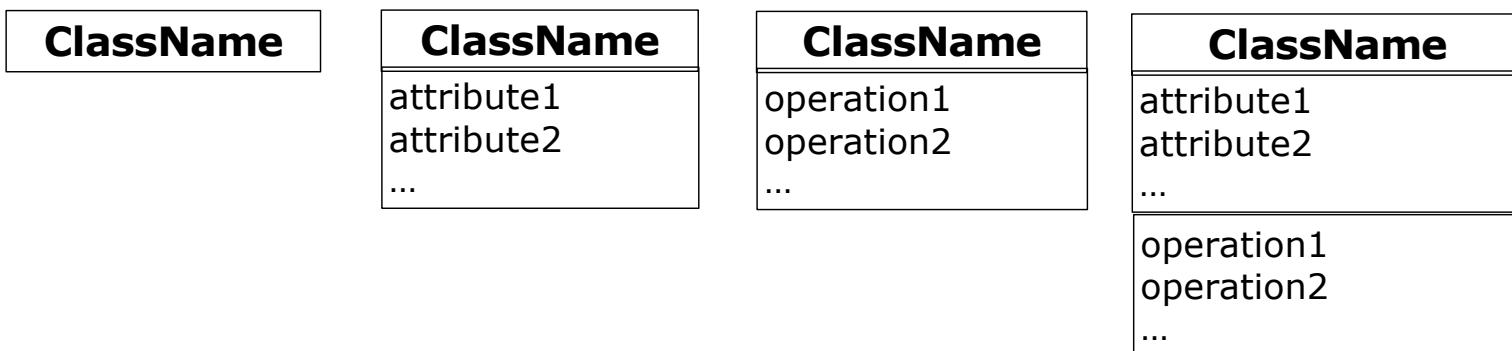
- A class is a template for actual, in-memory, instances



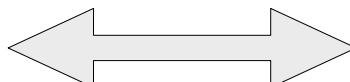
Class

- UML class

- represents the class or interface concept of object-oriented programming language
- consists of a set of attributes and operation
- can be graphically represented in several forms



ClassName



```
class ClassName {  
...  
};
```

C++

```
public class ClassName {  
...  
};
```

Java

Attributes

- Attributes represent the necessary data of class instances
- Attributes can have
 - a type
 - simple type
 - number : integer
 - length : double
 - text : string
 - complex type
 - center : point
 - date : Data
 - A value by default
 - number : integer = 10
 - A list of possible value
 - color : Color = red {red, blue, purple, yellow}

Person
name : string
firstName : string
dateOfBirth : Date
nbChildren : integer = 0
married : Boolean = false
profession : string = « not defined »

Attributes

- Identifying attributes
 - Attributes are **numbers** or **strings**
 - Since attributes represent the characteristics of the objects
 - Distinguishing between attributes and classes
 - If a characteristic of a class is not capable of doing something, then this is possibly an attribute
 - If we doubt that an attribute is a class, then it is considered as a class
 - Does that *salary* is an attribute of *Person* class?
 - If in doubt, then we consider that the *salary* is a class.

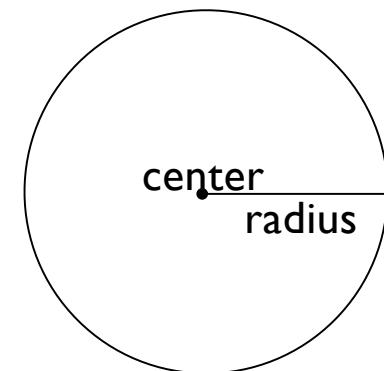
Operations / Methods

- Operations represent the **behaviours** of instance of the class
- The behaviour of a class includes
 - The **getters** and **setters** that manipulate the data of class instances
 - A certain number of tasks associated with the **responsibility** of the class

- Operations can have
 - a name
 - area, calculate, ...
 - a returned type
 - area() : double
 - arguments with type
 - move(p : Point)

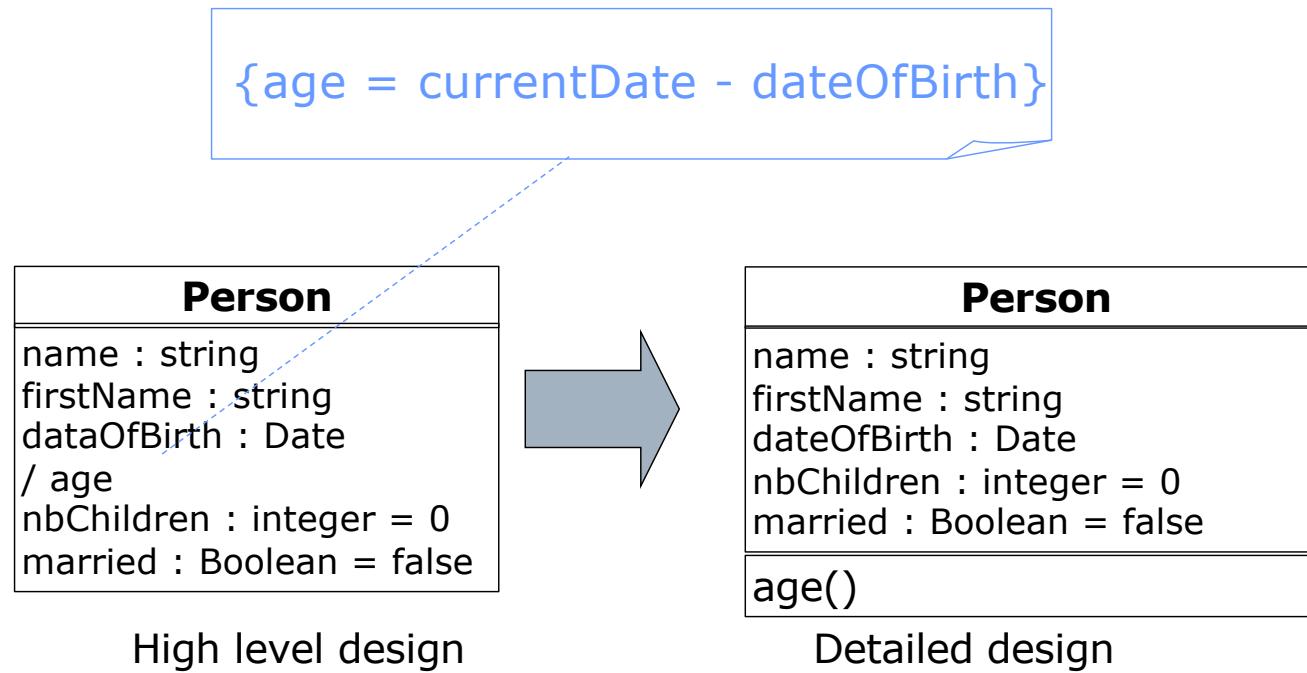
Circle
center : Point
radius : double

area() : real
move(p : Point)



Derived attributes

- Attributes can be deducted from other attributes
 - age of a person can be derived from date of birth

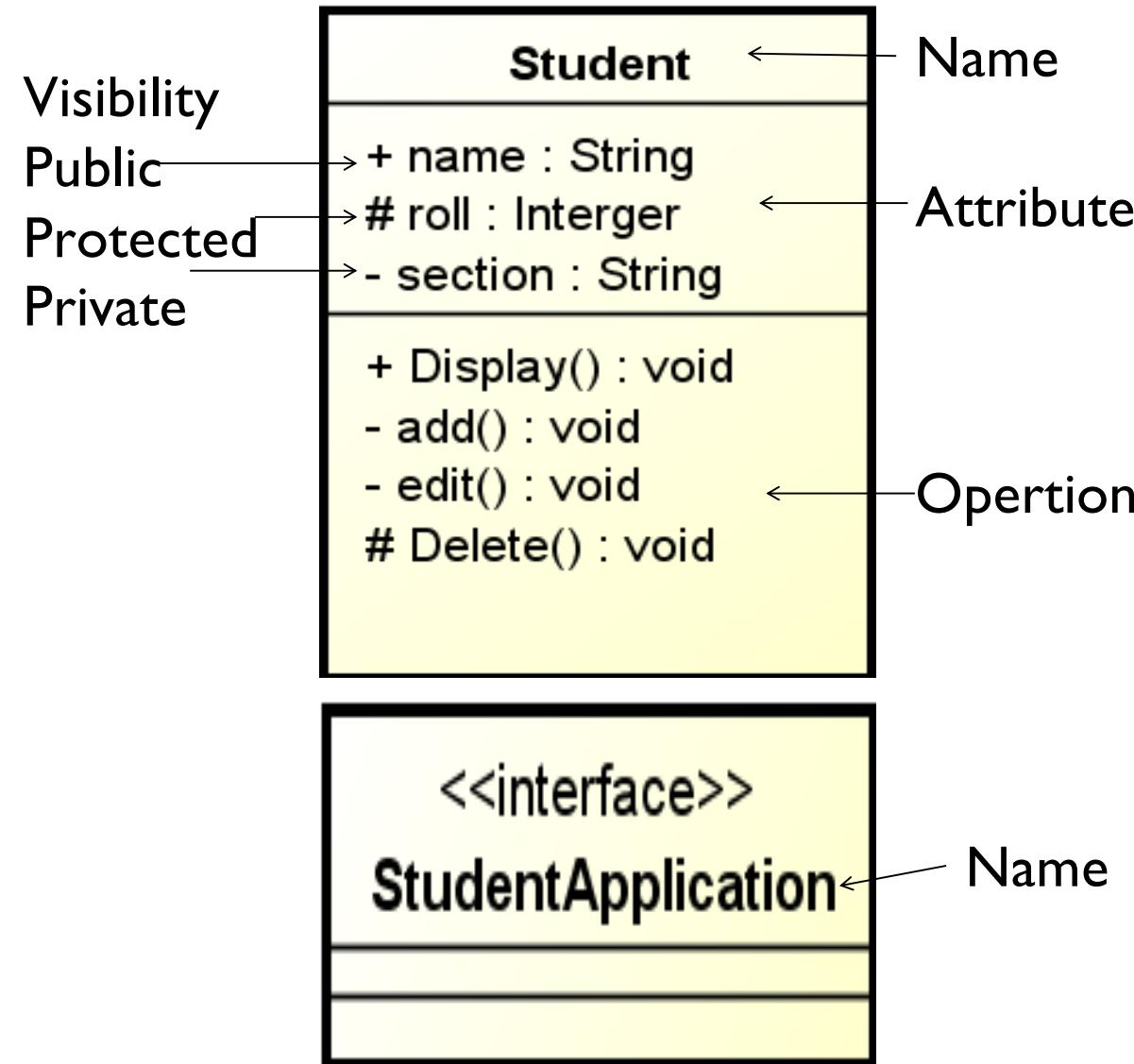


Visibility

- Attributes and operations have the visibility
 - Public
 - visible outside the class
 - notation " + "
 - Protected
 - visible only to objects of the same class and objects of sub-classes
 - notation " # "
 - Private
 - visible only to objects of the class
 - notation " - "

Shape
- origin : Point
+ setOrigin(p : Point) + getOrigin() : Point + move(p : Point) + resize(s : real) + display() # pointInShape(p : Point) : Boolean

Structural things





OOAD

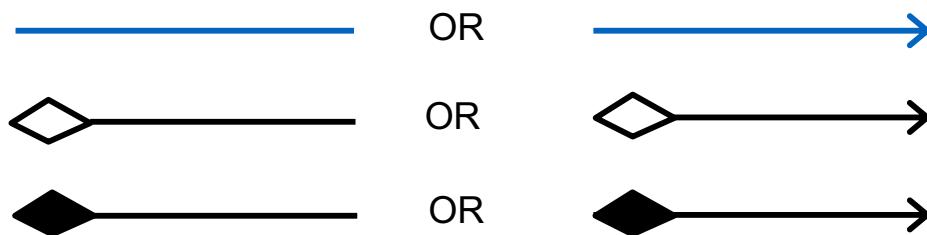
Relationship types

- Relationships between classes
 - Association
 - Semantic relation between classes
 - Aggregation
 - An association shows a class is a part of another class
 - Composition
 - A strong form of aggregation
 - Generalization (Inheritance)
 - A class can inherit one or more classes
 - Realization
 - The class implements the operations and attributes defined by the interface.
 - Dependency
 - shows the dependency between classes

Class diagram - Relationship

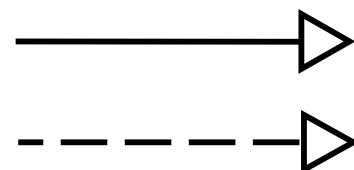
□ Instance level relationship

- Association
- Aggregation
- Composition



□ Class level relationship

- Generalization
- Realization

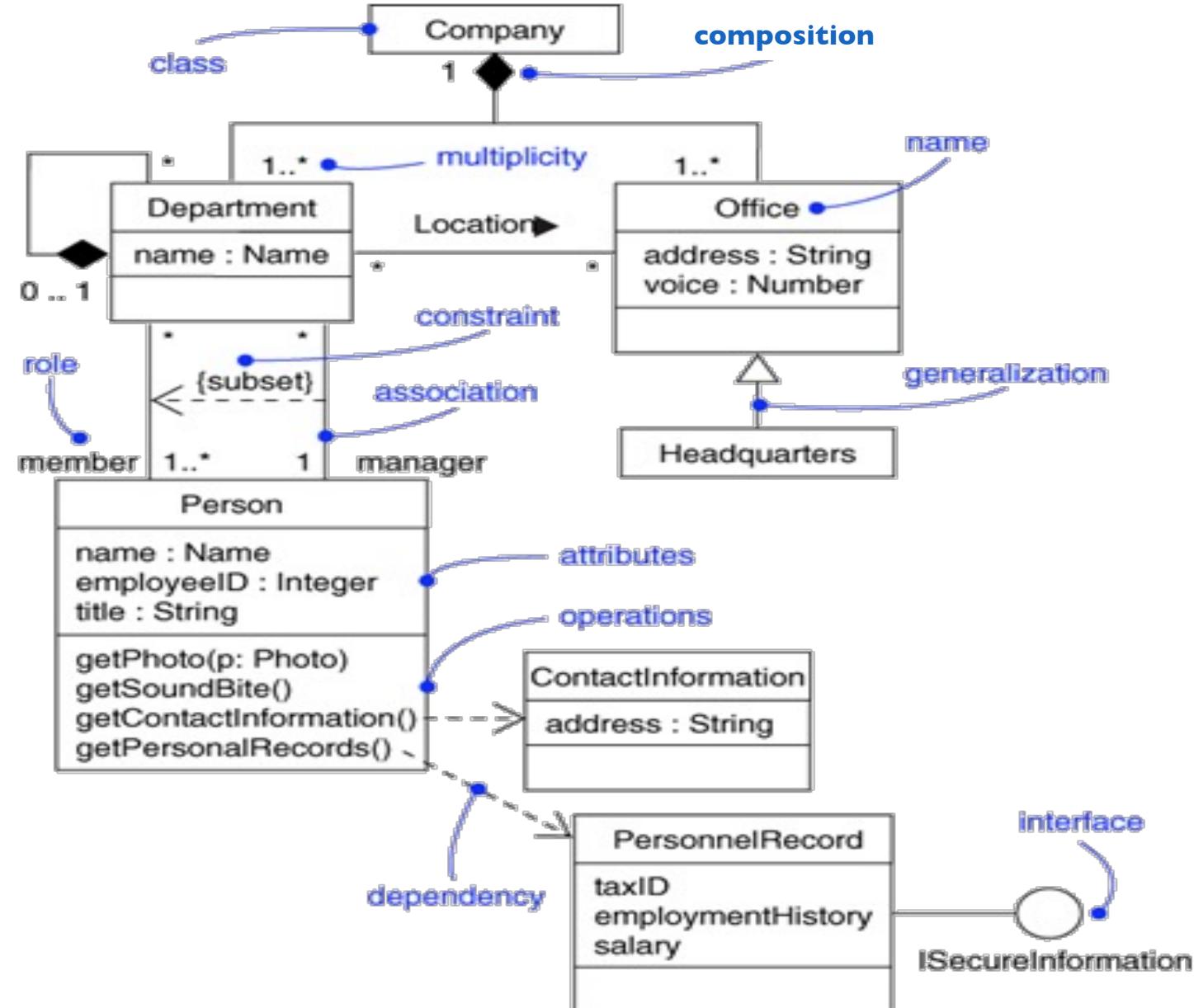


□ General relationship

- Dependency



Class diagram - Relationship



Structural things

□ **Class**

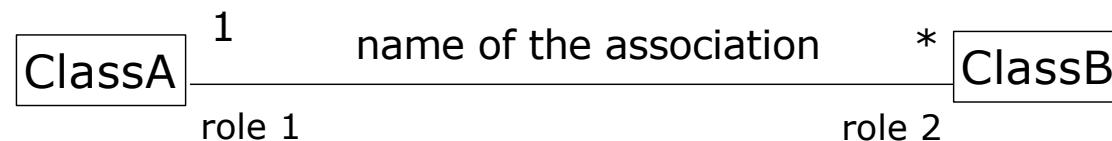
- Represent object, having properties and responsibilities

□ **Interface**

- Used to describe functionalities without implementation
- Just like a template
- Class implement a template → Implement the functionalities

Association

- An association
 - is used to show how two classes are linked together
 - expresses a bidirectional semantic connection between classes
 - is an abstraction of the links between instances of classes
 - Notation

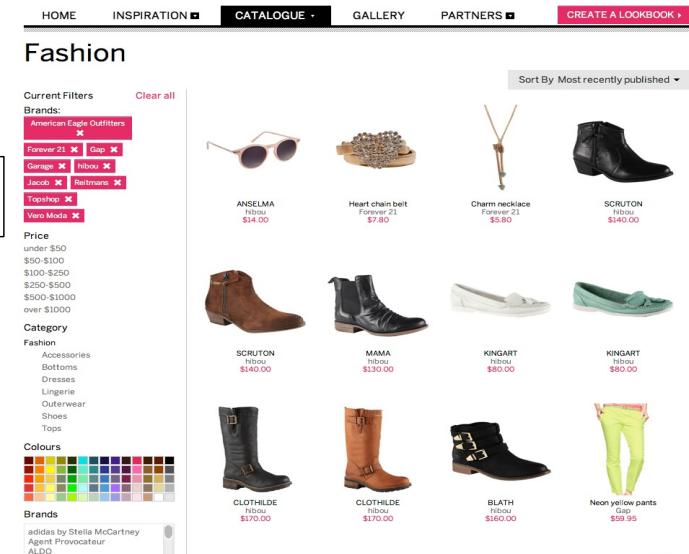


- Each end of an association is called a **role**
 - A role shows the purpose of the association
 - A role can have
 - a name
 - an expression of **multiplicity**
- Example: “An Employee works in a department of a Company”



Association

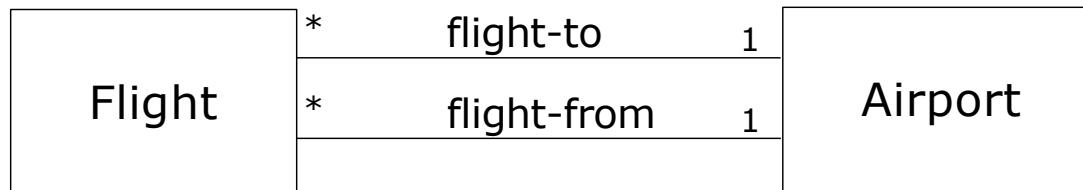
- Multiplicity
 - defines how many instances of a class A are associated with an instance of class B



- Different expressions of multiplicity
 - 1 : one and only one
 - 0..1 : zero or only one
 - m..n : from m to n (integer, $n \geq m \geq 0$)
 - n : exactly n (integer, $n \geq 0$)
 - * : zero or many
 - 1..* : from one to many

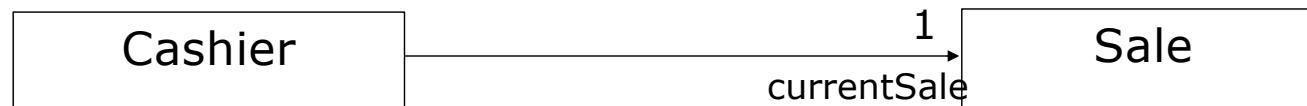
Association

- Multiple associations
 - Two classes can have several associations between them

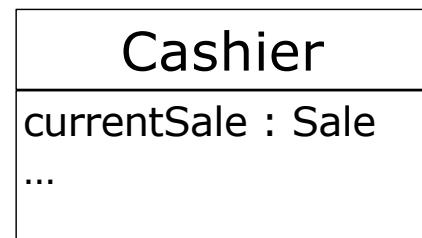


Association

- Directional association and attributes
 - By default, the associations are bi-directional
 - However, associations can be directional
 - Example

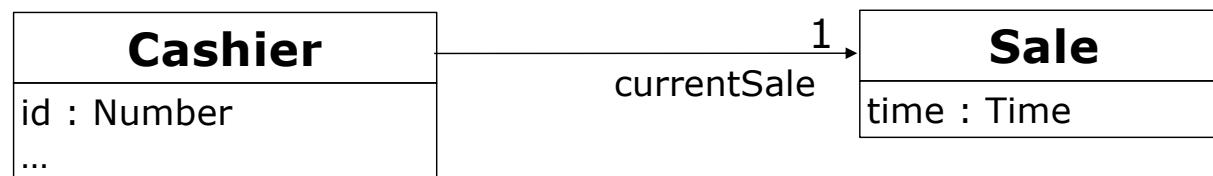


- The navigability pointing from *Cashier* to *Sale* shows that an attribute with *Sale* type
- This attribute is called *currentSale*
- Another form of representation: use of attributes



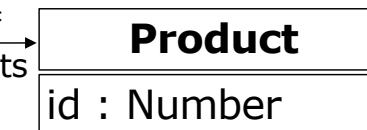
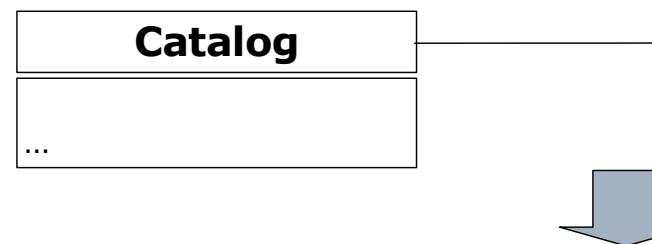
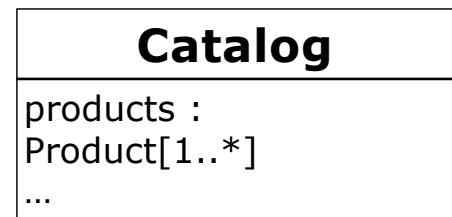
Association

- Directional association and attributes
 - When do we use the directional association or attribute?
 - We use the attribute for “primitive” data types, such as Boolean, Time, Real, Integer, ...
 - We use the directional association for other classes
 - To better see the connections between classes
 - It is just to better represent, these two ways are semantically equivalent
 - Example

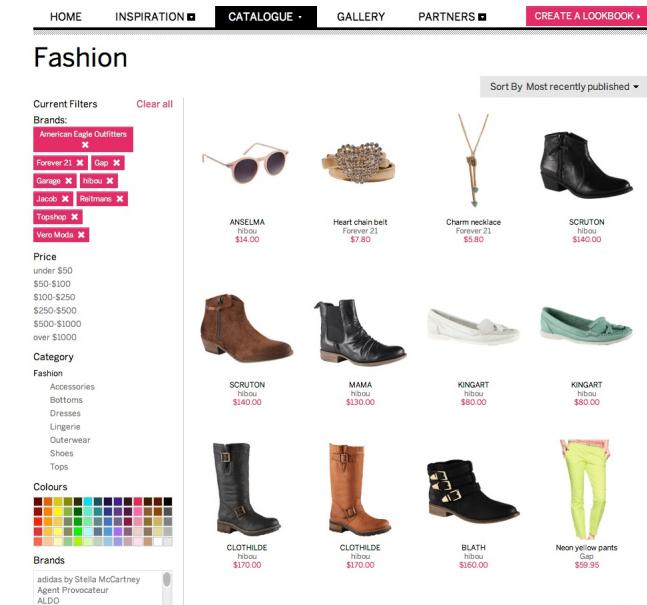


Association

- Directional association and attributes
 - Another example

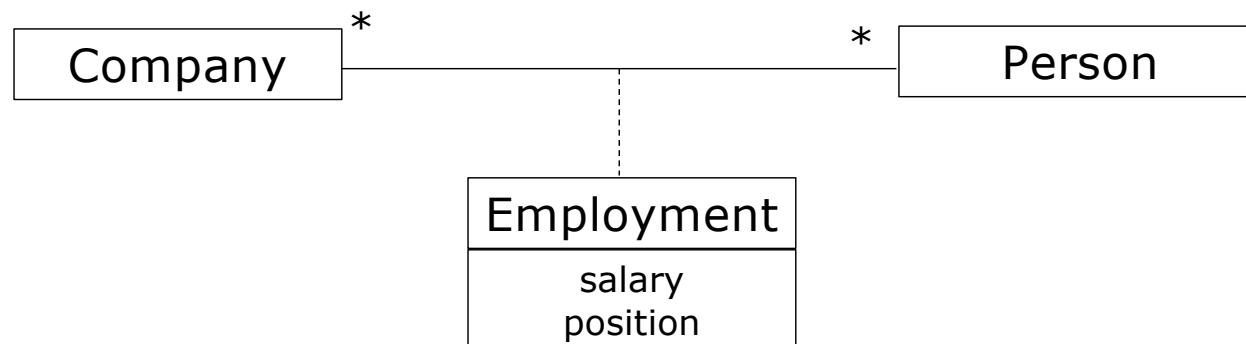


```
public class Catalog {  
    private List<Product> products =  
        new ArrayList<Product>();  
    // ...  
}
```



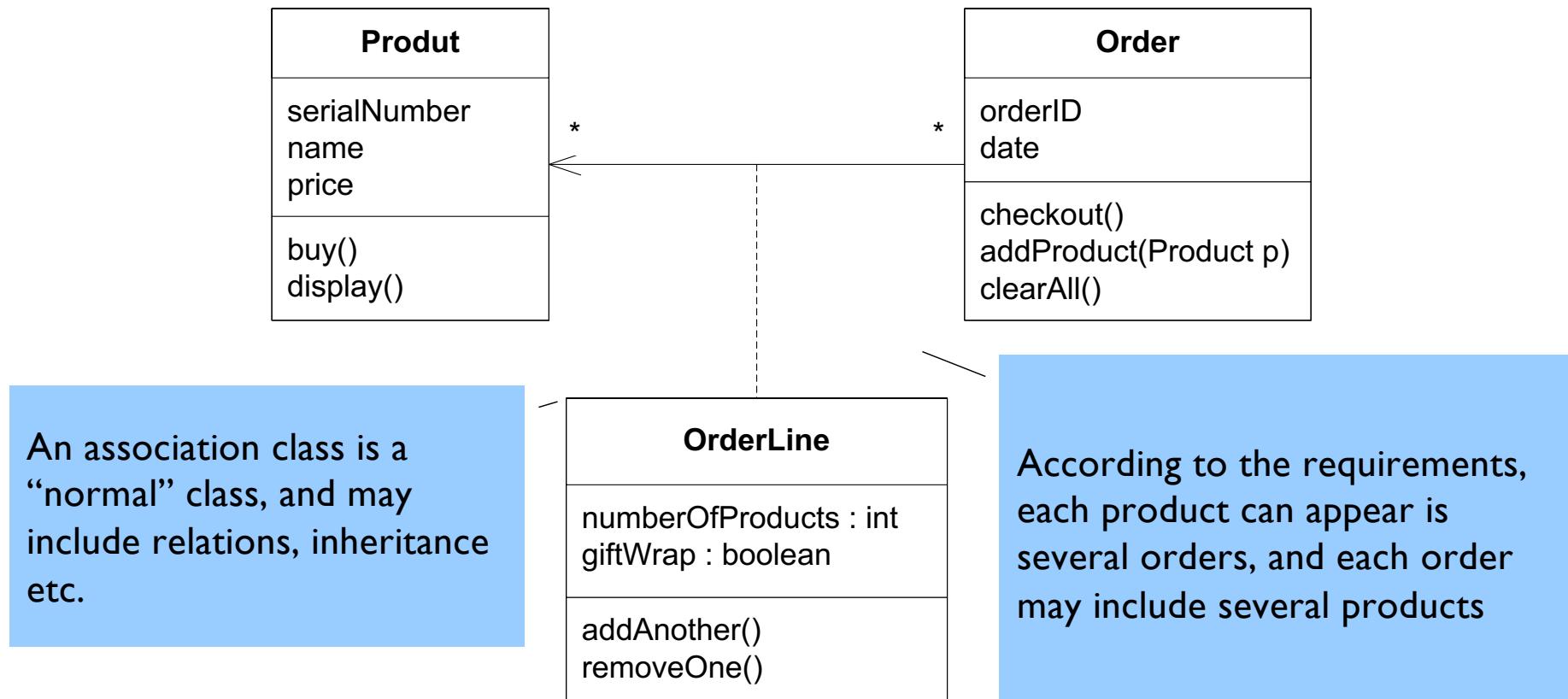
Association

- Association classes
 - An association class allows an association to be considered as a class
 - When an attribute cannot be attached to any of the two classes of an association
 - Example



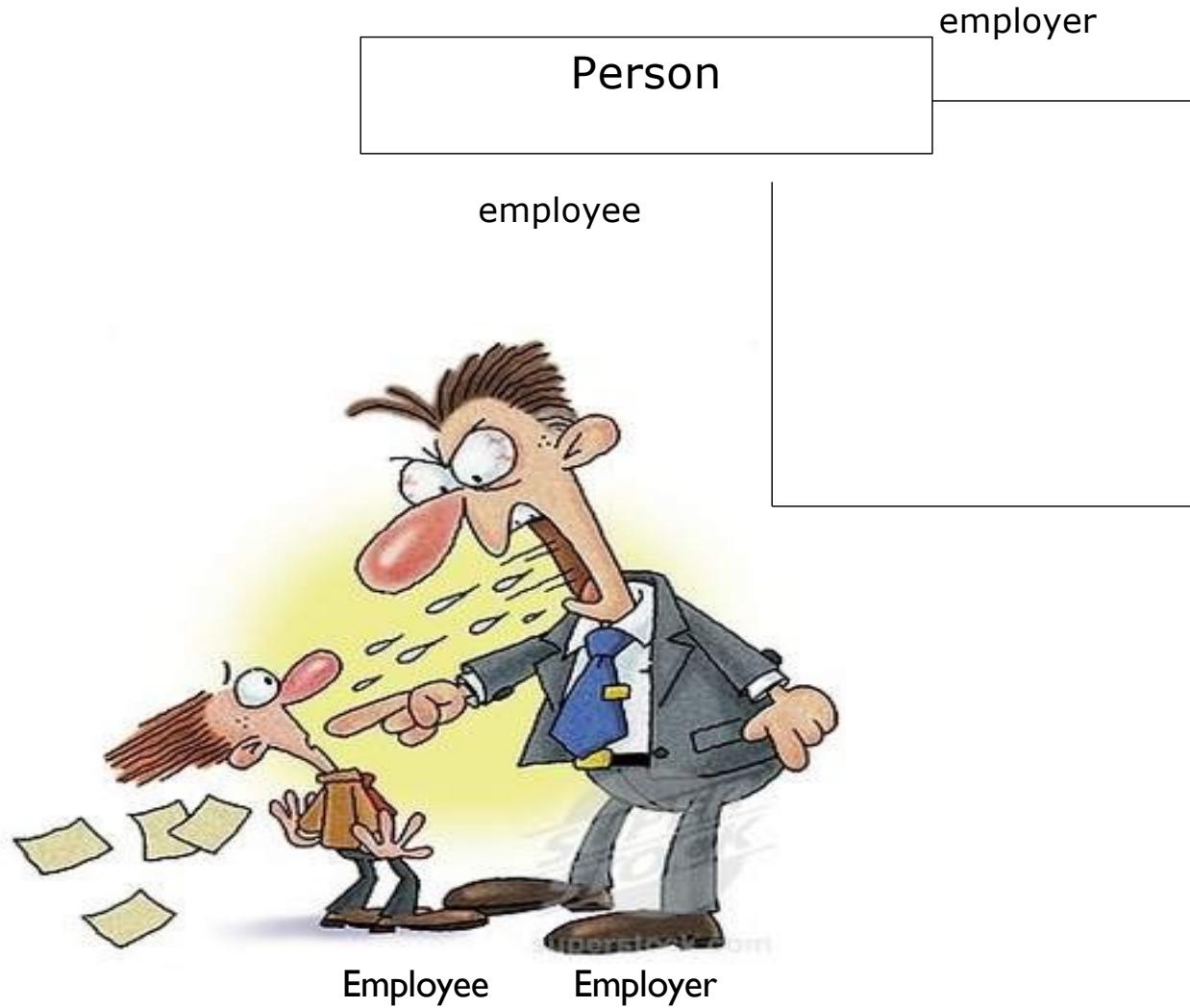
Association Classes

Denoted as a class attached to the association, and specify properties of the association



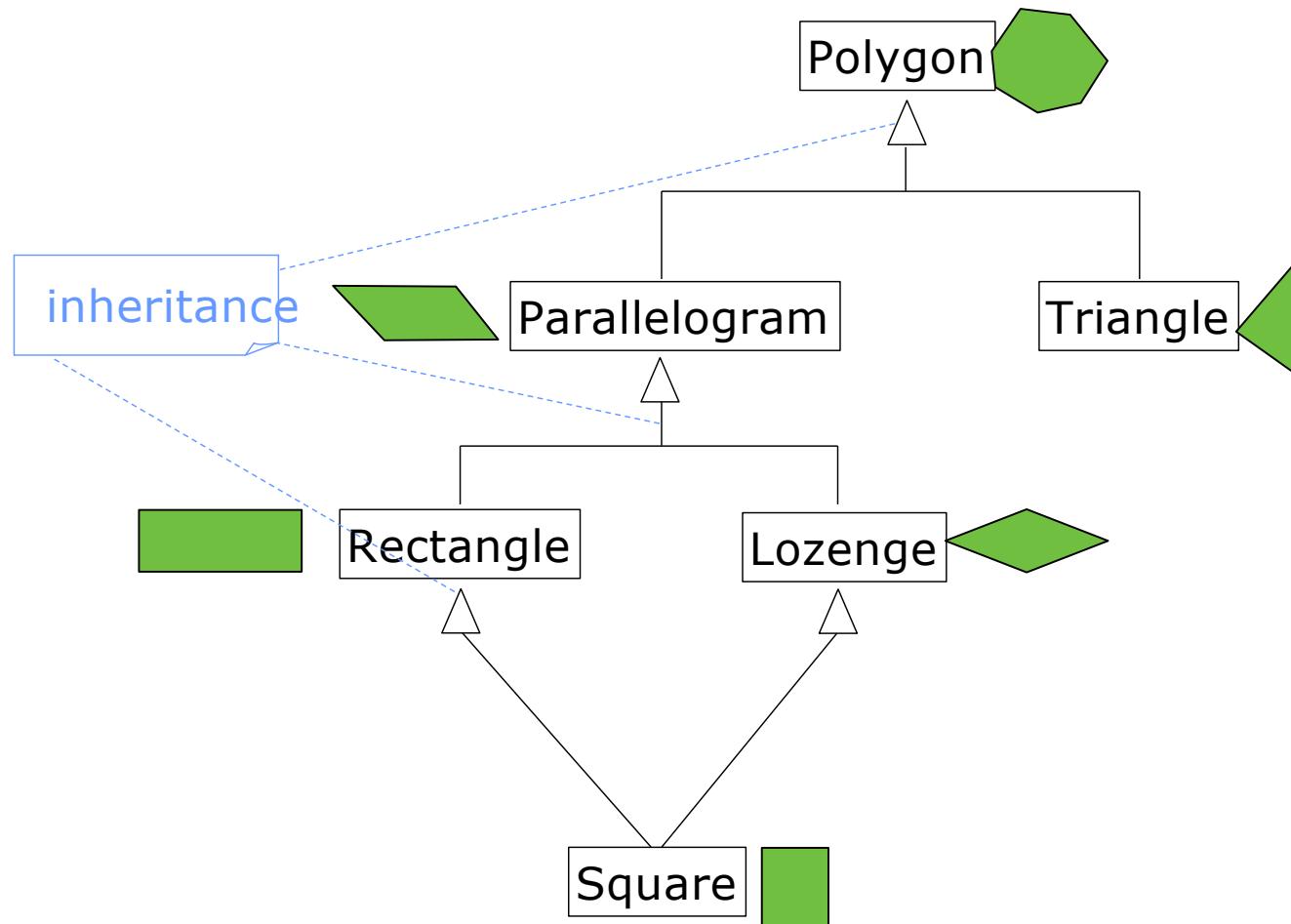
Association

- A class can be associated to itself
 - example



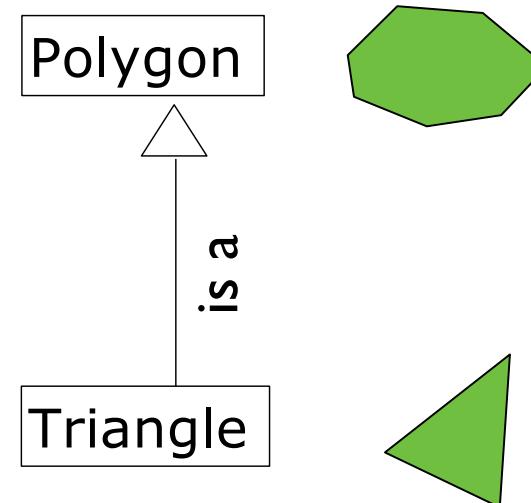
Inheritance

- A class can have several sub-classes



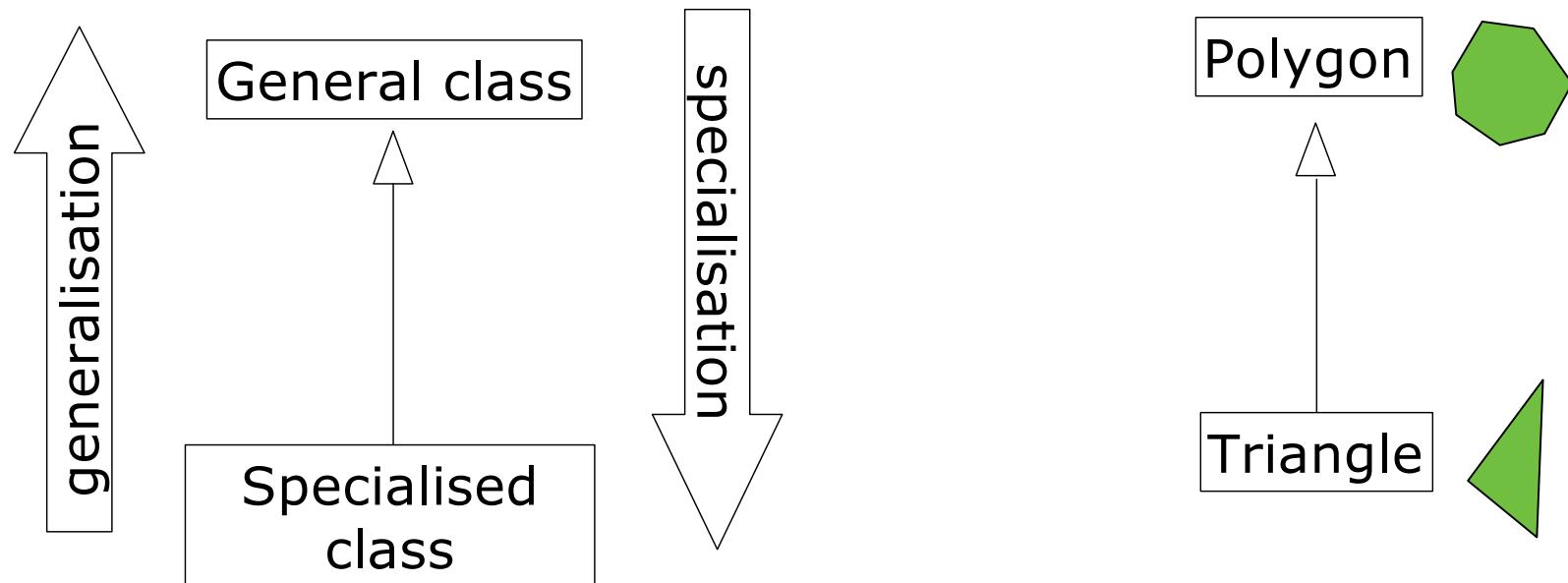
Inheritance

- Substitution principle
 - All subclass objects can play the role of an object of its parent-class
 - An object of a subclass can override an object of its superclass
- Informally
 - A subclass is a kind of superclass
- Example
 - A triangle is a polygon



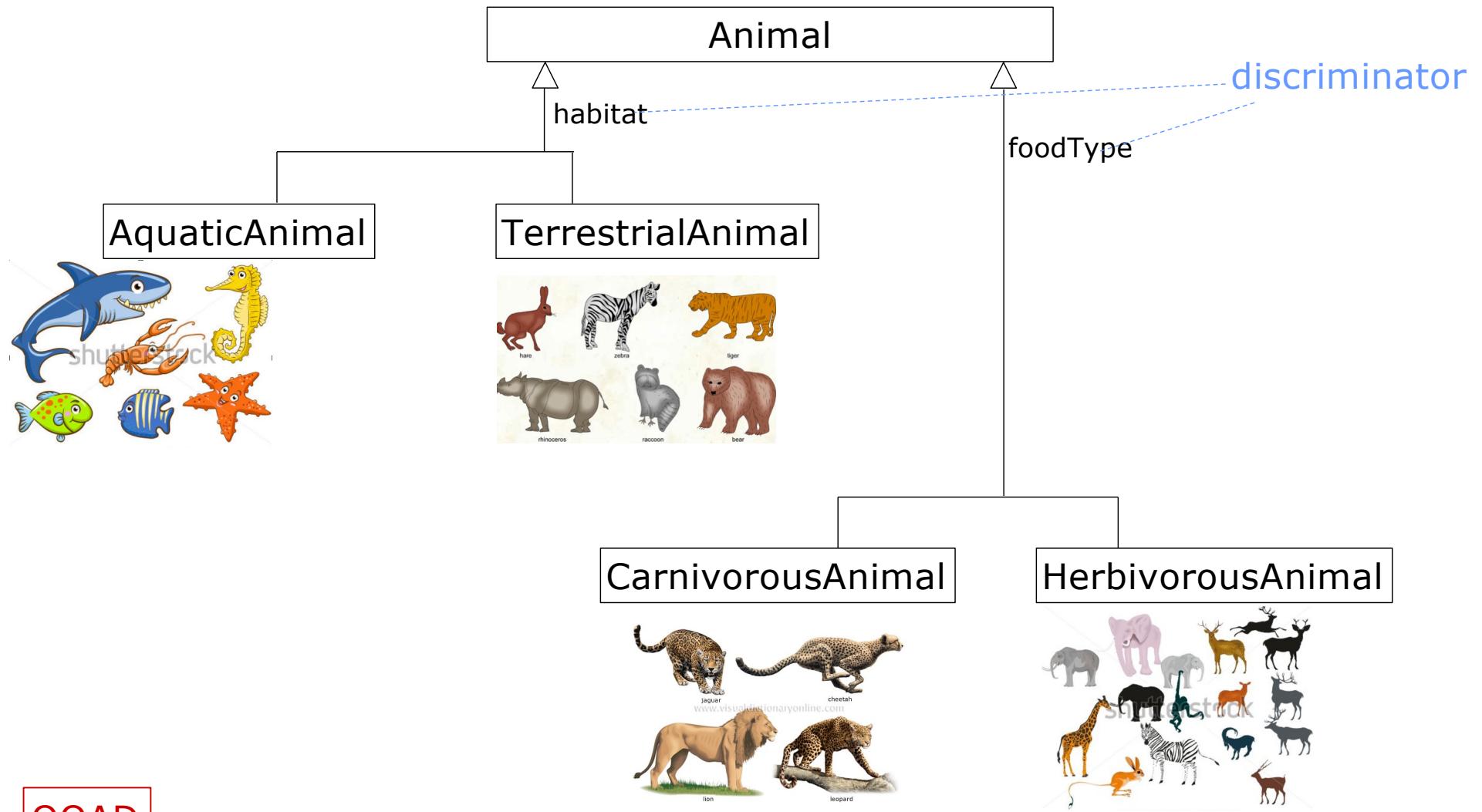
Inheritance

- The subclasses are also called **specialised classes**
- Parent-classes are also called **general classes**
- The inheritance is also called the **specialisation** or **generalisation**



Inheritance

- The (optional) **discriminator** is a label describing the criterion that the specialisation bases on

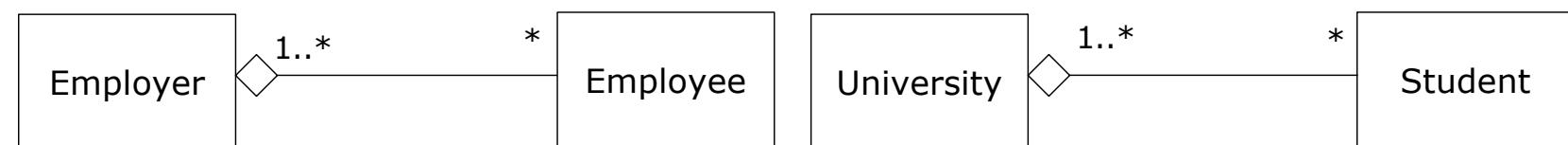
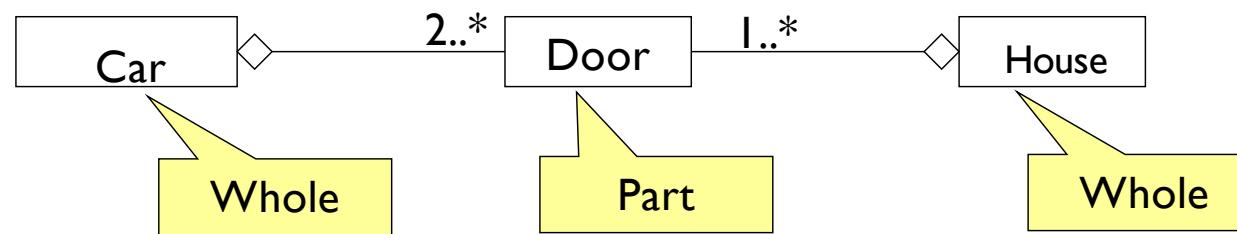


Aggregation

- An aggregation is a form of association that expresses a stronger (than normal association) coupling between class
- An aggregation is used between two classes
 - master and slave: “belongs to”
 - whole and part: “is a part of”
- Notation
 - The symbol denoting the place of aggregation of the aggregate side



- Examples

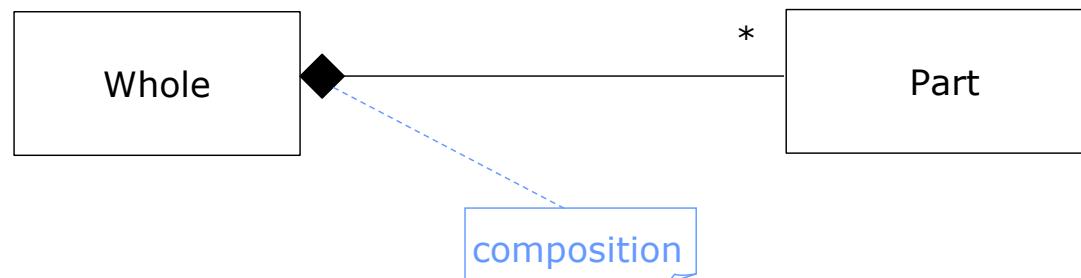


Aggregation (cont'd)

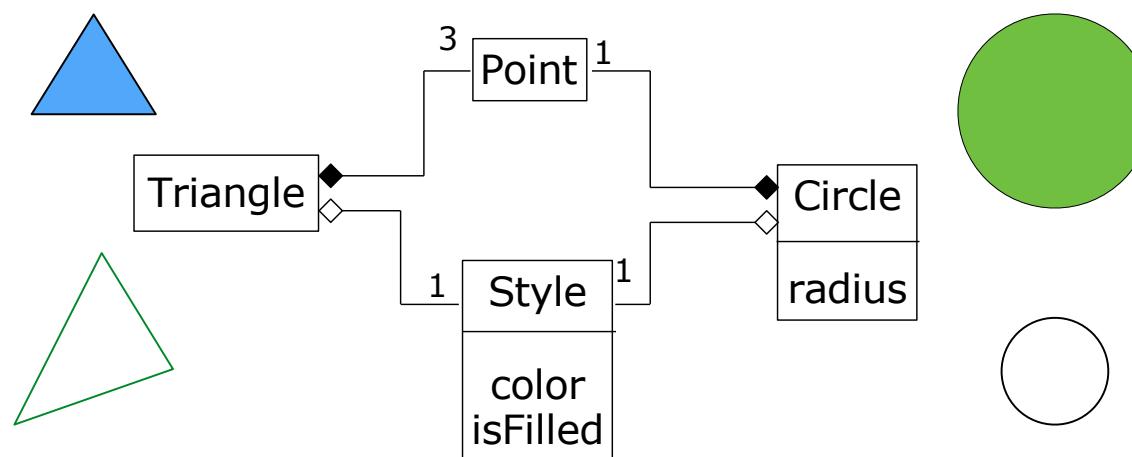
- Aggregation tests:
 - Is the phrase “part of” used to describe the relationship?
 - A door is “part of” a car
 - Are some operations on the whole automatically applied to its parts?
 - Move the car, move the door.
 - Are some attribute values propagated from the whole to all or some of its parts?
 - The car is blue, therefore the door is blue.
 - Is there an intrinsic asymmetry to the relationship where one class is subordinate to the other?
 - A door is part of a car. A car is not part of a door.

Composition

- A composition is a strong form of aggregation
- A composition is also a “whole-part” relationship but the aggregate is stronger
 - If the whole is destroyed then parts will be also destroyed

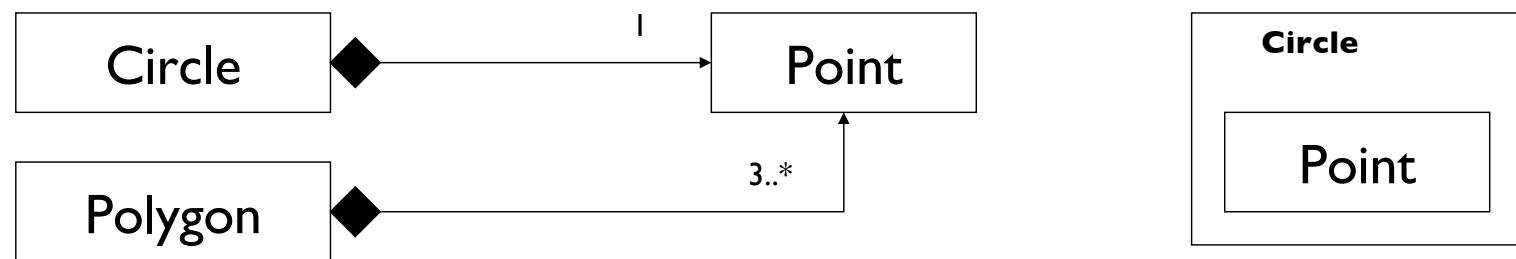


- Example



Composition

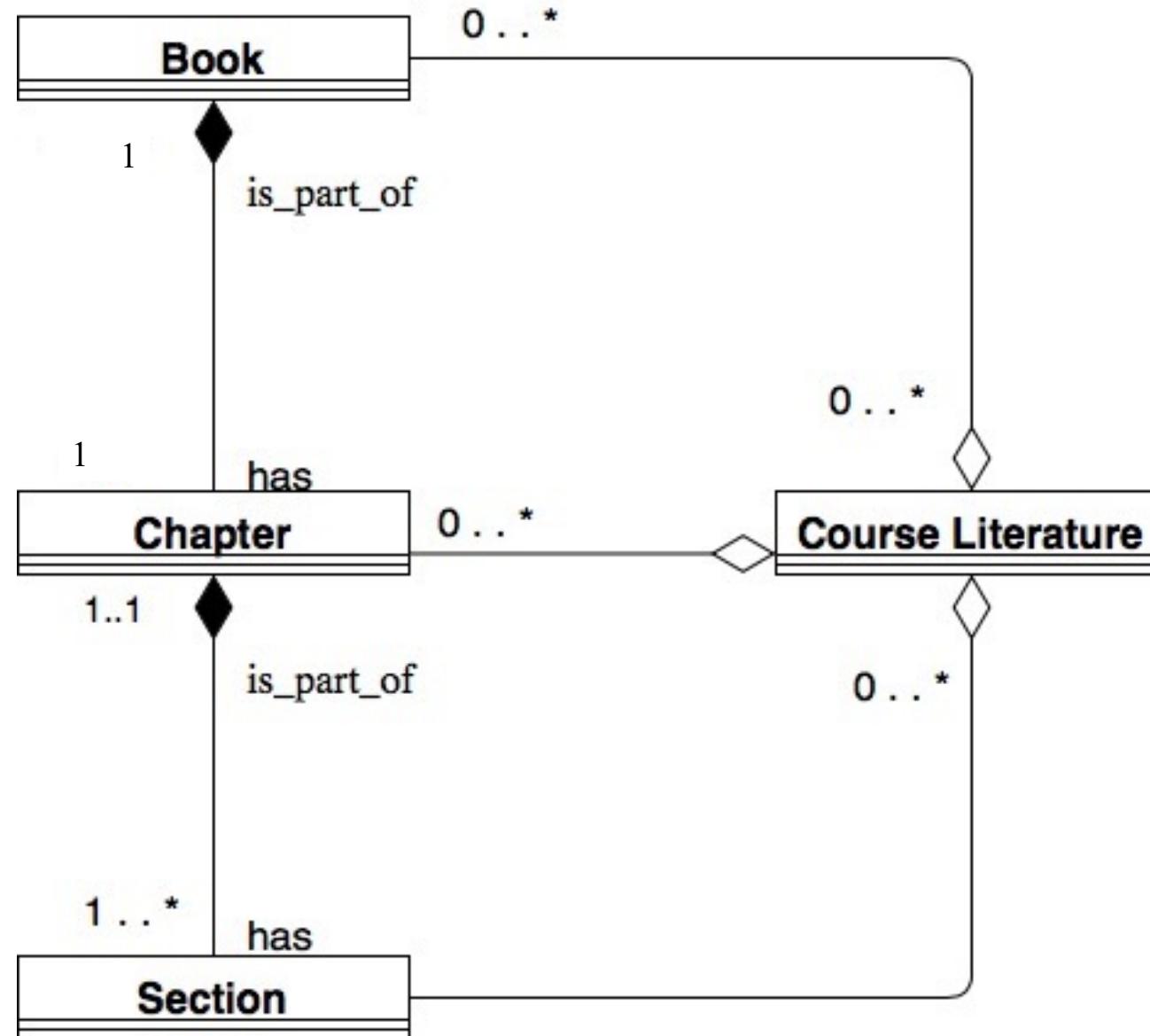
- A strong form of aggregation
 - The whole is the sole owner of its part.
 - The part object may belong to only one whole
 - Multiplicity on the whole side must be zero or one.
 - The life time of the part is dependent upon the whole.
 - The composite must manage the creation and destruction of its parts.



Composition vs. Aggregation

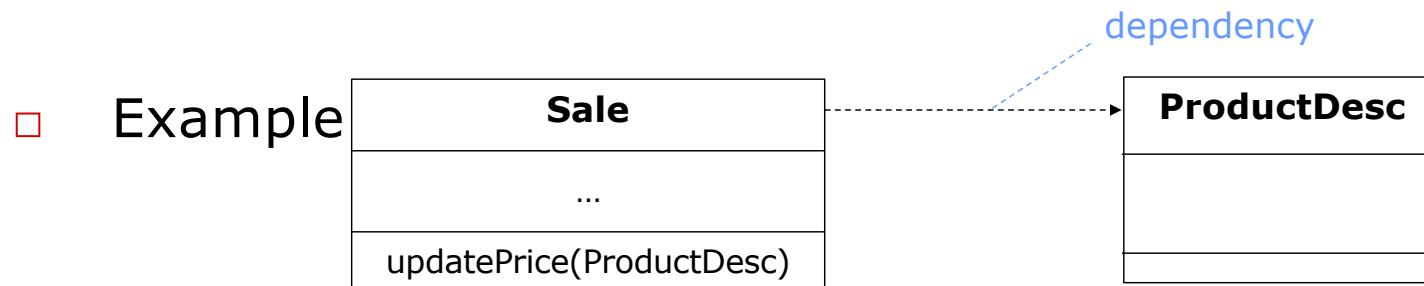
Aggregation	Composition
<p>Part can be shared by several wholes</p> <p></p>	<p>Part is always a part of a single whole</p> <p></p>
<p>Parts can live independently (i.e., whole cardinality can be 0..*)</p>	<p>Parts exist only as part of the whole. When the wall is destroyed, they are destroyed</p>
<p>Whole is not solely responsible for the object</p>	<p>Whole is responsible and should create/destroy the objects</p>

Aggregation vs Composition - Example



Dependency

- A class may depend on another class
- The dependency between classes can be implemented in different ways
 - Having an attribute with the type of another class
 - Sending a message using an attribute, a local variable, a global variable of another class or static methods
 - Receiving a parameter having type of another class



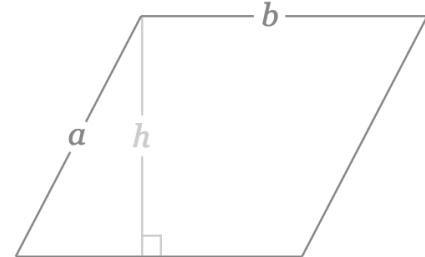
CHANEL
N°5
Parfum Bottle 30ml
4174518

£230.00
30 ML | £766.67 per 100ML

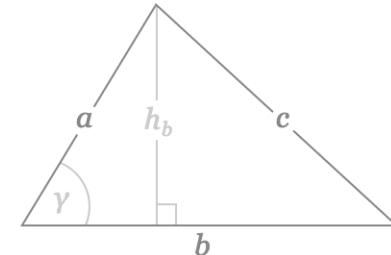
Online only - Save 10 percent when you spend £40 on selected fragrance & luxury beauty

Abstract class

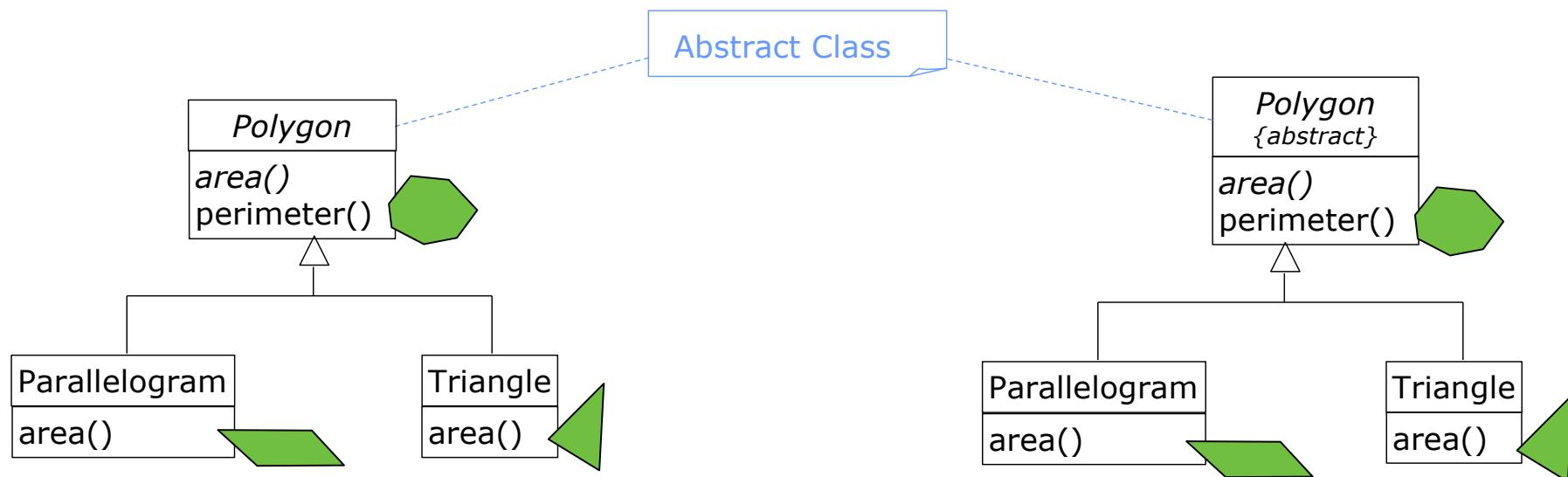
- An abstract class is a class that has no instances
 - inheritance: *area()*, *perimeter()*
 - polymorphism: *area()*
 - Parallelogram = $b * h$



$$\text{Triangle} = (h * b) / 2$$

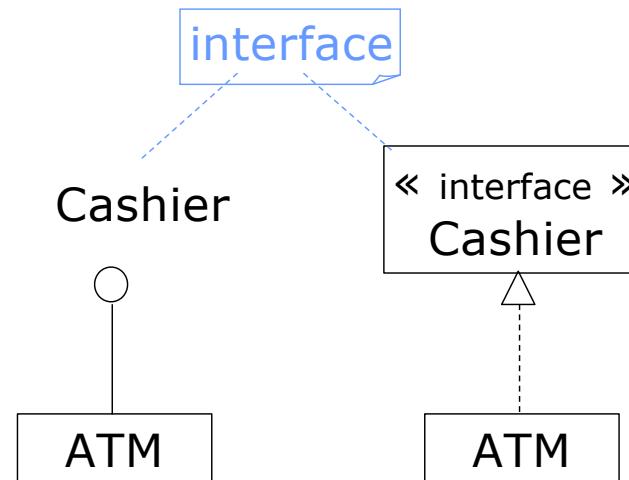


- Notation



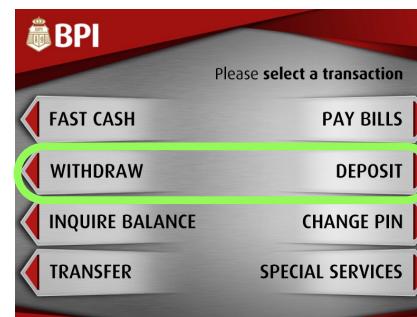
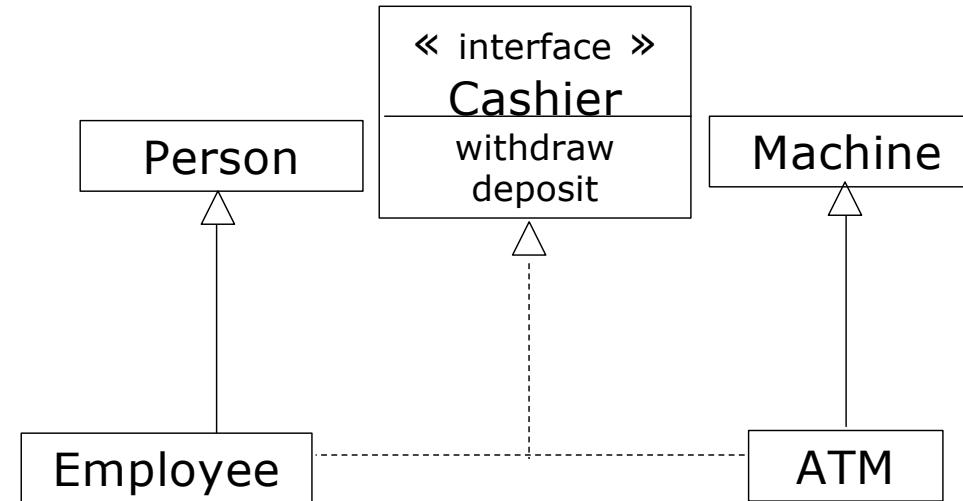
Interface

- An interface
 - describes a portion of the visible behaviour of a set of objects
 - is very similar to an abstract class that contains only abstract operations
 - **specifies only the operations without implementation**
- Two notations



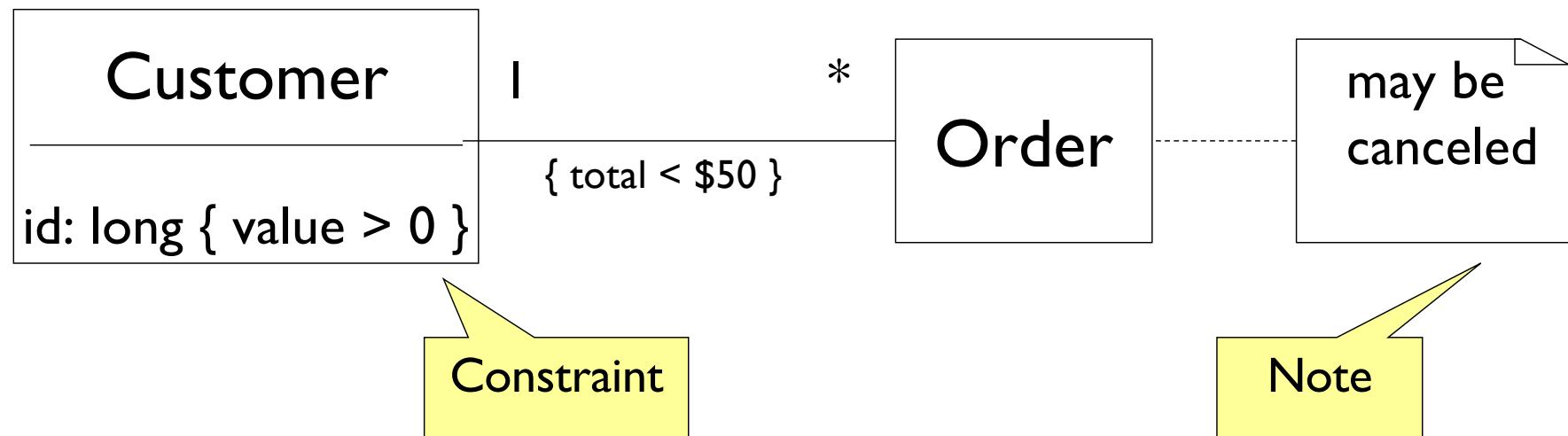
Interface

□ Example

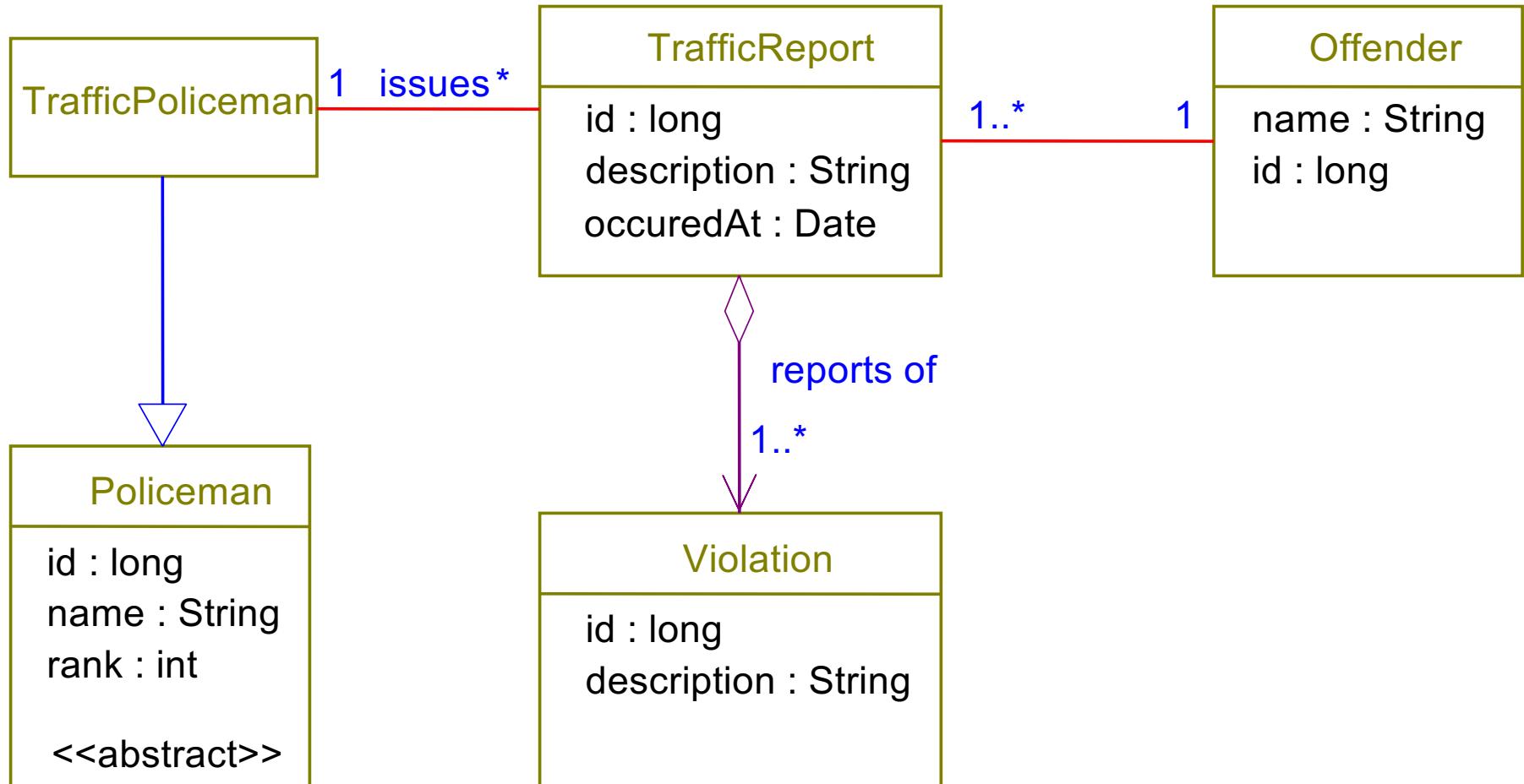


Constraint Rules and Notes

- Constraints and notes annotate among other things associations, attributes, operations and classes.
- Constraints are semantic restrictions noted as Boolean expressions.

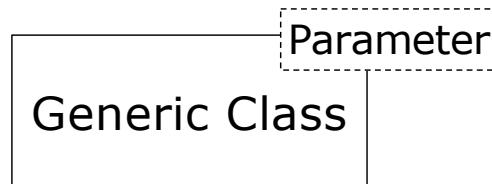


TVRS Example

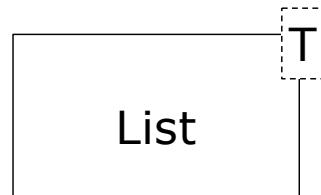


Generic class

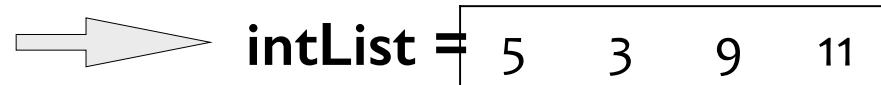
- A generic class (or parameterised) allows to consider the types of data as parameters
- Generic classes are often used for the types of collection classes: vector, table, stack, ...
- Notation



- Example

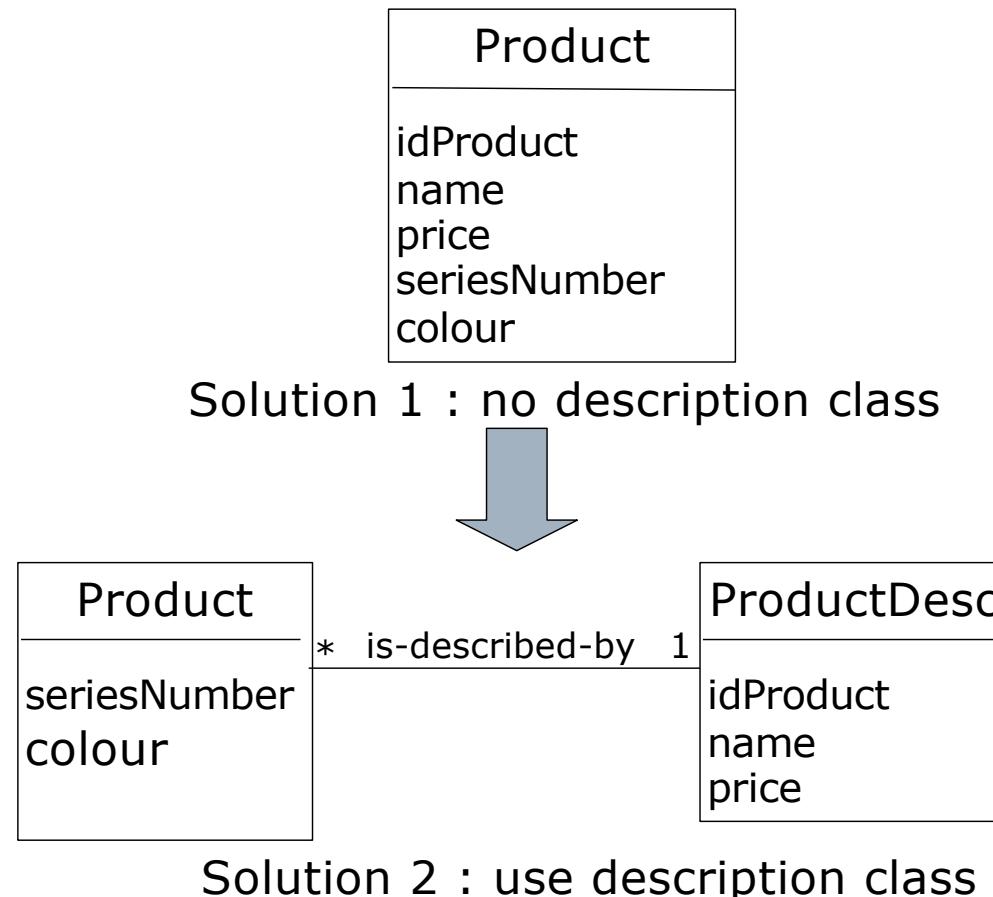


- “template” in C++
- Generic type in Java
 - `List<Integer> intList = new ArrayList<Integer>();`



Description class

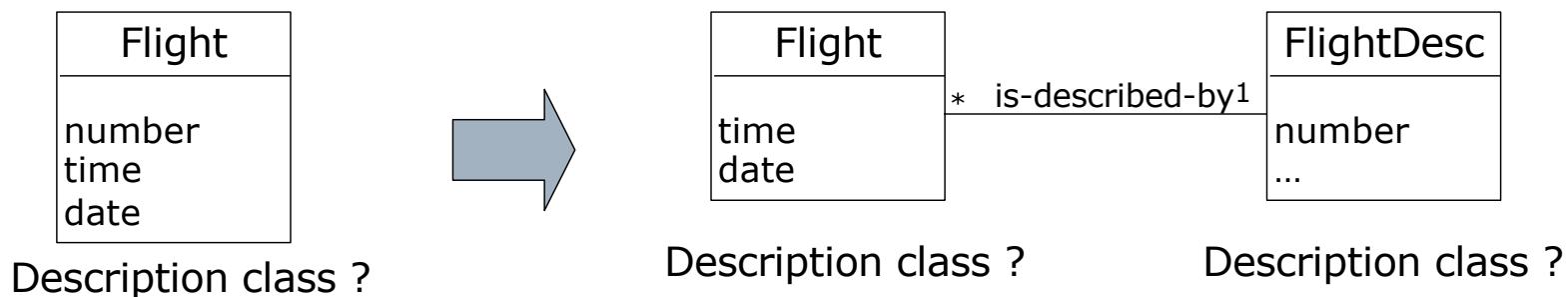
- Description class contains information describing the objects
 - increase cohesion, reusability
- Example
 - Description of a product with certain information



Description class

- When do we use description class?
 - Reduce redundant information
 - Avoid repetition of information
 - Describe some objects independent of real-world objects
 - Keep the information in the object even if real-world objets are removed

- Example
 - Describe the information (number, time, date, ...) of a flight



Building class diagrams

- Class diagrams are built at different stages and at different levels
 - Domain model
 - Model a set of **conceptual classes (concepts)** and the relationships between them
 - System model
 - Model all classes and their relationships, including the architecture classes and user interface classes

Building class diagrams

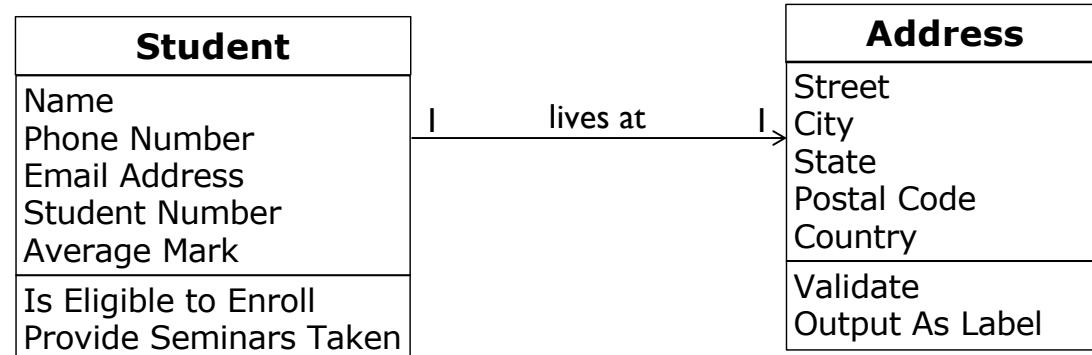
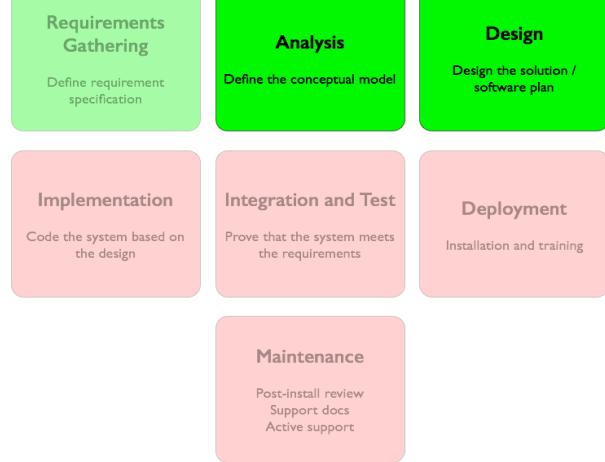
- Suggested steps to build class diagrams
 - Build the domain model
 - Identify conceptual classes
 - Identify the relationships
 - Identifier the attributes
 - Identify the inheritances and the interfaces
 - Determine the main responsibilities of each conceptual class
 - Build the system model
 - Introduce new classes
 - Detail the attributes
 - Detail the relationships
 - Decide the operations of each class
 - During the development, refine progressively the class diagrams until satisfaction
 - Add or remove classes, attributes, operations, relationships

Building class diagrams

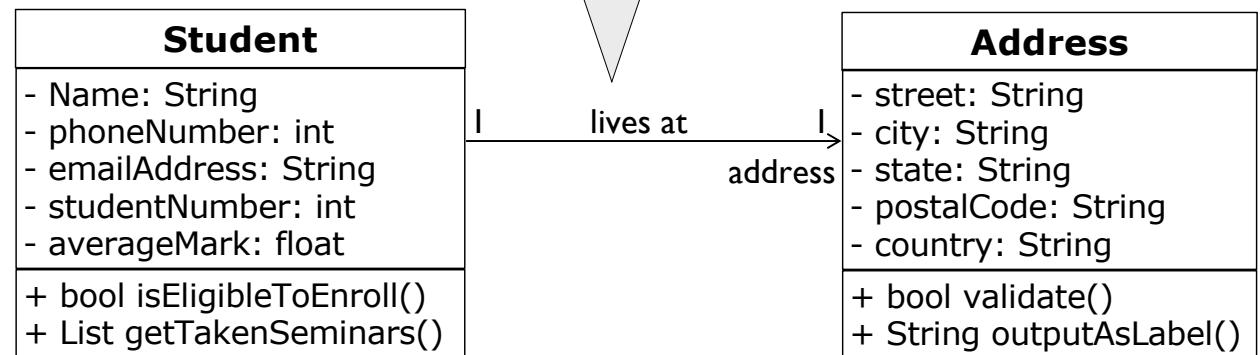
- Building domain model
 - The **domain model** is the important model in object-oriented analysis
 - This model is also called the **analysis class diagrams**

Building class diagrams

- Class diagrams are progressively built at different phases of software development process



Analysis class diagram



Design class diagram (for Java implementation)

Building class diagrams

- Identifying classes
 - The question “How to find classes?”
 - The concepts in the studied domain can be also classes
 - These concepts are called **conceptual classes**
 - So, we **firstly** identify the conceptual classes, and **then other** classes are added during the development
- The principles for finding conceptual classes
 - Use of a **list of categories**
 - Identification of **nouns**

Building class diagrams

- Identifying classes
 - Use of a list of categories

Categories of conceptual classes	Examples
transaction (of business)	Reservation, Payment
product or service relating to the transaction	Product, Flight
where transactions are recorded?	Cash desk, Cash
actors of use-cases	Cashier, Customer
location (of service, of transaction)	Station, Store
important events	purchase
physical objects	Car
description of things	Description of products
catalog	Product catalog
containing things	Store
other collaboration systems	Bank, database
organisations	University
policy, principle	Tax
...	

Building class diagrams

- Identifying classes
 - Identification of **nouns**
 - Review written documents such as specification or description of use-cases
 - Extract names and consider them as conceptual class candidates
 - Remove the nouns which
 - are redundant
 - are vague or too general
 - aren't conceptual classes by experience and knowledge in the context of the application

Building class diagrams

- Identifying classes
 - Identification of **nouns** from use-case spec
 - Example

Actions of actor

- The **customer** comes to the **cash desk** with the **products** to buy
- The **cashier** encodes the **identifier** of each **product**
If a **product** has more than one **item**, the **cashier** inputs the number of **items**
- After having encoded all of the **products**, the **cashier** signals the end of the **purchase**
- The **cashier** announces the total amount to the customer
- The **customer** pays
- The **cashier** input the amount of **money** paid by the customer

Actions of system

- The **cash desk** displays the description and price of the product
This number is displayed
- The **cash desk** calculates and displays the total amount that the customer has to pay
- The **cash desk** displays the balance



Building class diagrams

- Identifying classes
 - Identification of **nouns**
 - Example (continue)



Actions of actor

- The cashier receives the cash payment
- The **cashier** gives **change** to the customer and the receipt
- The **customer** leaves the **cash desk** with the bought **products**

Actions of system

- The **cash desk** prints the receipt
- The **cash desk** saves the **purchase**

Building class diagram

- Candidate classes from nouns identified from use-case description
 - customer, cash desk, product, item, cashier, purchase, change

Building class diagrams

- Identifying the relationships and attributes
 - Starting with central classes of the system
 - Determining the attributes of each class and associations with other classes
 - Avoiding adding too many attributes or associations to a class
 - To better manage a class

Building class diagrams

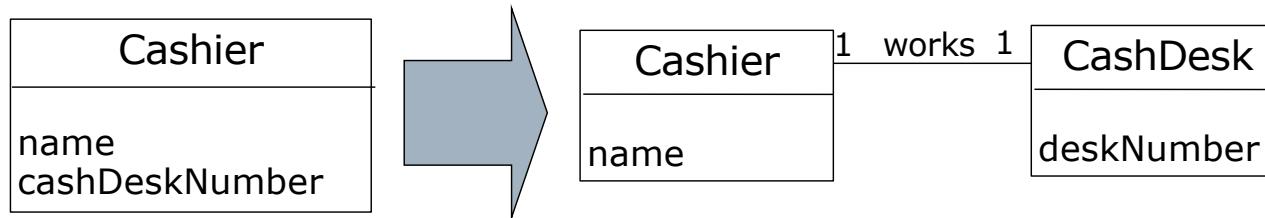
- Identify the relationships
 - A association should exist between class A and class B, if
 - A is a service or product of B
 - A is a part of B
 - A is a description for B
 - A is a member of B
 - A is connected to B
 - A possesses B
 - A controls B
 - ...
 - Specify the multiplicity at each end of the association
 - Label associations

Building class diagrams

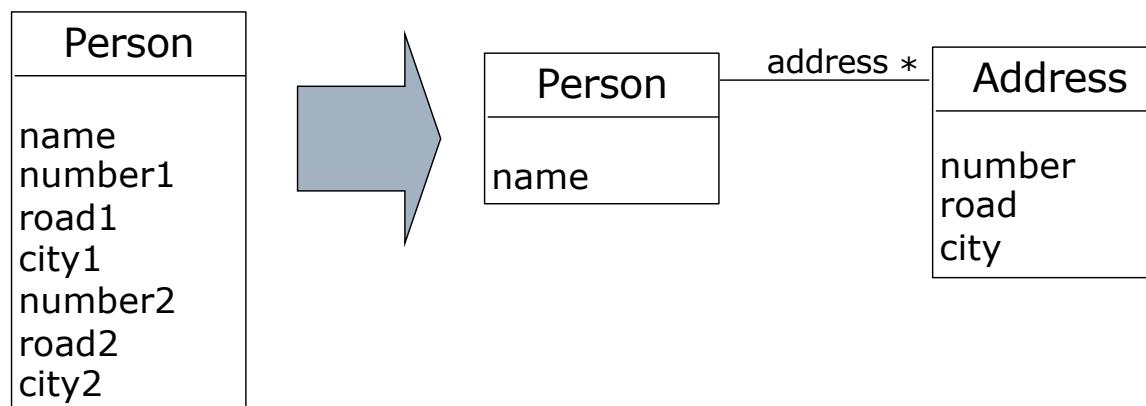
- Identifying attributes
 - For each class, determine the information needed to store according to the requirement specification or use-case
 - Example: Cashier needs an identifier, a name, ...
 - Principle to determine attributes
 - An attribute represents only data related to the class that owns the attribute
 - If a subset of the attributes form a coherent group, it is possible that a new class is introduced
 - Determine only the names of attributes at this stage (i.e., analysis phase)

Building class diagrams

- Identifying attributes
 - Example
 - An attribute represents only data related to the class that owns the attribute

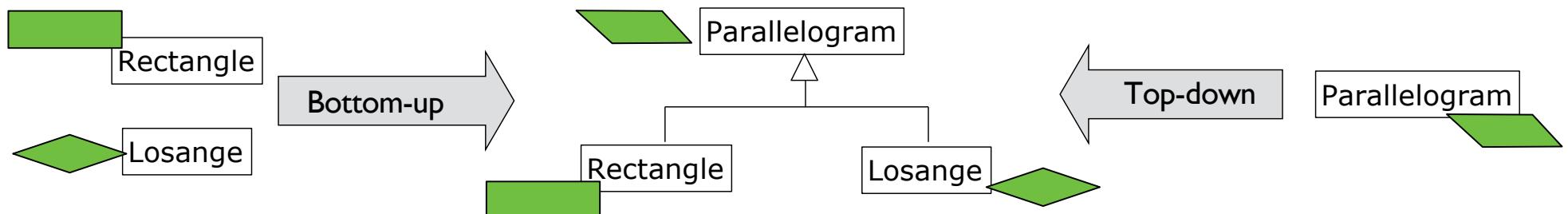


- If a subset of the attributes form a coherent group, it is possible that a new class is introduced



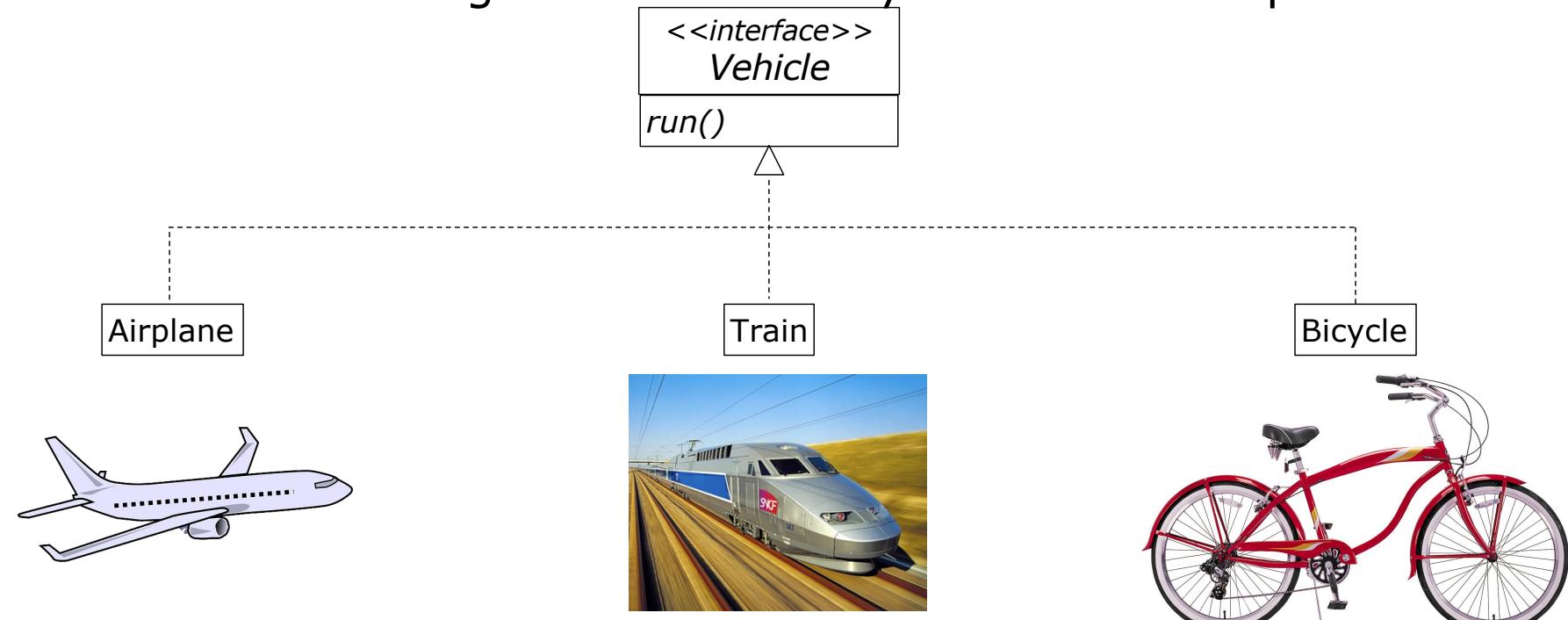
Building class diagrams

- Identifying inheritances
 - Two approaches
 - Bottom-up
 - Generalisation: group similar classes to create super-classes
 - Top-down
 - Specialisation: build sub-classes from existing general classes



Building class diagrams

- Identifying interfaces
 - Create interfaces rather than super-class, if
 - It is necessary to realise different implementations of the same class
 - Two classes to generate share the operations that are not similar
 - The class to generalise already has its own super-class



Building class diagram

- Determining the responsibilities of classes
 - A responsibility is one or several tasks that the system has to perform
 - Each functional requirements must be attributed to one of the classes
 - All the responsibilities of a class must be attributed to one of the classes
 - If a class has too many responsibilities, it must be divided into several classes
 - If a class has no responsibility, it should be probably be useless
 - If responsibility can not be assigned to any class, a new class can be introduced
 - The responsibilities can be determined by analysing the actions/verbs in the use-case specification.

Building class diagrams

- Developing design class diagrams
 - Basing on analysis class diagrams (domain models)
 - Detailing analysis class diagrams
 - Introducing new classes, if necessary
 - For example, an association of class becomes a new class
 - Detailing attributes
 - Adding and detail relationships
 - Determining operations

Building class diagrams

- Detailing attributes
 - Determining the types of attributes
 - Using primitive types: boolean, int, real, ...
 - Defining new type for an attribute (new class), if
 - It consists of several sections
 - It has other attributes
 - It is associated with other operations
 - Determining initial values if necessary
 - Determining the visibility of attributes

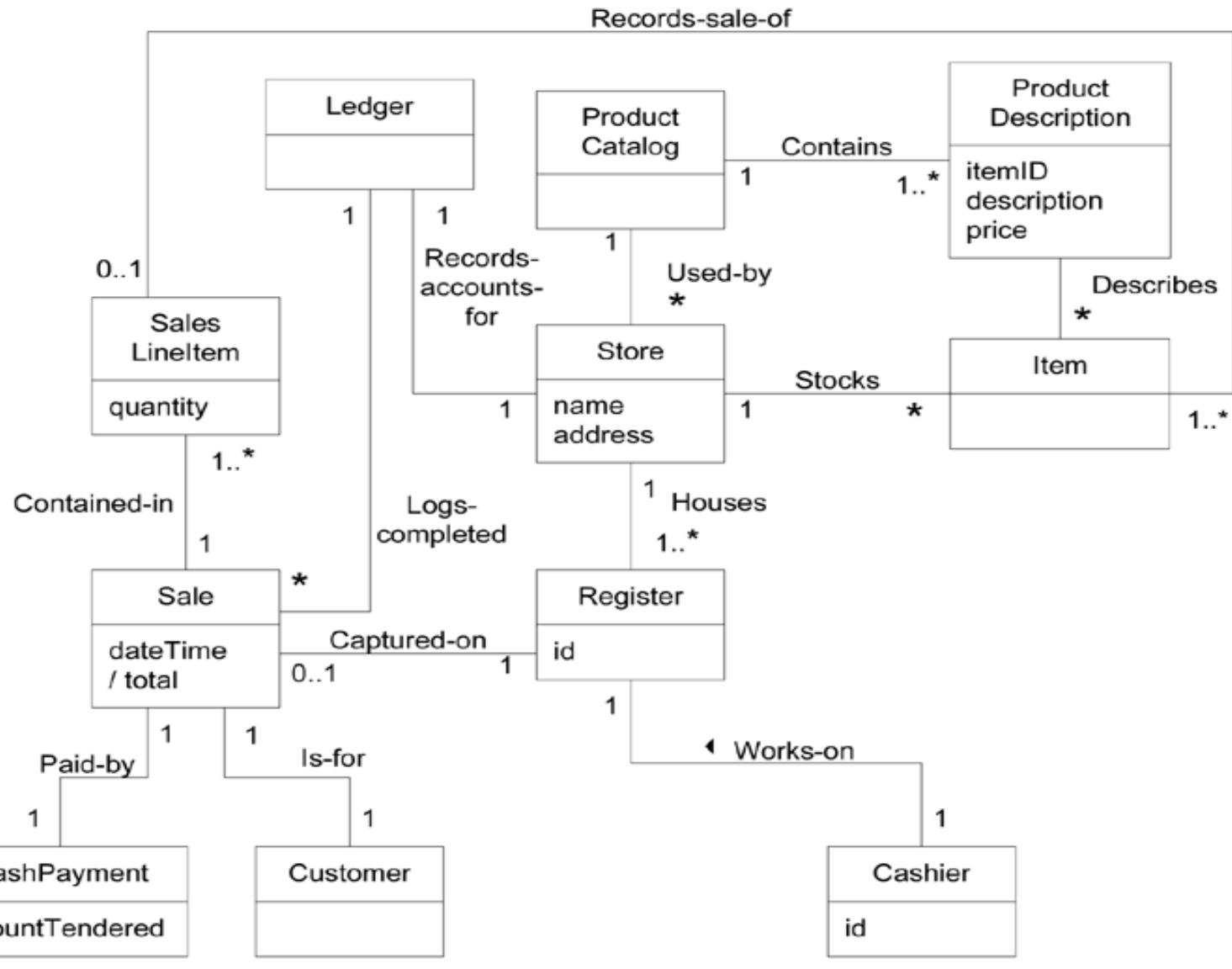
- Detailing relationships
 - Introducing relationships according to newly added classes
 - Specifying if an association is an aggregate or composition
 - Naming the relationship
 - Giving the direction

Building class diagrams

- Determining the operations of each class
 - getters and setters
 - Operations are used to achieve the identified responsibilities
 - A responsibility can be carried out by several operations
 - Determining the visibility of operations
 - Essential operations carrying out responsibilities are declared “public”
 - Operations serving only in the class are declared “private” or “protected” if the class should be inherited

Building class diagrams

- Example: supermarket cash register system



Building class diagrams

- Example: Sentinel HASP Licensing Domain
 - When software vendor purchases a Sentinel HASP License Development Kit (LDK), the vendor is also provided with a unique **batch code** and corresponding **vendor key** that contains unique vendor code specific to the company. The code is used by Sentinel LDK to communicate with protection keys, and to differentiate vendor keys from those of other software vendors.
 - A **batch code** consists of five characters that represent company's unique vendor code. When Sentinel protection keys are ordered from SafeNet, batch code is written to the keys before sending to the vendor. The batch code is also written on the outside of each key.
 - A **feature** is some identifiable functionality of a software application. Features may be used to identify entire executables, software modules, .NET or Java methods, or a specific functionality such as print or save. Each feature is assigned a unique identifier and is associated with a specific batch code. Feature id and feature name both should be unique in the associated batch. The maximum length for a feature name is 50 characters.

Building class diagrams

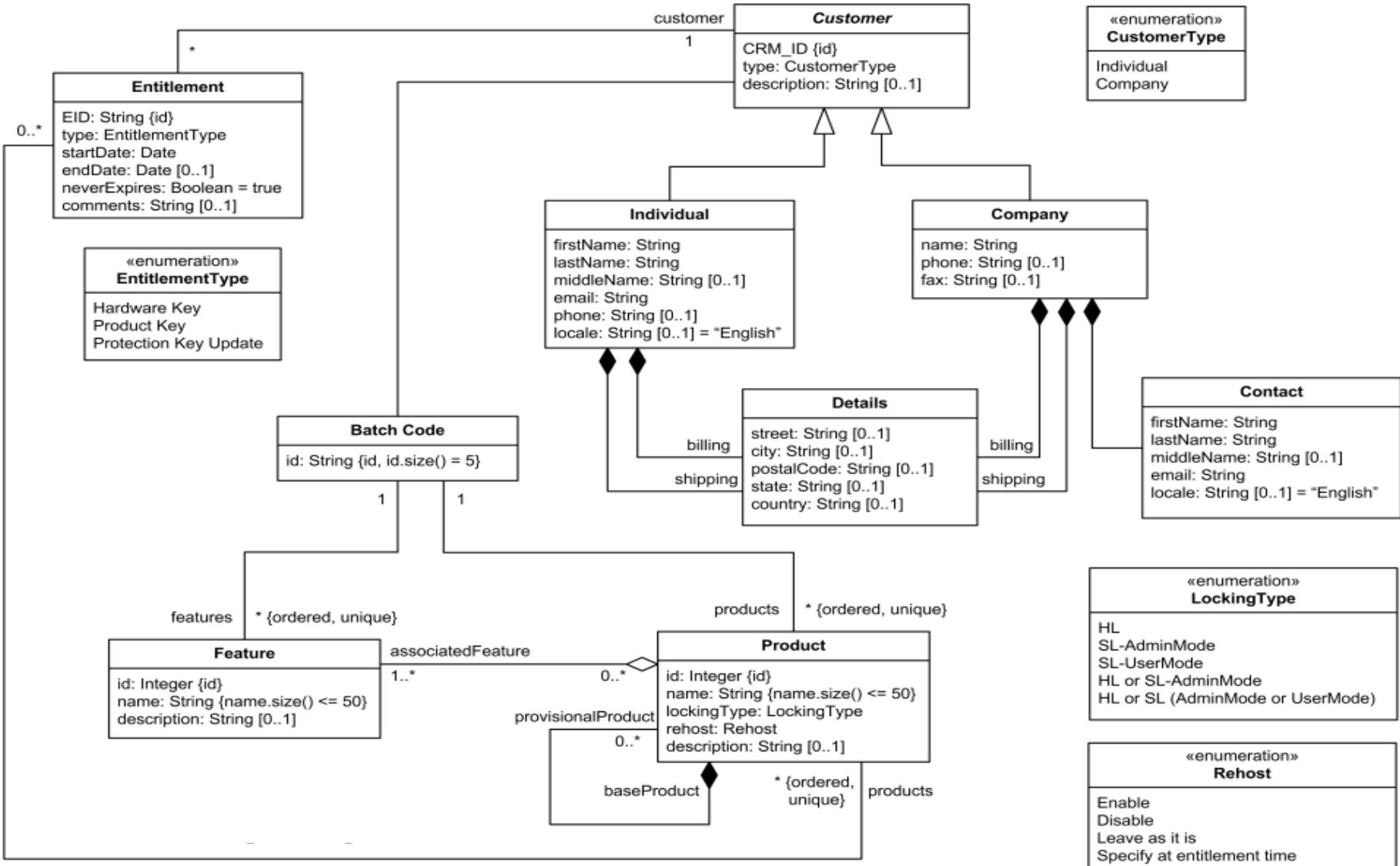
- Example: Sentinel HASP Licensing Domain
 - Each protected software **product** has some features and is associated with a batch code. A product name that identifies the product is unique in the selected batch. The maximum length for a product name is 50 characters. Product has some specific **locking type** (type type of protection) to be applied to the product.
 - The basic unit on which all products are created is the **base product**. A base product can contain all the product attributes such as features, licensing data and memory, and can be used as a product offered for sale. **Provisional products** could be also defined to be distributed as trialware. The properties for provisional products are not completely identical to those of the base product. After a product has been defined, it can be included in **entitlements** (orders).
 - An **entitlement** is an order for products to be supplied with one or more Sentinel protection keys. Orders or sales department receives and fulfils entitlements. Order processing personnel process the entitlement details using Sentinel EMS. An entitlement can contain one or more products. When entitlement is defined in Sentinel EMS, they can specify the **customer** who placed the order. The customer could be either an **individual** customer or a **company**.

Building class diagrams

- Example: Sentinel HASP Licensing Domain
 - The locking type assigned to a product may determine the type of entitlement that can be produced. They cannot add a product defined only with the HL locking type and another product defined only with the SL locking type (whether AdminMode or UserMode) to the same entitlement:
 - Products defined only with the HL locking type can be included in entitlements for HASP HL keys, product keys, or for protection key updates.
 - Products defined only with the SL AdminMode or SL UserMode locking type can be included only in entitlements for product keys or for protection key updates.
 - Products defined with the HL or SL AdminMode or HL or SL AdminMode or SL User-Mode locking type can be included in entitlements for HASP HL keys, product keys, or for protection key updates.

Building class diagrams

Example: Sentinel HASP Licensing Domain

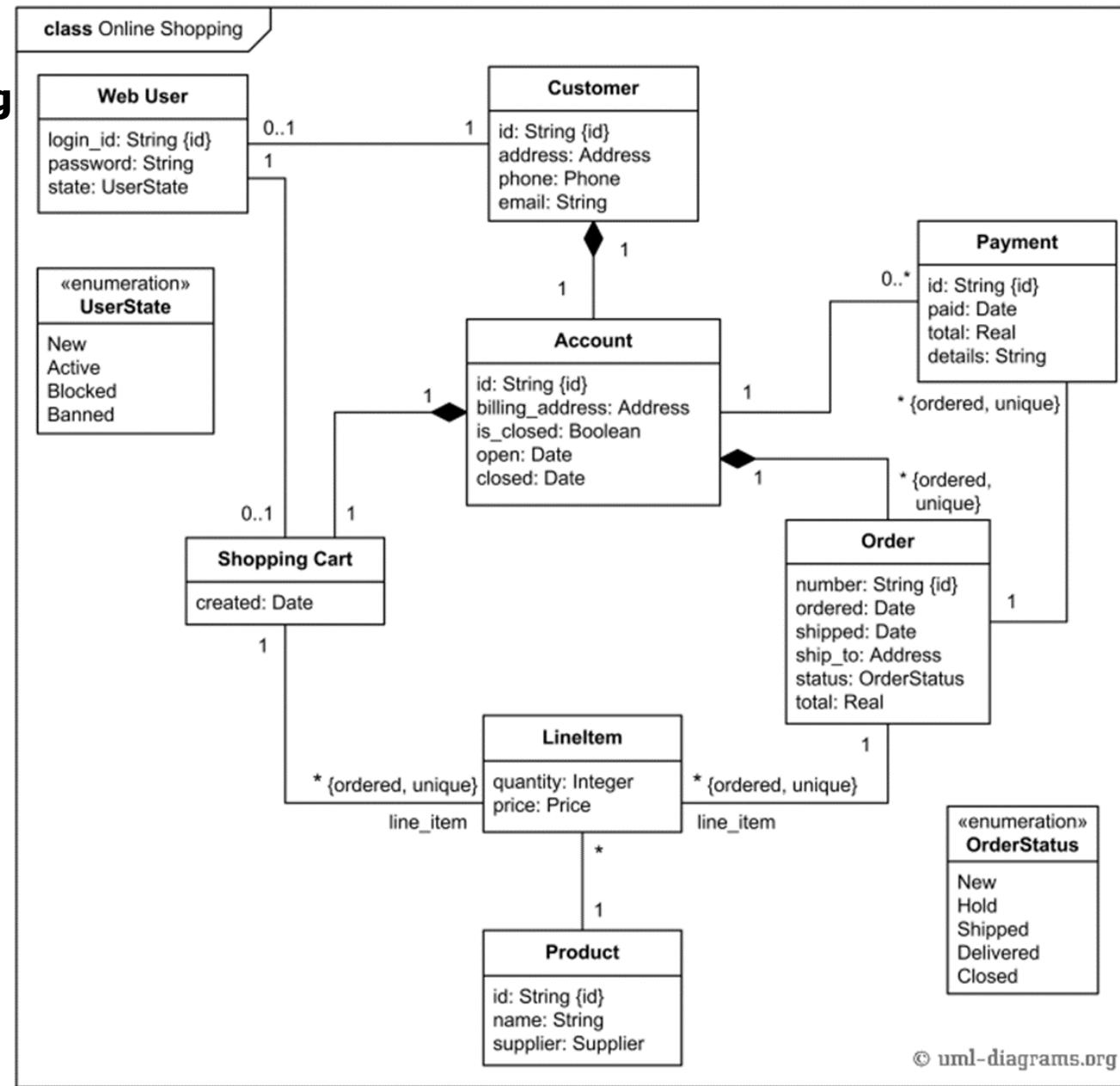


Building class diagrams

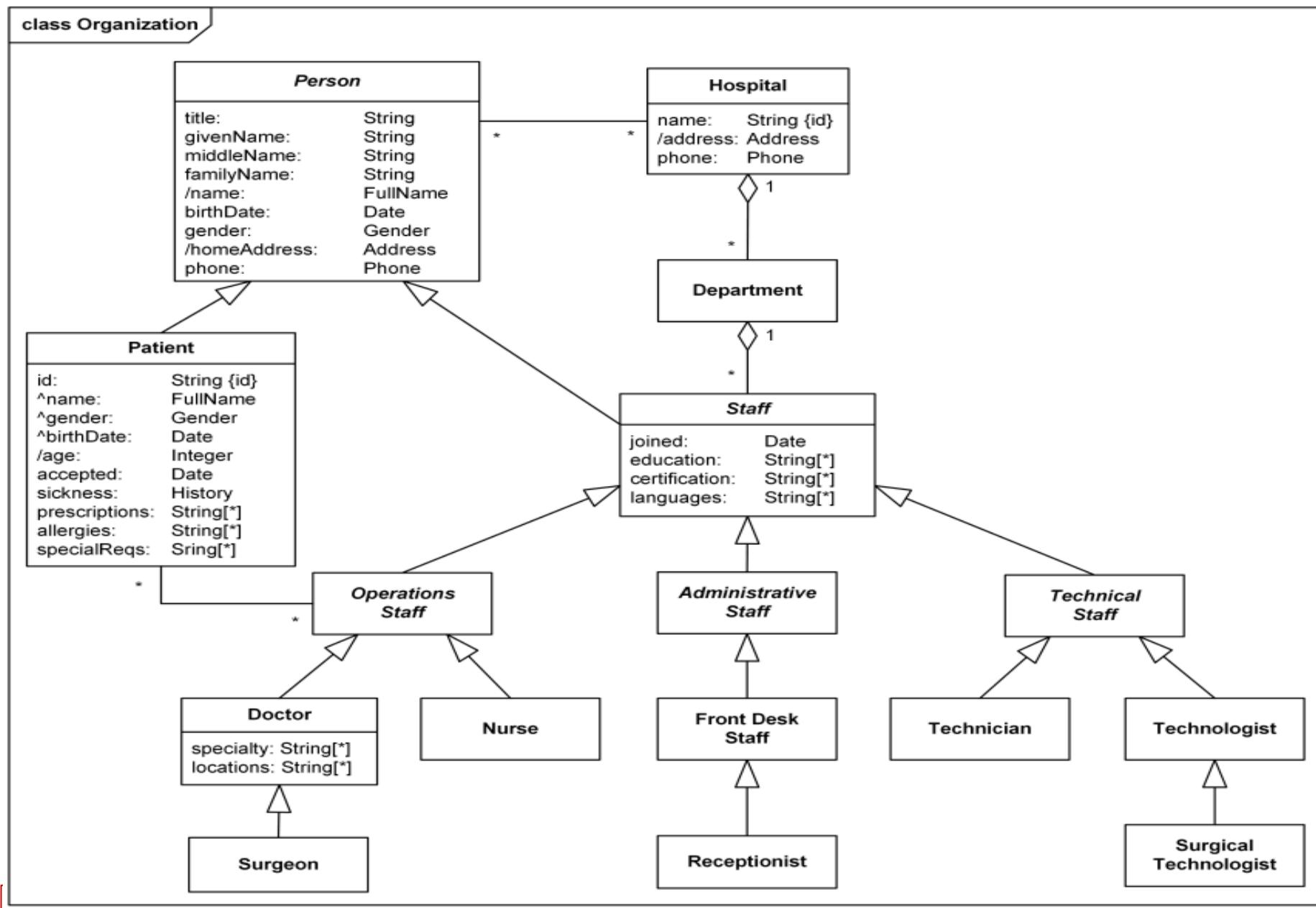
- Example: **Online Shopping**
- Each customer has unique id and is linked to exactly one **account**.
Account owns shopping cart and orders. Customer could register as a web user to be able to buy items online. Customer is not required to be a web user because purchases could also be made by phone or by ordering from catalogues. Web user has login name which also serves as unique id. Web user could be in several states - new, active, temporary blocked, or banned, and be linked to a **shopping cart**. Shopping cart belongs to account.
- Account owns customer orders. Customer may have no orders.
Customer orders are sorted and unique. Each order could refer to several **payments**, possibly none. Every payment has unique id and is related to exactly one account.
- Each order has current order status. Both order and shopping cart have **line items** linked to a specific product. Each line item is related to exactly one product. A product could be associated to many line items or no item at all.

Building class diagrams

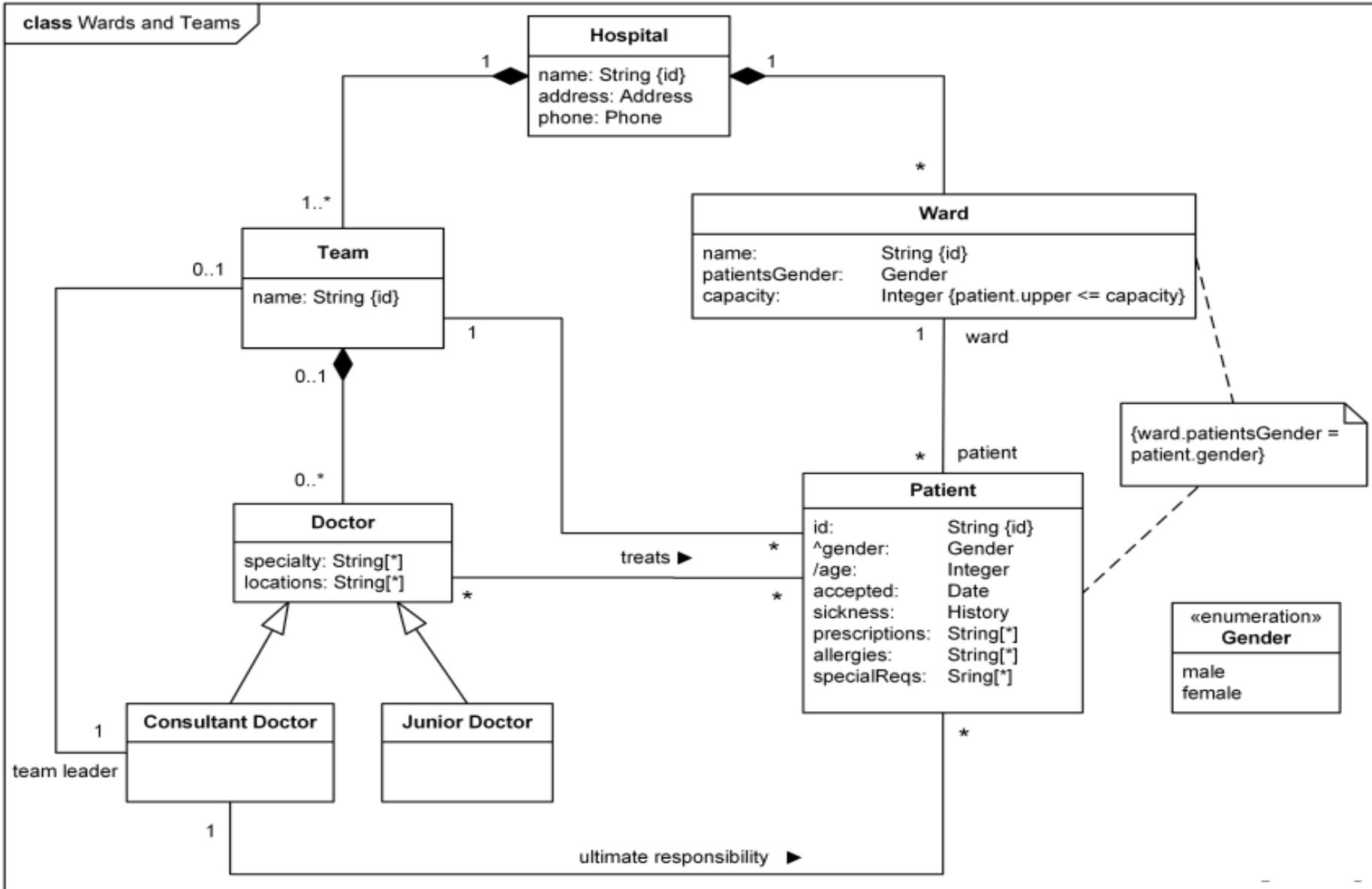
- Example:
Online Shopping



Building class diagrams

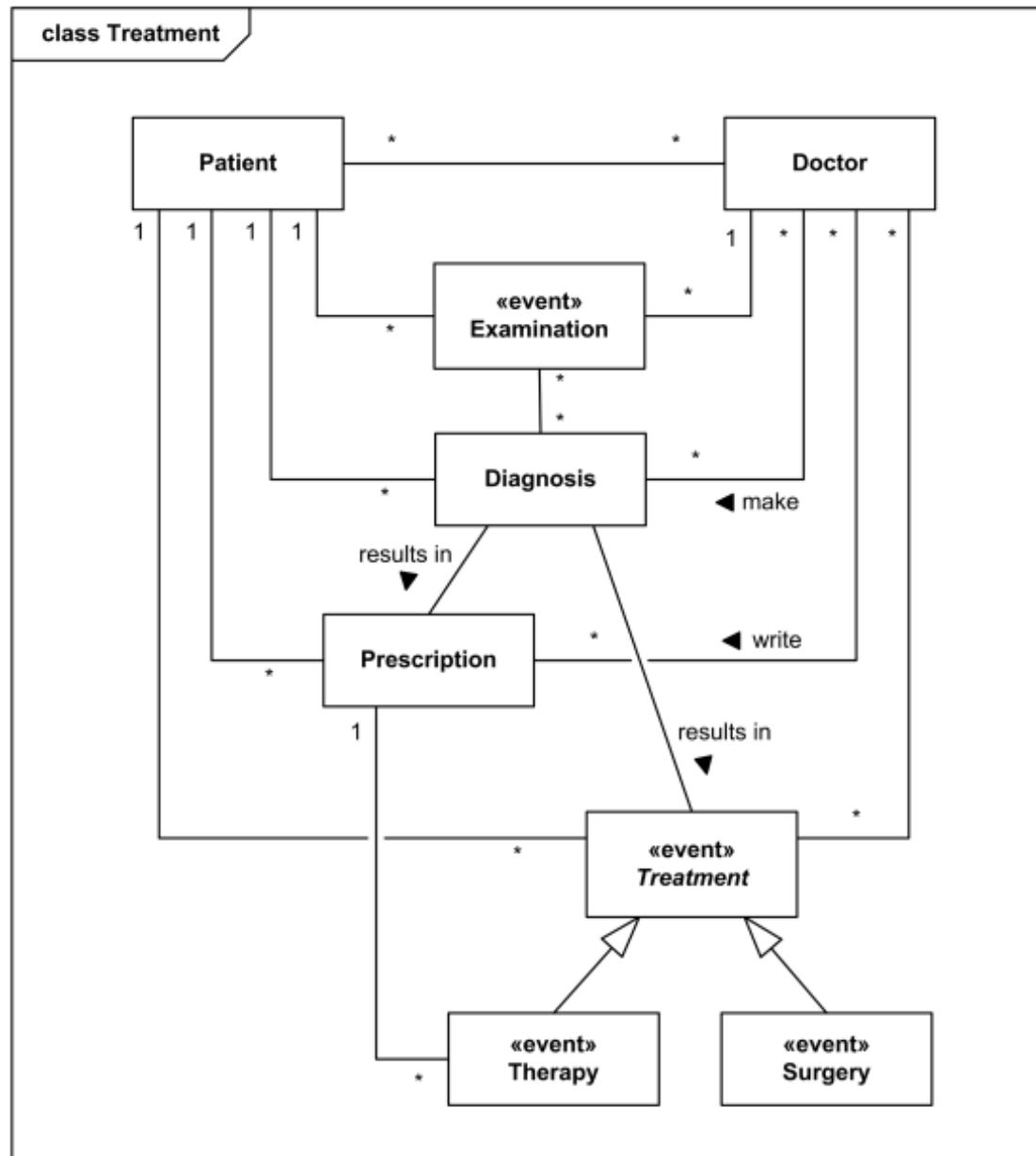


Building class diagrams



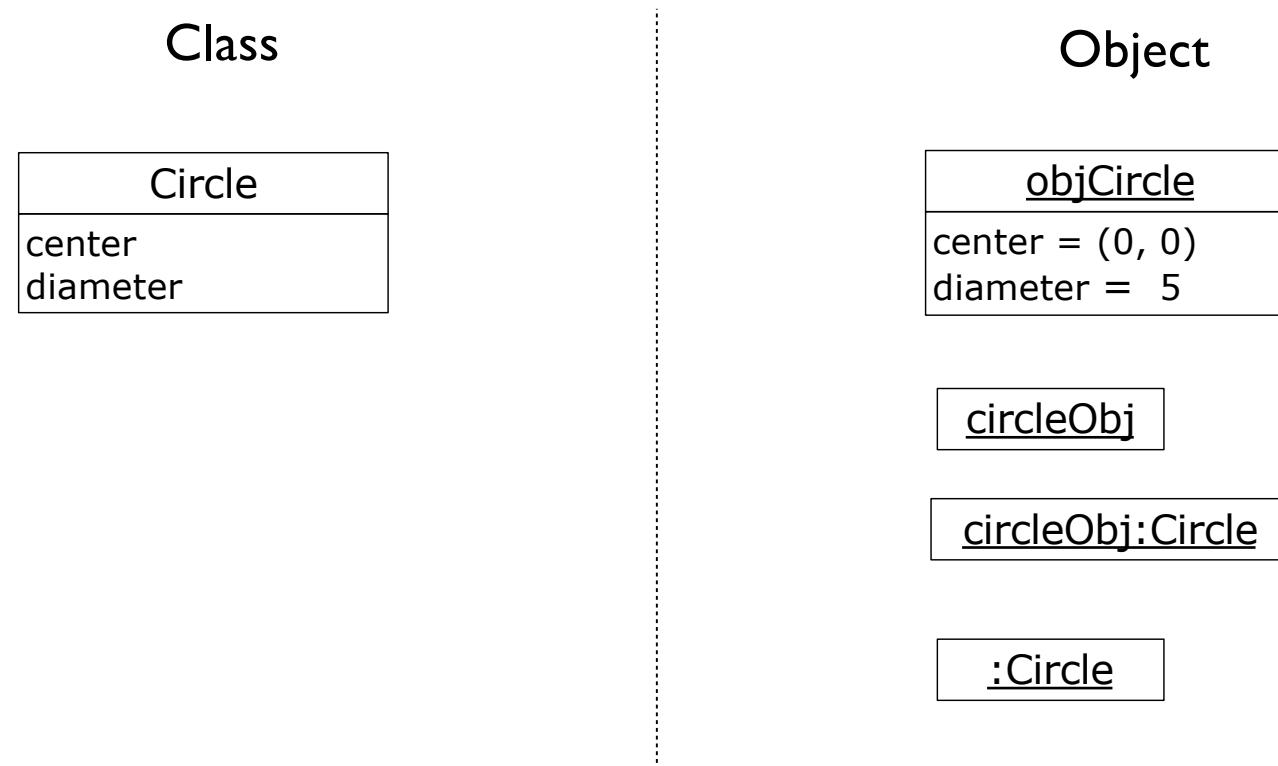
Building class diagrams

- Example:



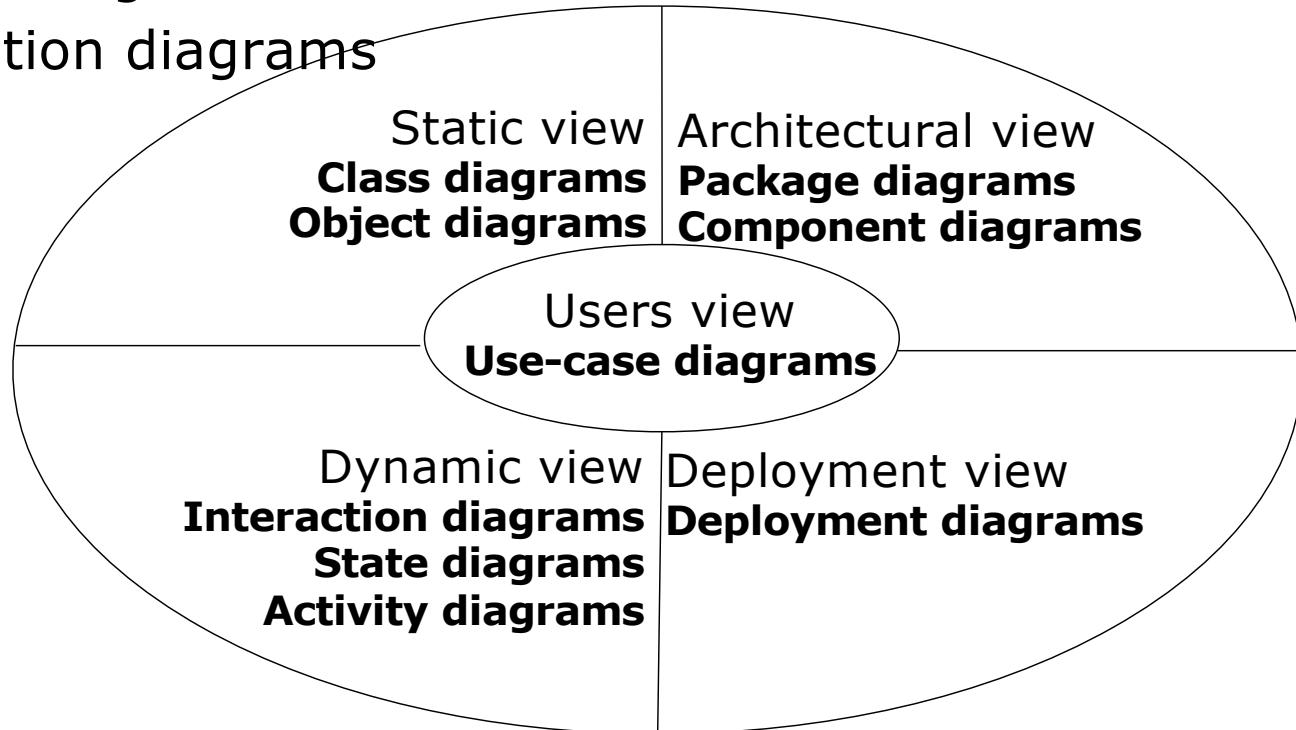
Object diagrams

- Objects
 - Objects are instances of classes
 - Notation
 - Values of attributes can be indicated
 - Name of object is underlined



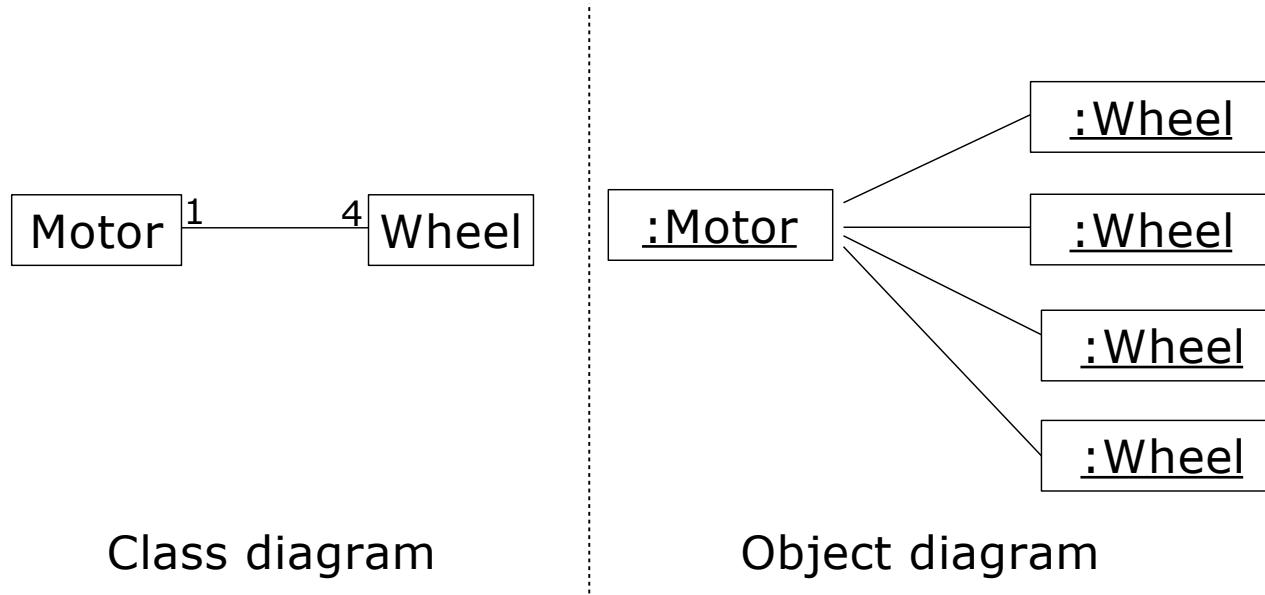
Object diagrams

- Objects
 - Three types of diagrams with objects
 - Static view
 - Object diagrams
 - Dynamic view
 - Sequence diagrams
 - Collaboration diagrams



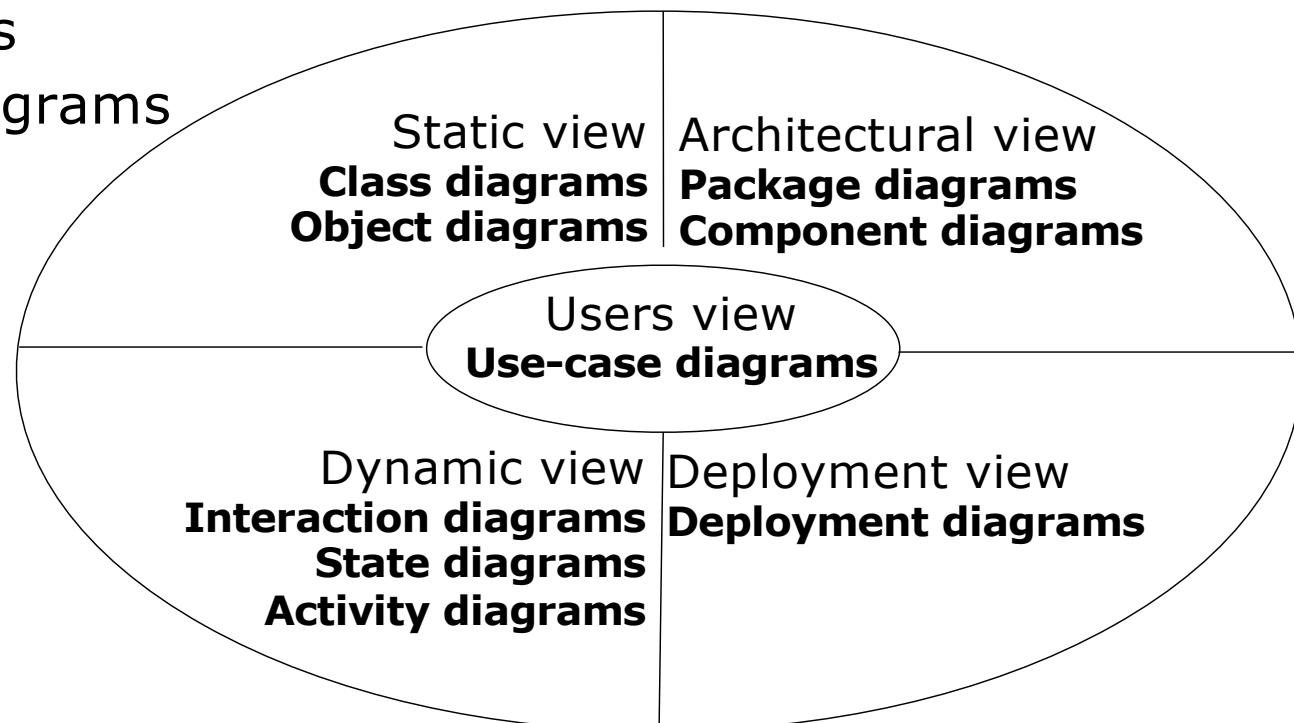
Object diagrams

- Object diagrams
 - represent a set of objects and links between them
 - are static views of instances of the elements appearing in class diagrams
- An object diagrams is an instance of a class diagram



Modelling dynamic behaviour

- Activity diagrams
- State diagrams
- Interaction diagrams



Main Activities of Software Development

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

Deployment

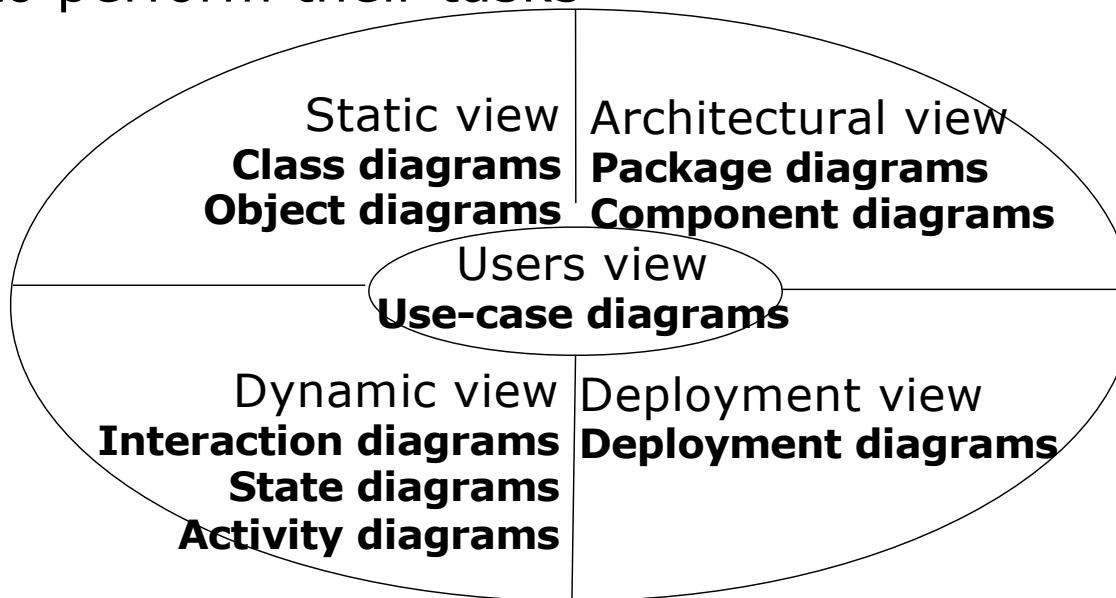
Installation and training

Maintenance

Post-install review
Support docs
Active support

Modelling dynamic behaviour

- Describing what happens during the execution of the system
 - The behaviour of the objects
- Dynamic behaviour modelling allows to complete the information in the static diagram
 - How the elements in the static diagrams
 - provide the functionality of the system
 - change their states
 - communicate with each other
 - cooperate to perform their tasks



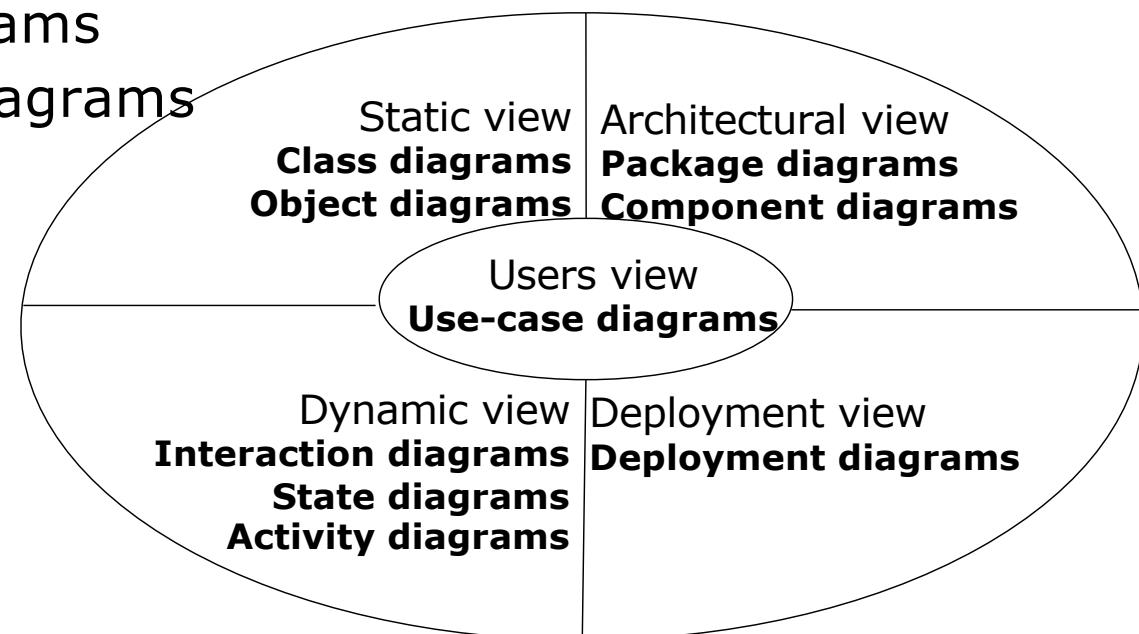
Modelling dynamic behaviour

- Difficulties
 - Modelling of the dynamic behaviours of a complex system is always difficult
 - Too many features
 - Too many paths
 - The collaborations between objects are complicated
 - It is not easy to allocate and carry out responsibilities

- Suggestions
 - Focusing on the communication of one dynamic aspect of the system to better master the complexity
 - Modelling only the essential elements
 - Providing a detail suitable for each level of abstraction

Modelling dynamic behaviour

- Diagrams
 - Activity diagrams
 - High level dynamic behaviour
 - Performing objectives of the system
 - State diagrams
 - Internal behaviour of the system
 - Interaction diagrams
 - Communication between objects
 - Sequence diagrams
 - Collaboration diagrams



Activity diagrams

- Allowing to determine the dynamic behaviour of the system from **one or several use-cases**
- Being useful to model the flows of treatment in the system
- Modelling the control flow and also the data stream

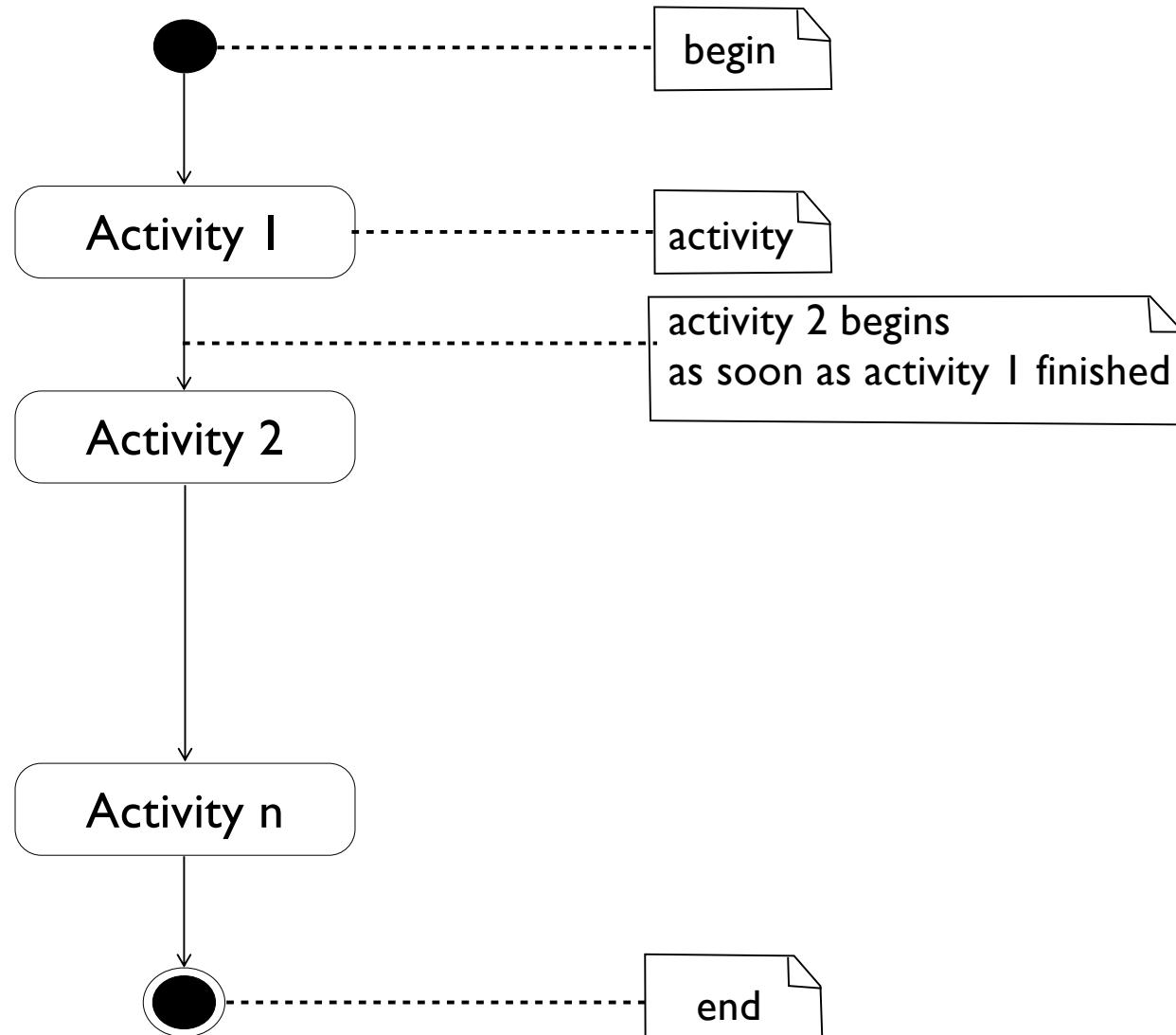
- An activity diagram includes
 - the activities carried out by the system and the actors
 - the order in which these activities are carried out
 - the possible dependencies between activities.

Activity diagrams

- An activity corresponds to a high-level task in the system
- Distinction between the activities and operations in the static structure
 - Activities are carried out by the system or the actors
 - Operations are related to classes
 - In general, activities do not correspond to operations
- Activity diagrams are generally built before (design) class diagrams
 - Activity diagrams are used to determine which operations to add to class diagrams

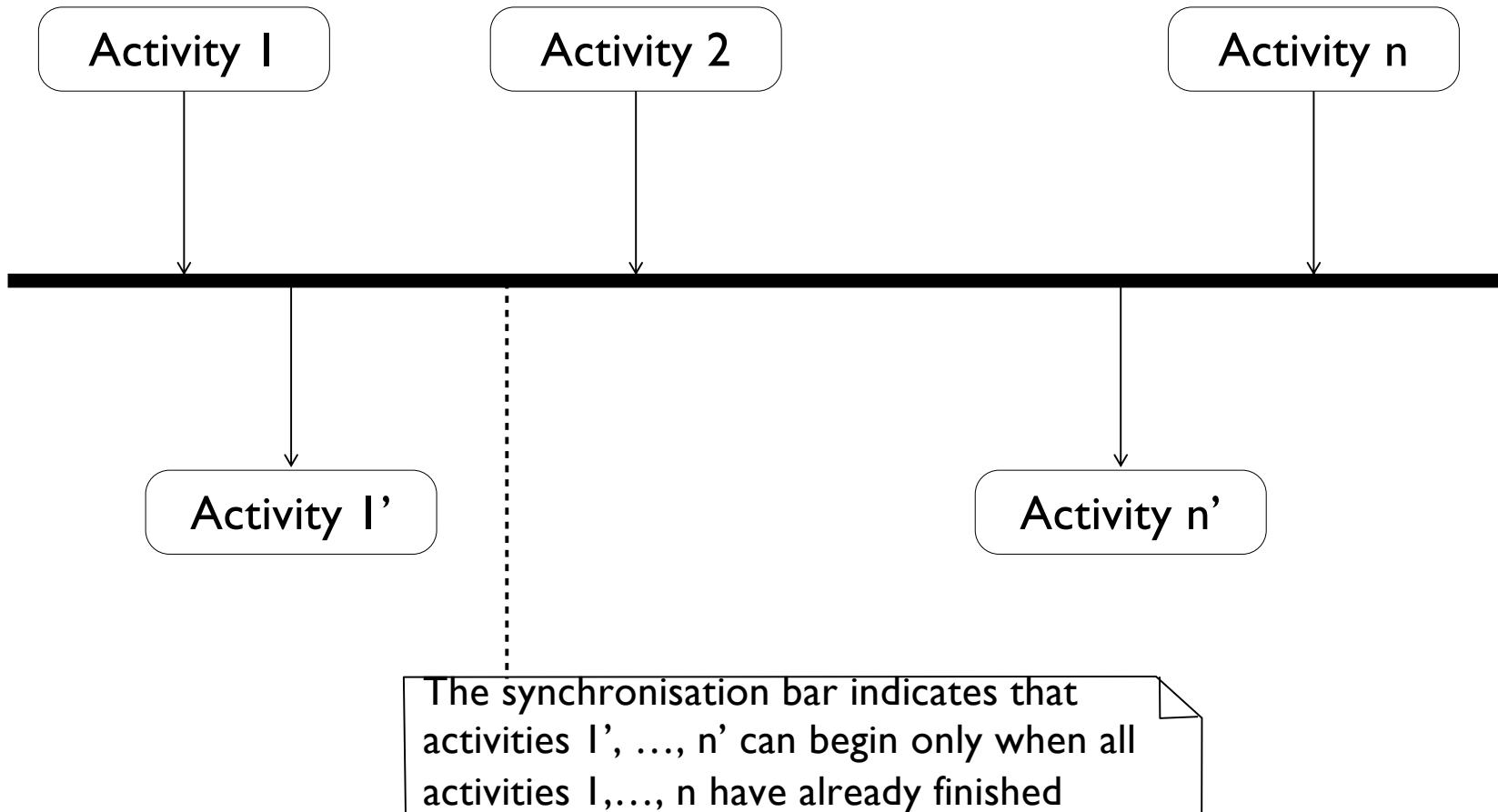
Activity diagrams

□ Notation



Activity diagrams

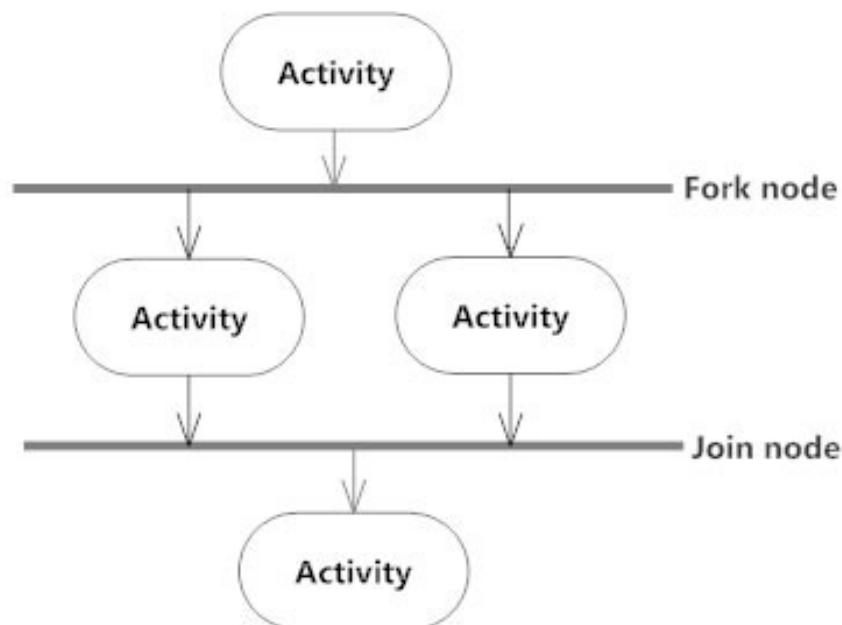
- Synchronisation of activities



Activity diagram

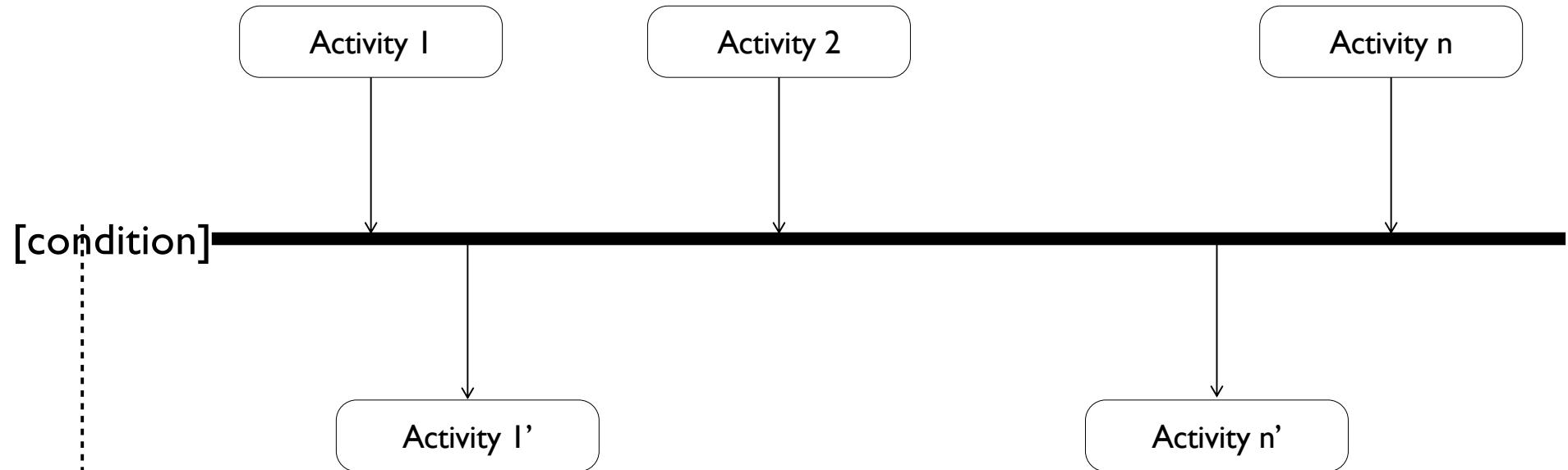
□ Synchronisation

- A **fork node** is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- A **join node** joins multiple concurrent flows back into a single outgoing flow.
- A fork and join node used together are often referred to as synchronization.



Activity diagram

- Synchronisation guarded by a condition

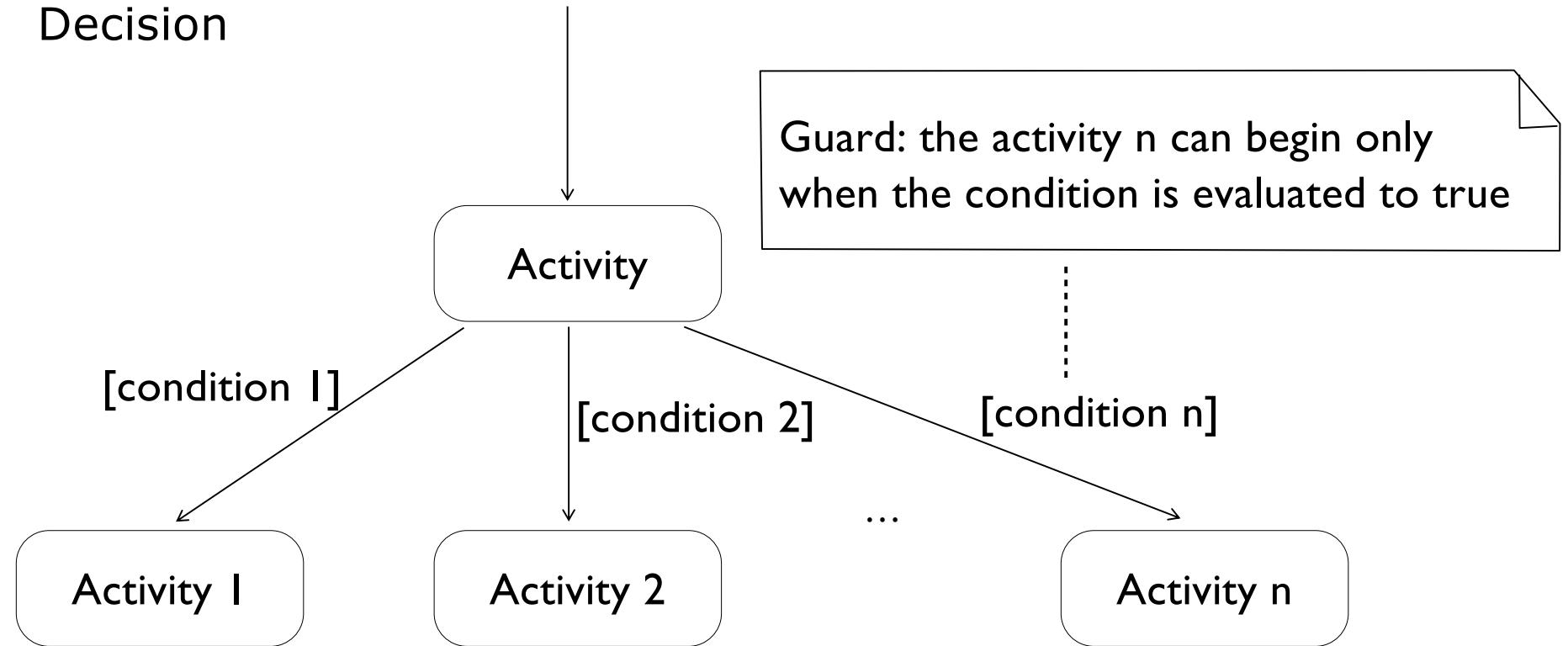


The guard expresses that the condition must be evaluated whenever one of the activities has finished. Activities 1', ..., n' can begin only when the condition becomes true

- The absence of guard can be considered as a special guard. This one becomes true when all the activities at the entrance to the synchronisation bar have finished

Activity diagrams

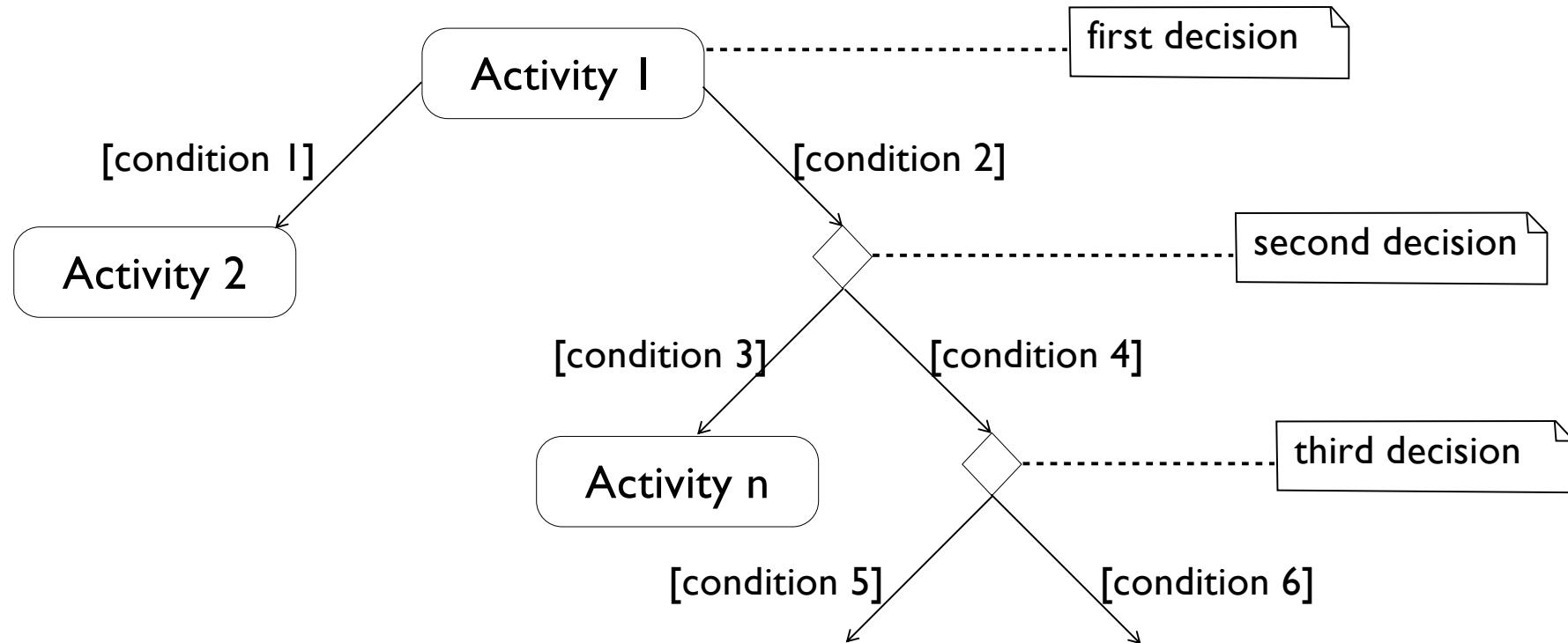
- Decision



- The transition guards coming out of the same activity should be mutually exclusive

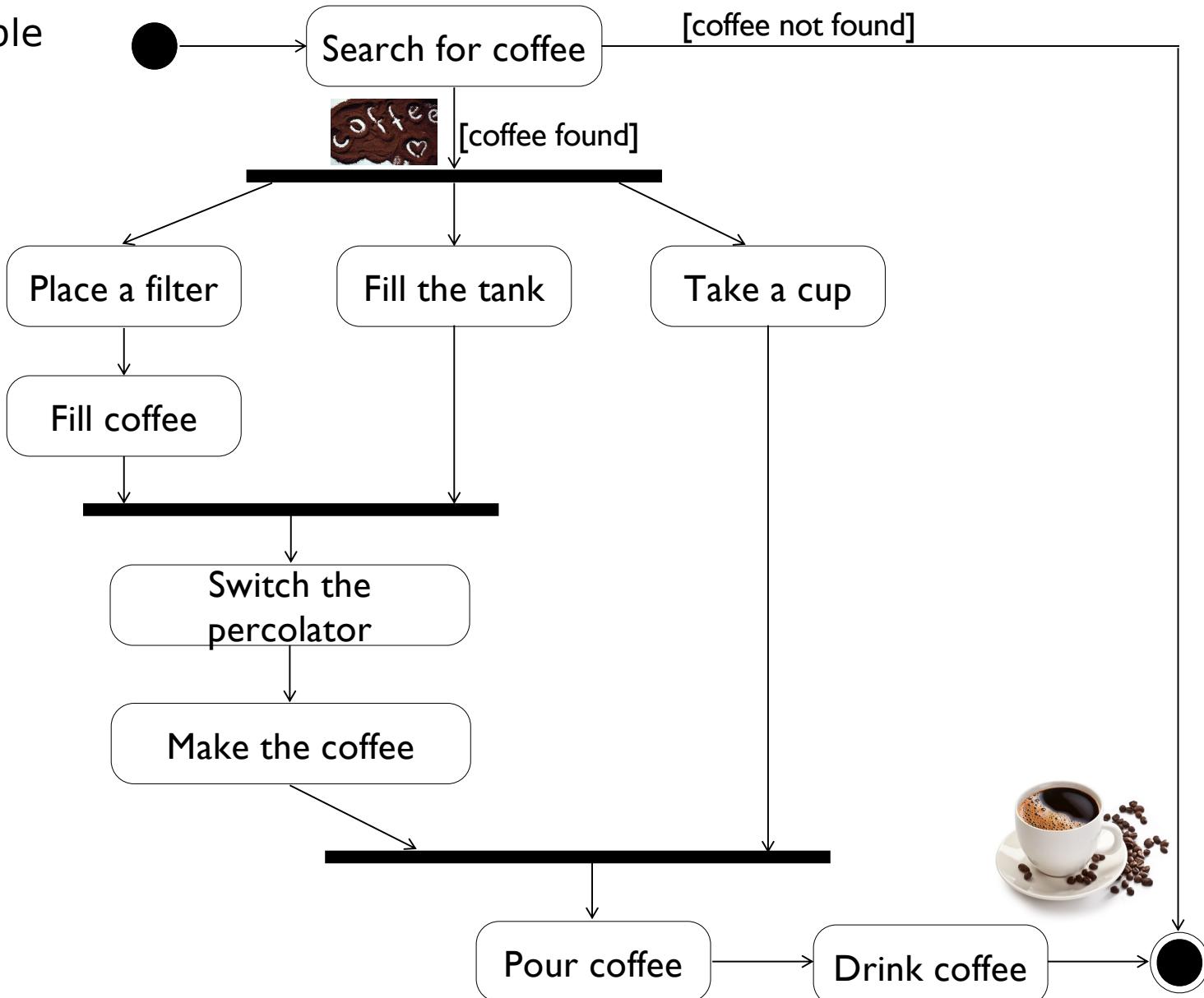
Activity diagrams

- Multiple decisions



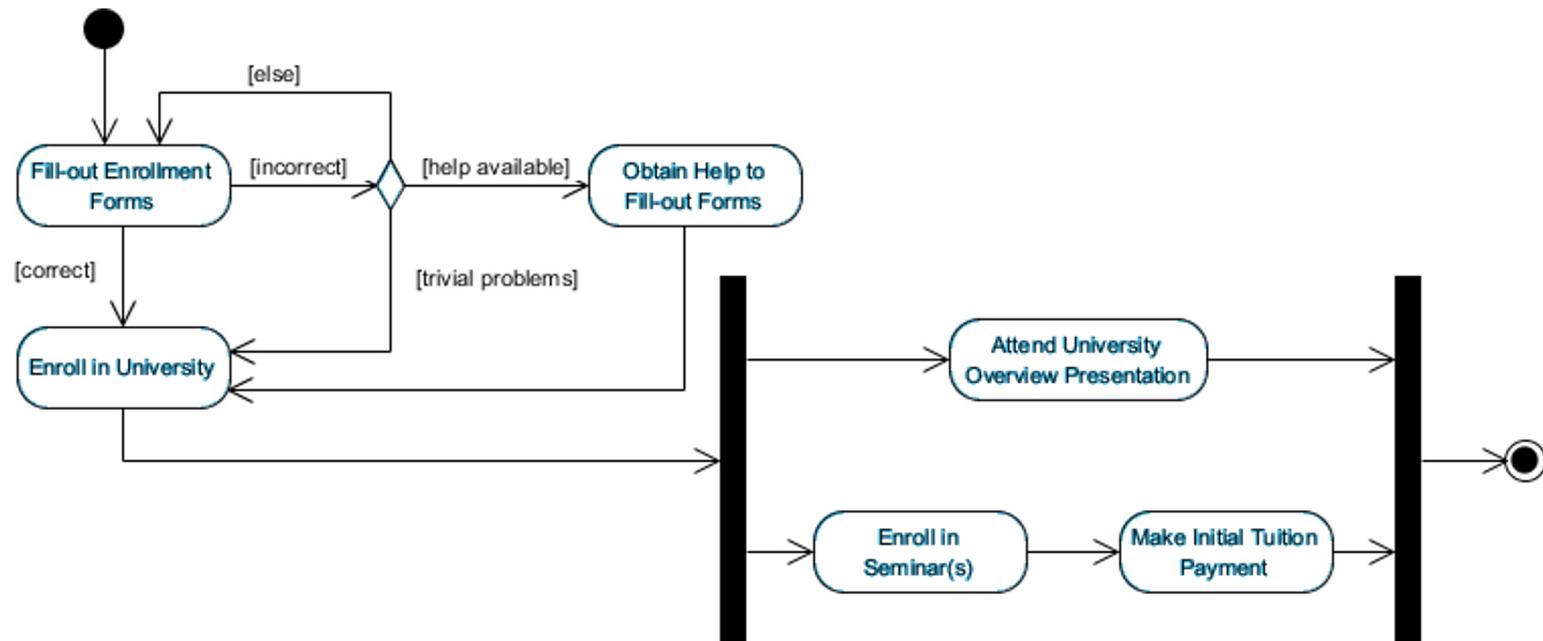
Activity diagrams

- Example



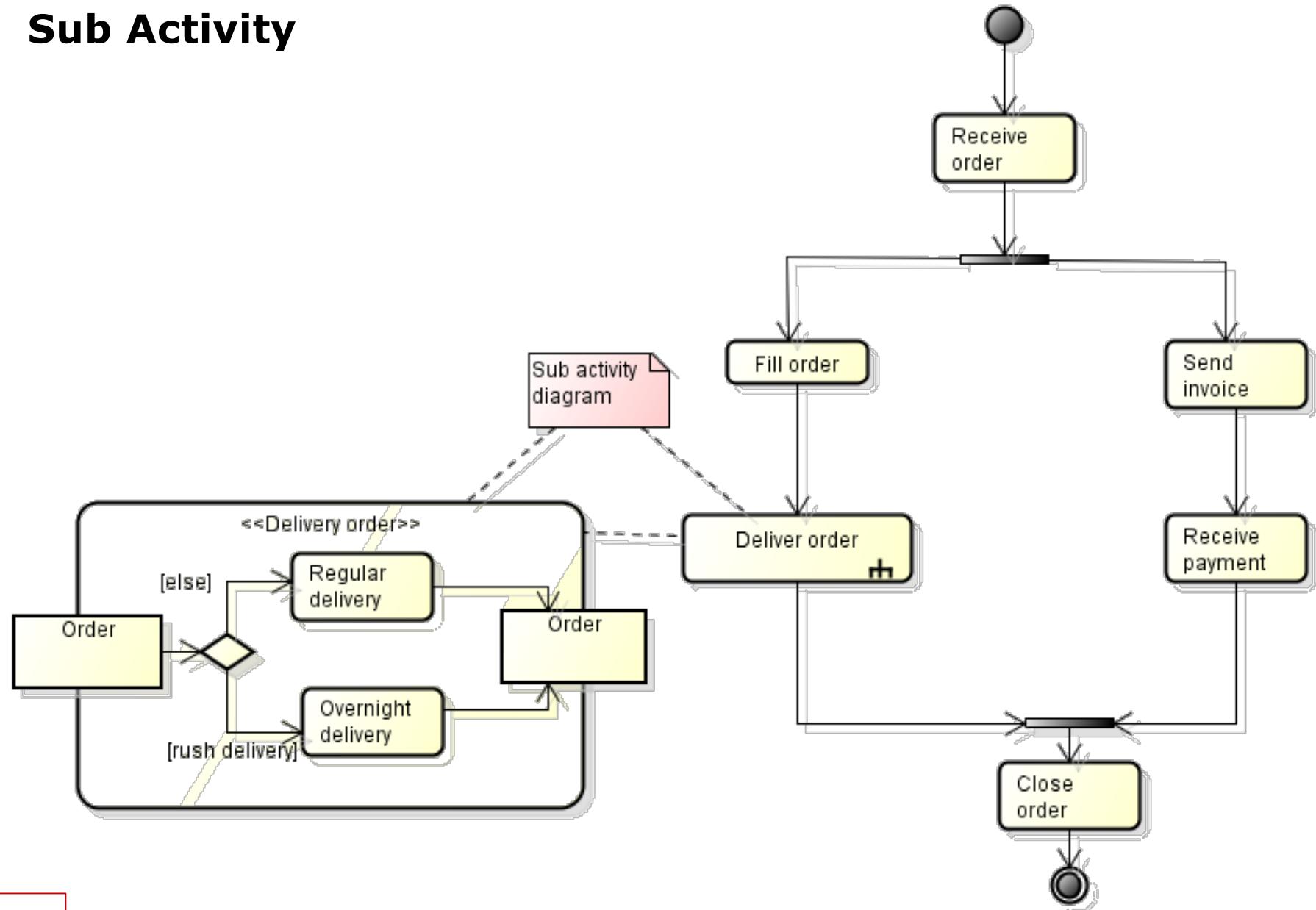
Activity diagrams

- Example - Student Enrollment
 - a process for student enrollment in a university as follows:
 - An applicant wants to enroll in the university.
 - The applicant hands a filled out copy of Enrollment Form.
 - The registrar inspects the forms.
 - The registrar determines that the forms have been filled out properly.
 - The registrar informs student to attend in university overview presentation.
 - The registrar helps the student to enroll in seminars
 - The registrar asks the student to pay for the initial tuition.



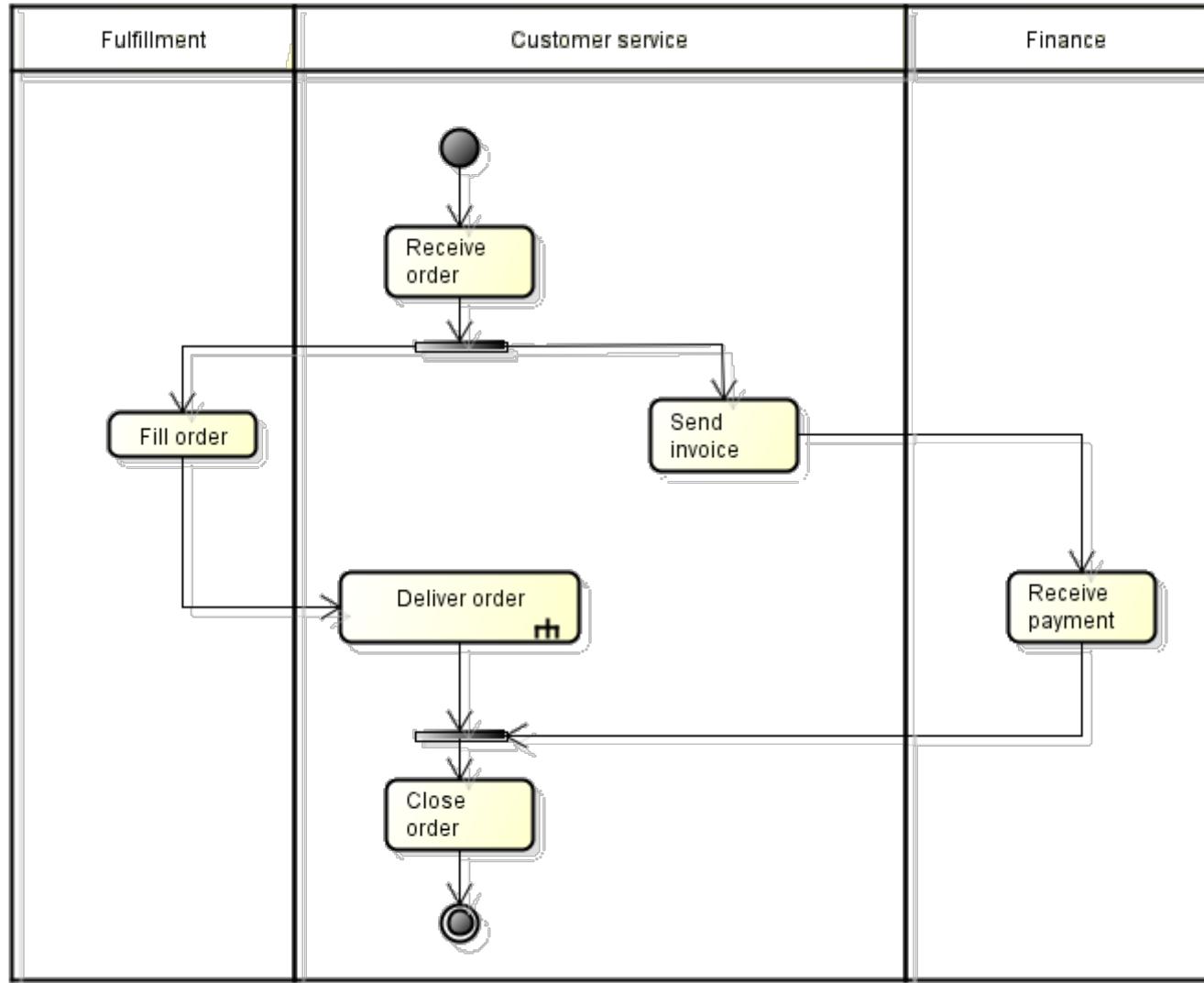
Activity diagram types

□ Sub Activity



Activity diagram types

□ Partitions



OOAD

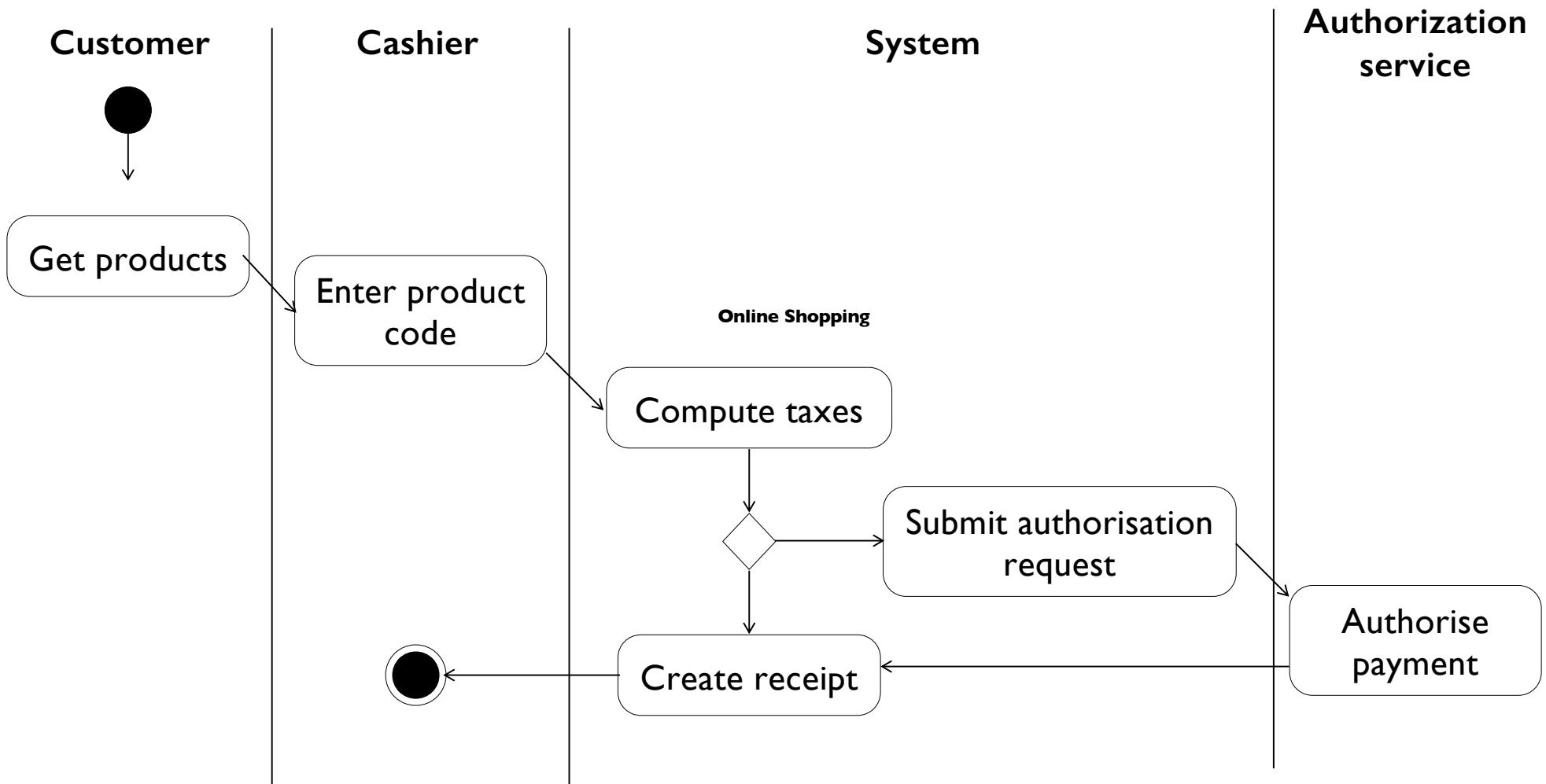
Activity diagram

- **Swimlane** helps to clarify on the activity diagrams the actors or components of the system that perform different activities
- Example: Activity diagram for “**sale**” use-case



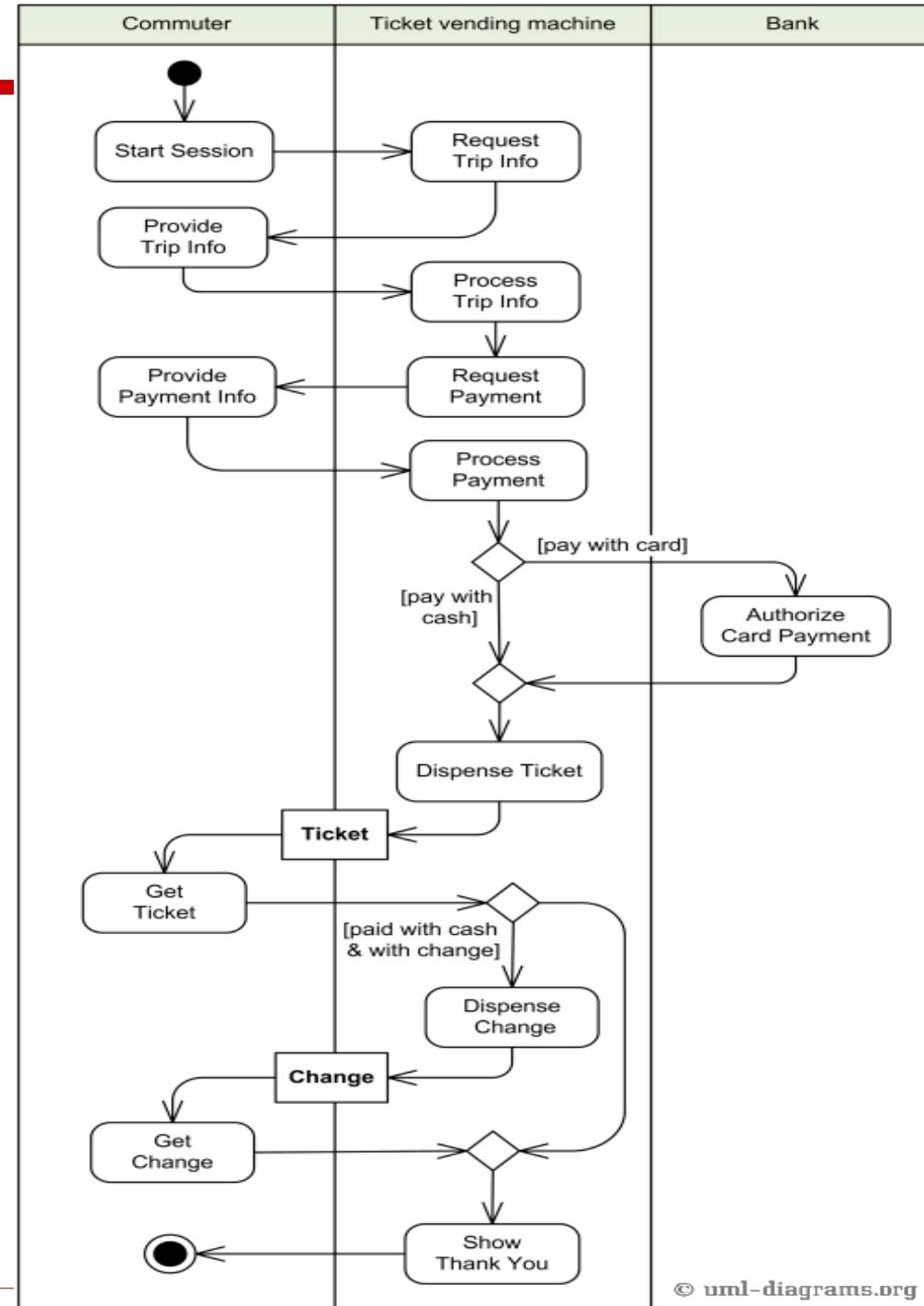
Activity diagram

- Example: Activity diagram for “**purchase products**” use-case

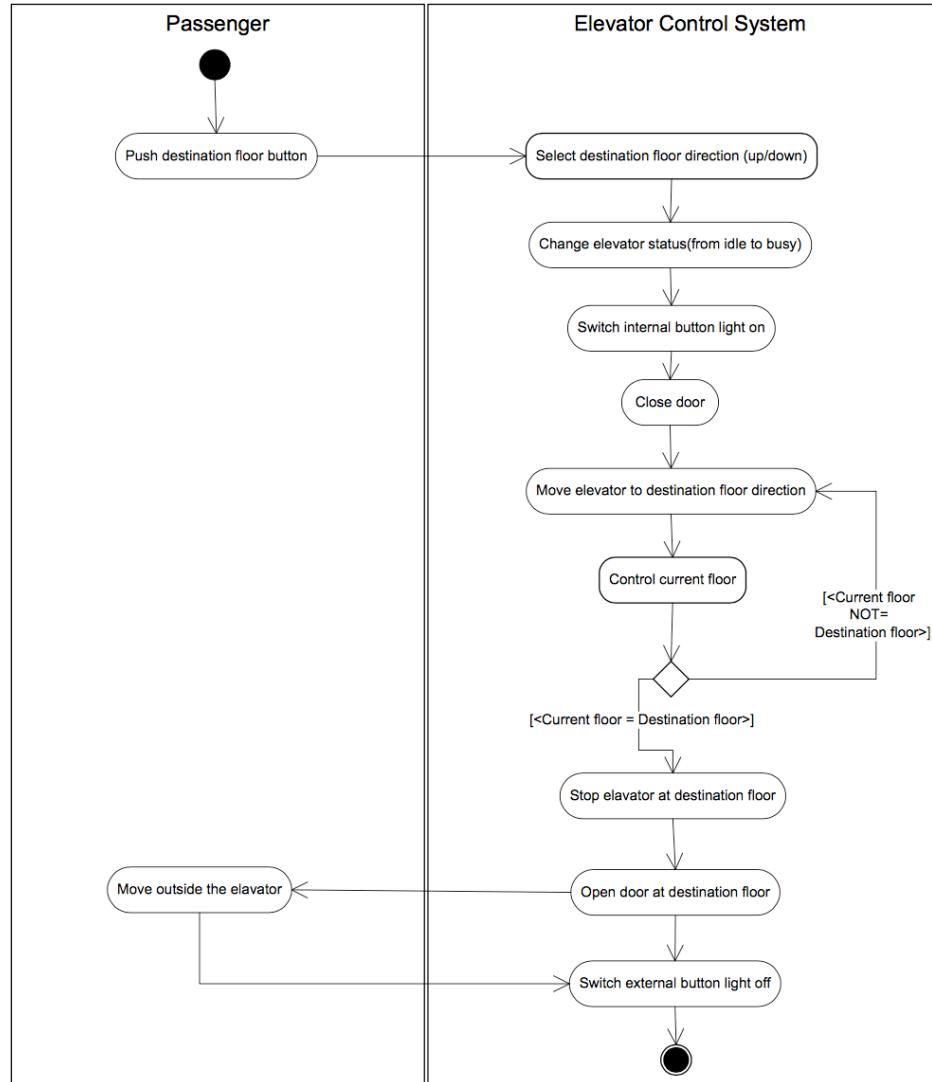


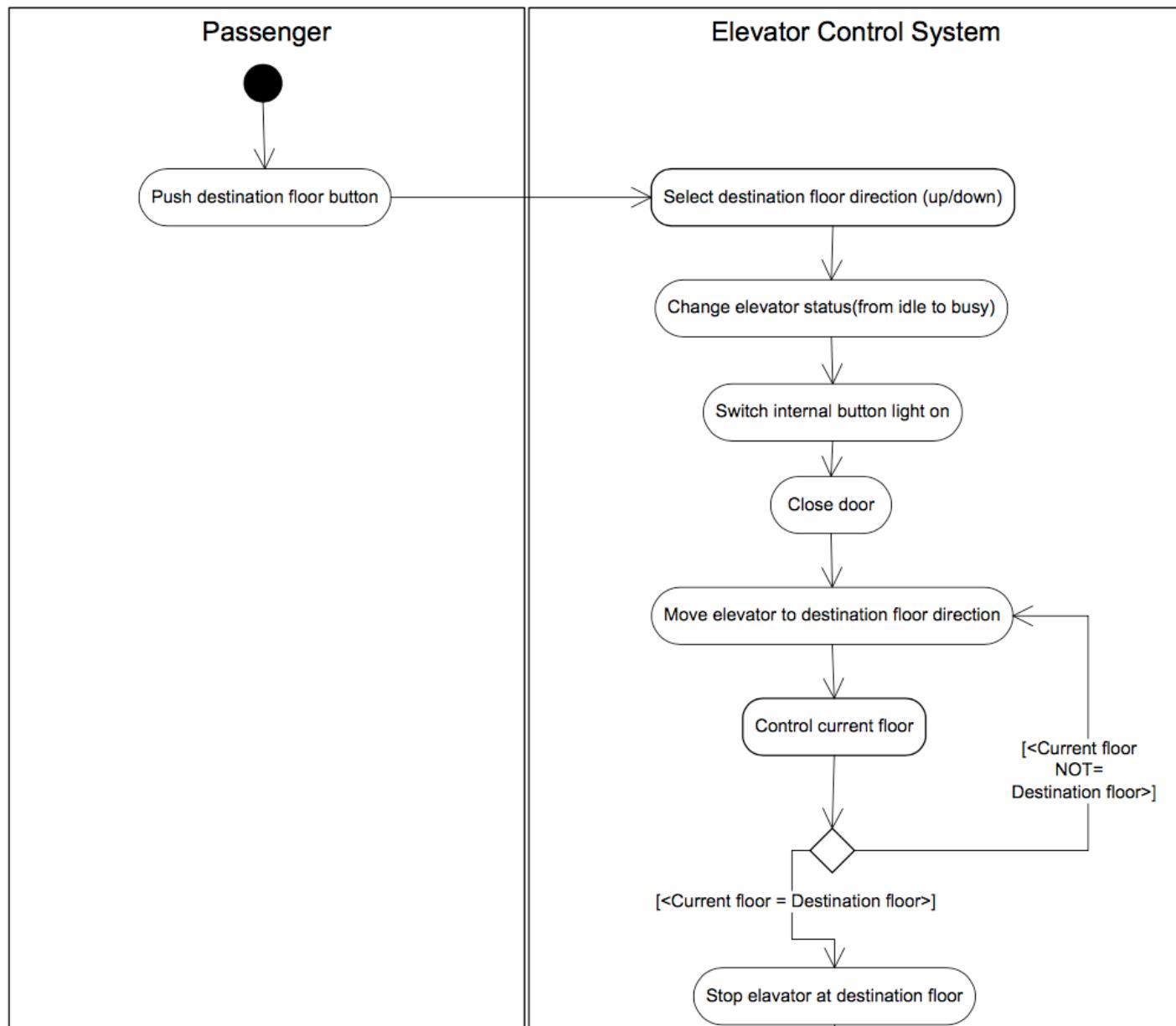
Examples

□ Ticket Vending Machine

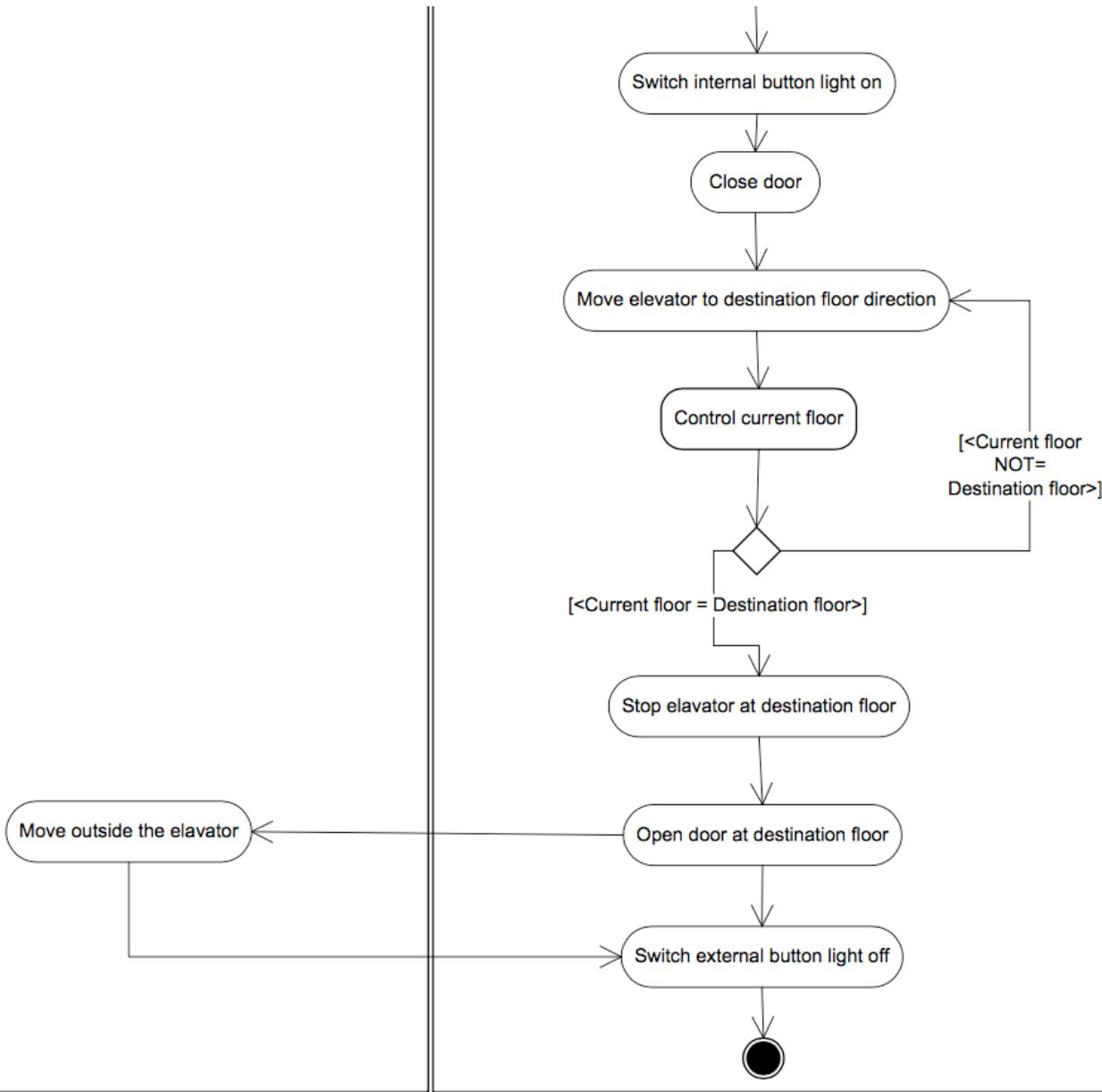


<i>Use case name</i>	Select Floor
<i>Participating actors</i>	Passenger
<i>Description</i>	It allows to move the passenger to the desired floor.
<i>Entry condition</i>	The passenger is inside the elevator
<i>Flow of events</i>	<ol style="list-style-type: none"> 1. The passenger pushes the destination floor button (internal button); 2. The system switches the floor button (internal button) light on; 3. The system closes the elevator door; 4. The system moves the elevator according to the destination floor direction (up/down); 5. The system stops the elevator at the destination floor; 6. The system opens the door at the destination floor; 7. The passenger moves outside the elevator; 8. The system switches the floor internal button off.
<i>Exit condition</i>	The passenger is outside the elevator in the desired floor; the elevator is <i>idle</i>



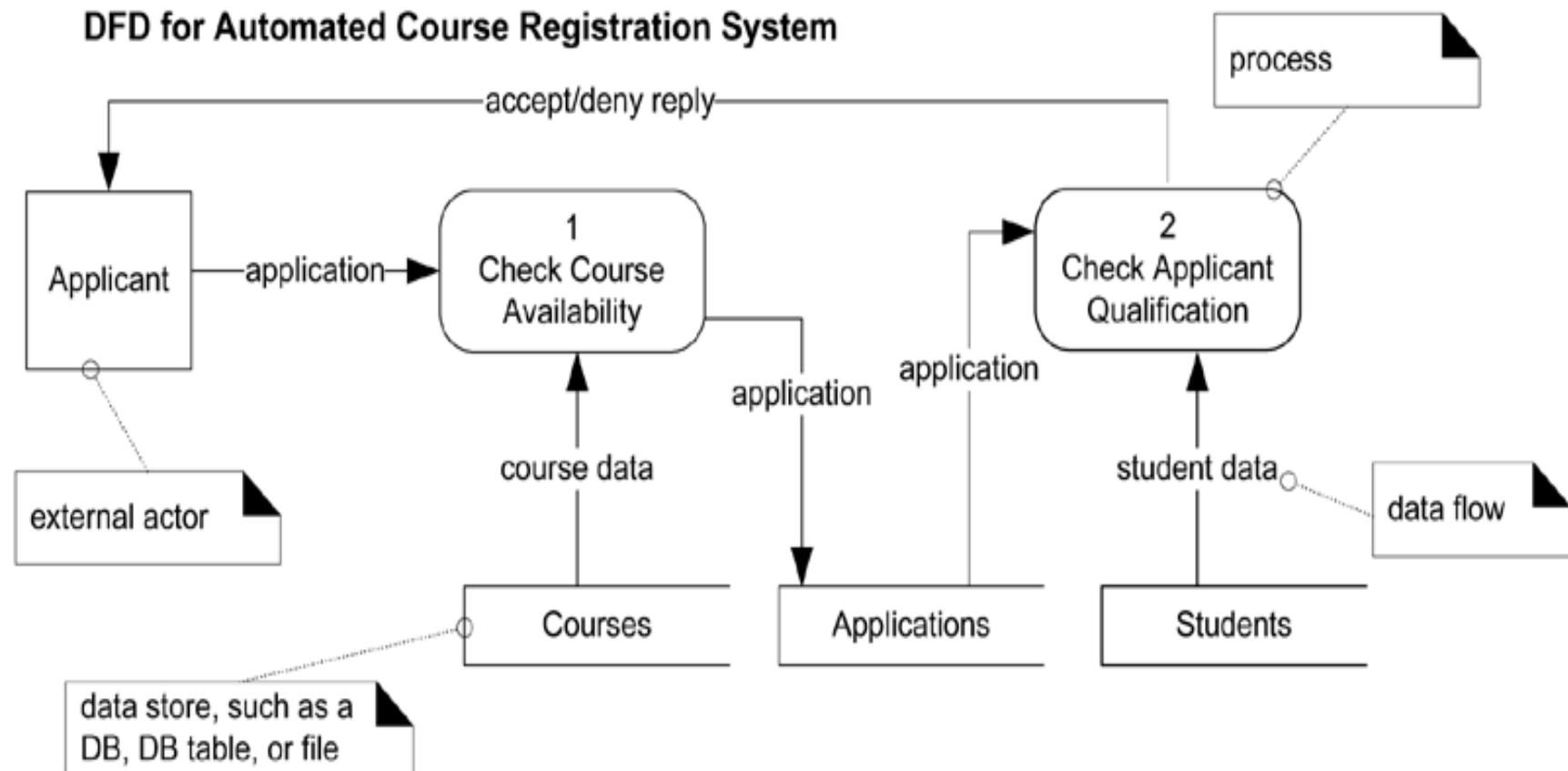


OOAD



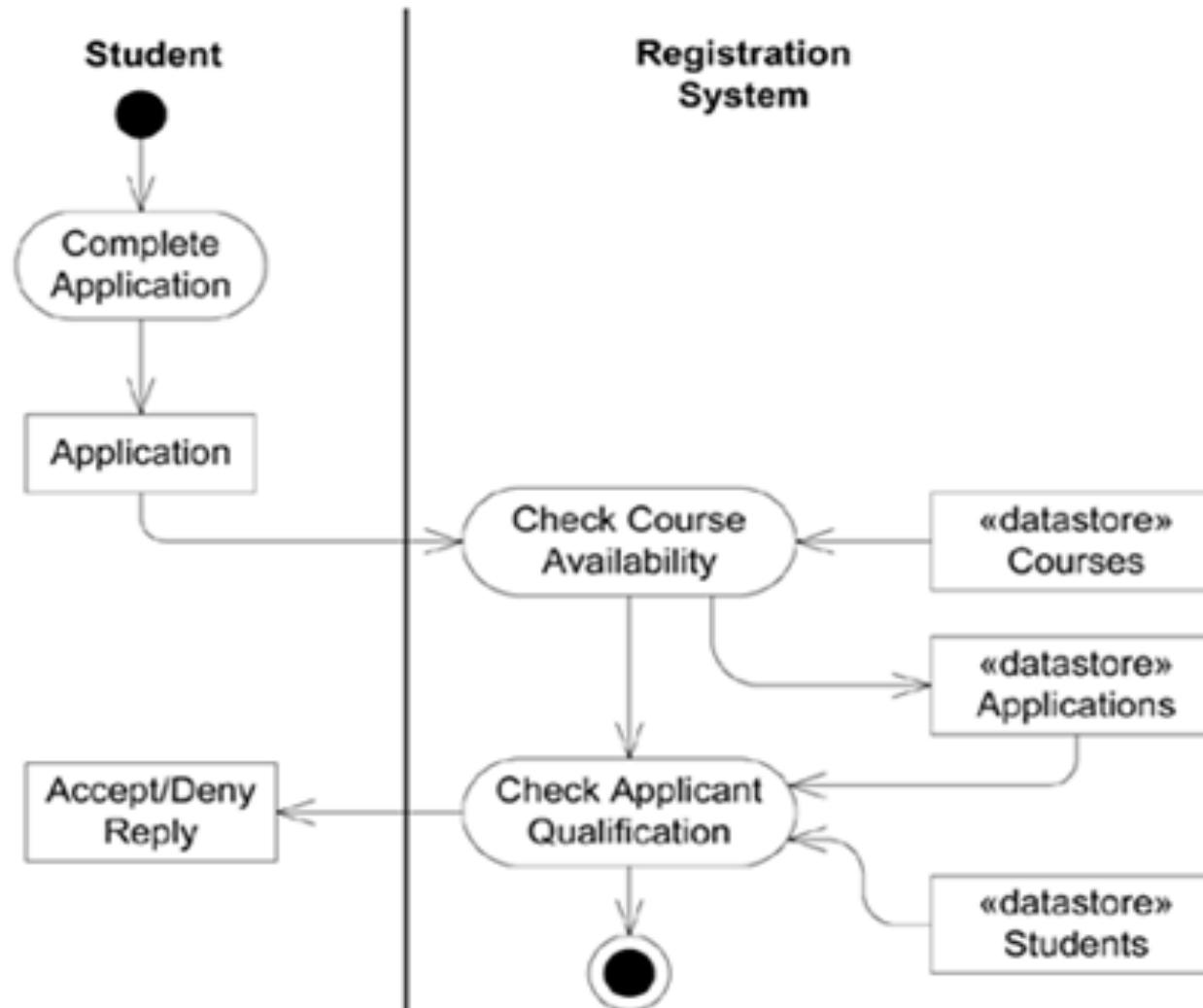
Activity diagram

- Activity Diagrams v.s. Data Flow Diagrams
 - Data Flow Diagram (DFD)



Activity diagrams

- Activity Diagrams v.s. Data Flow Diagrams
 - Activity Diagram



-
- Cả lớp đọc trước bài vẽ Sơ đồ trạng thái (State Diagram)

State Diagrams

- State diagrams
 - are **finite state automata**
 - allow to model the dynamic behavior of a collaboration or a class
 - focus on the behavior of objects, ordered by events
 - are especially used for modelling reactive systems

State diagrams

- State diagrams describe the behaviour of a system, part of a system or an object in the system
 - Each system or object has a **state** at a given time
 - In a given state, the system behaves in a specific manner to respond to the coming events
 - The **events** trigger state changes
- Specifically, a state diagram models the changes of states of a system/object in response to events
- A state diagram includes
 - **State**: state of a system/object at a given time
 - **Transition**: allows to switch a state to another
 - **Event**: activates the transition

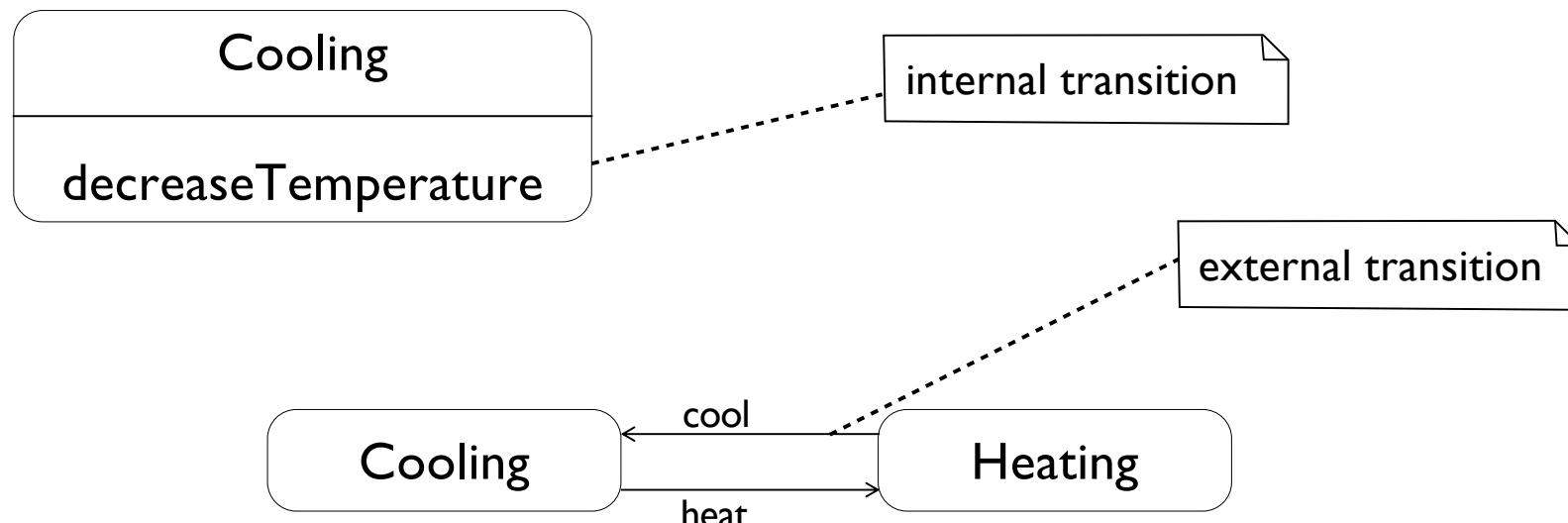
State diagrams

- State
 - Represents a situation of a system/object at an instance
 - System/object remains in a state for a while. Meanwhile, it can
 - perform certain **activities**
 - wait until an **event** occurs
- Notation



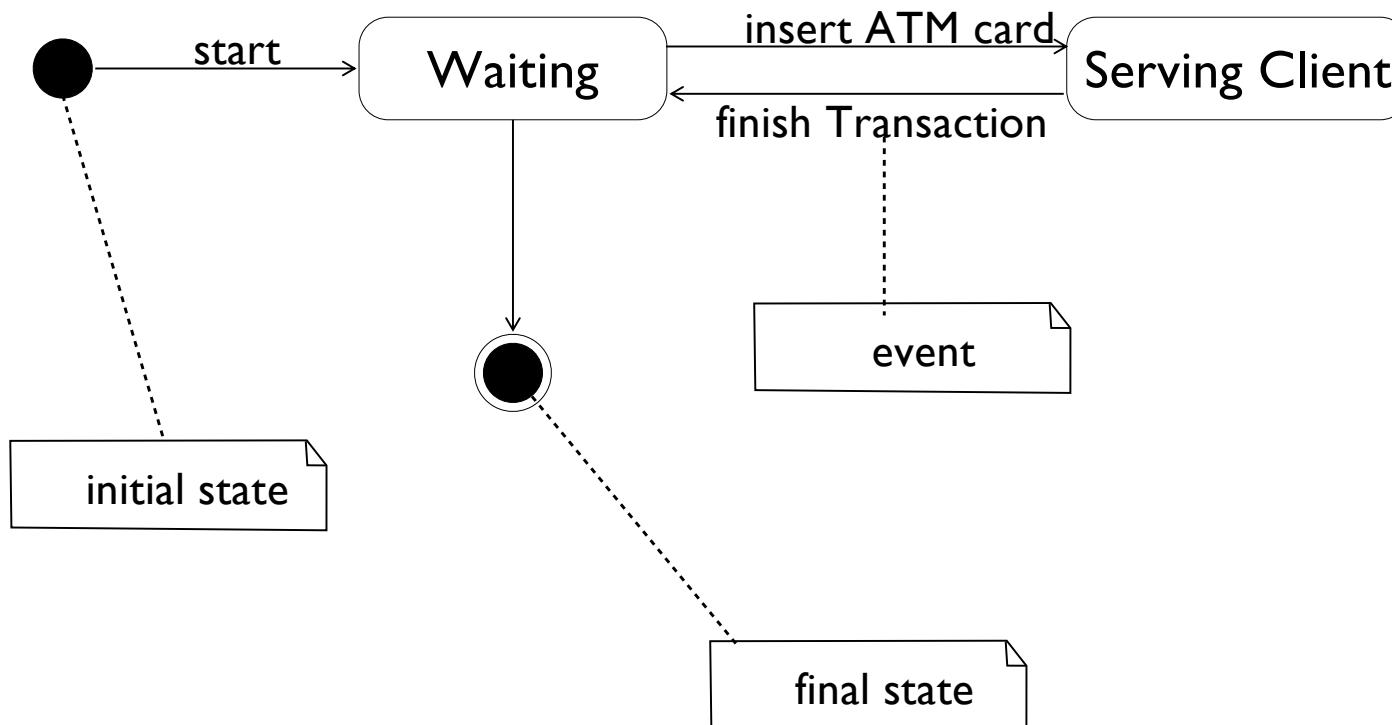
State diagrams

- Transitions
 - Transitions are related to actions that can be performed by the system/object associated with the diagram
 - Two transition types
 - **Internal transitions** to a state that react to an event without changing the current state of the system or object
 - The **transitions between states** or **external transitions**, which express a change in state
 - Example: States of an air conditioner



State diagrams

- Example: Describing the states of an ATM machine



State diagrams

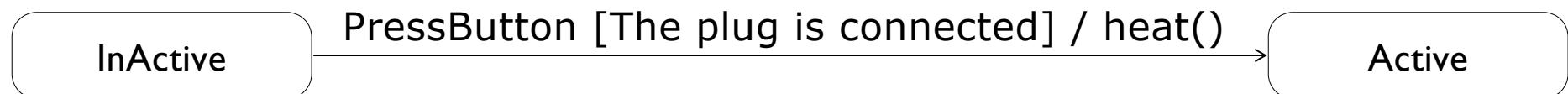
- Event
 - Events of a transition have the following general form

Event [guard] / action

- Event: the event name leading to the transition
- Guard: the condition must be satisfied in order to overcome the transition
- Action: the operation performed when crossing the transition
- Remark: some of these elements may be omitted

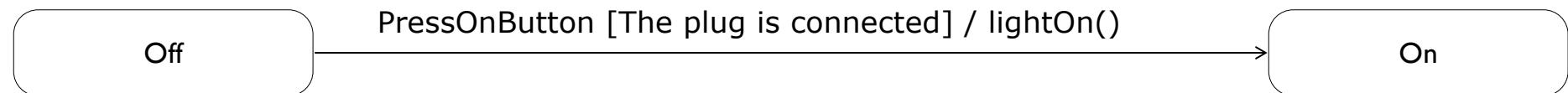
State diagrams

- Event
 - Example: states of a heater



State diagrams

- Example: State of a lightbulb



State diagrams

- Three **special events** associated with state transition
 - **entry**: allows to specify an action to be performed when entering the state
 - **exit**: allows to specify an action to be performed when going out of a state
 - **do**: allows to specify an action to be performed while the system/object is in the state
- Example

TypingPassword

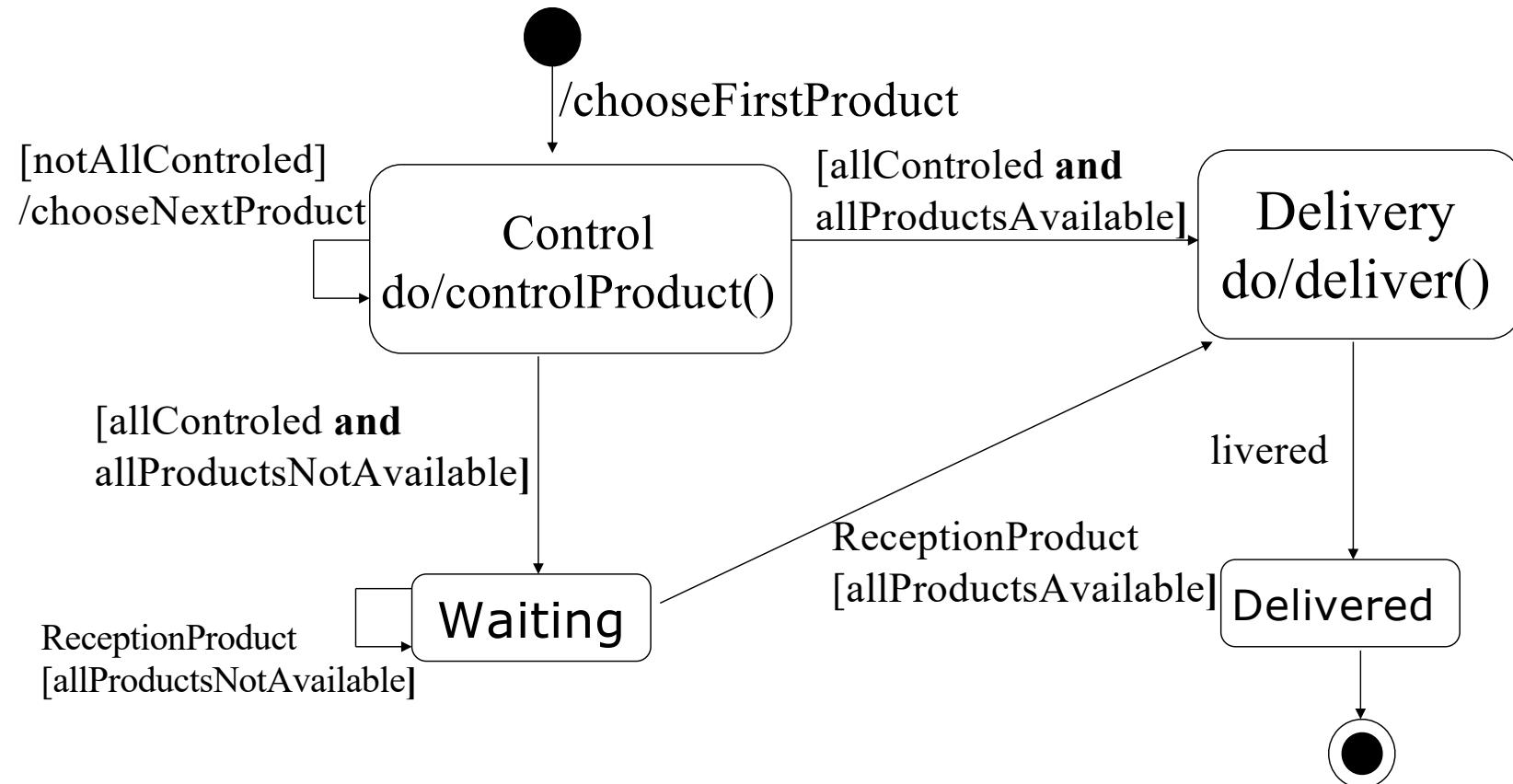
entry / setEchoInvisible
exit / setEchoNormal
do / handleCharacter

ReceivingPhoneCall

entry / pickup
exit / disconnect

State diagrams

- Example
 - Describe the behaviour an “Order”

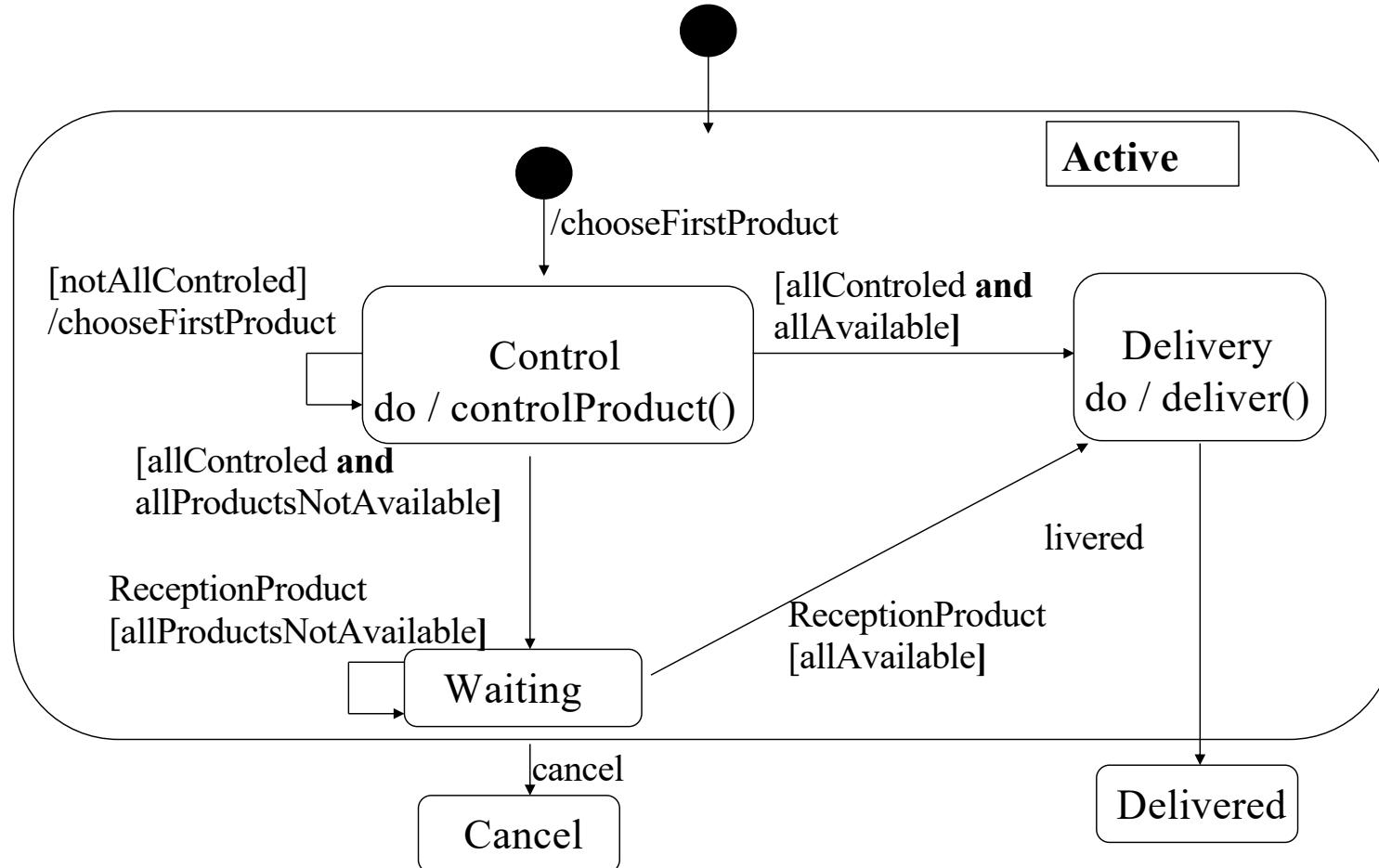


State diagrams

- Composite state
 - Several states and the transitions between these states can be combined into a composite state
 - Principles
 - The composite state has an initial state
 - The **transition to the composite state** is immediately followed by its initial state
 - The **transition from the composite state** may be originated from any of its belonging states

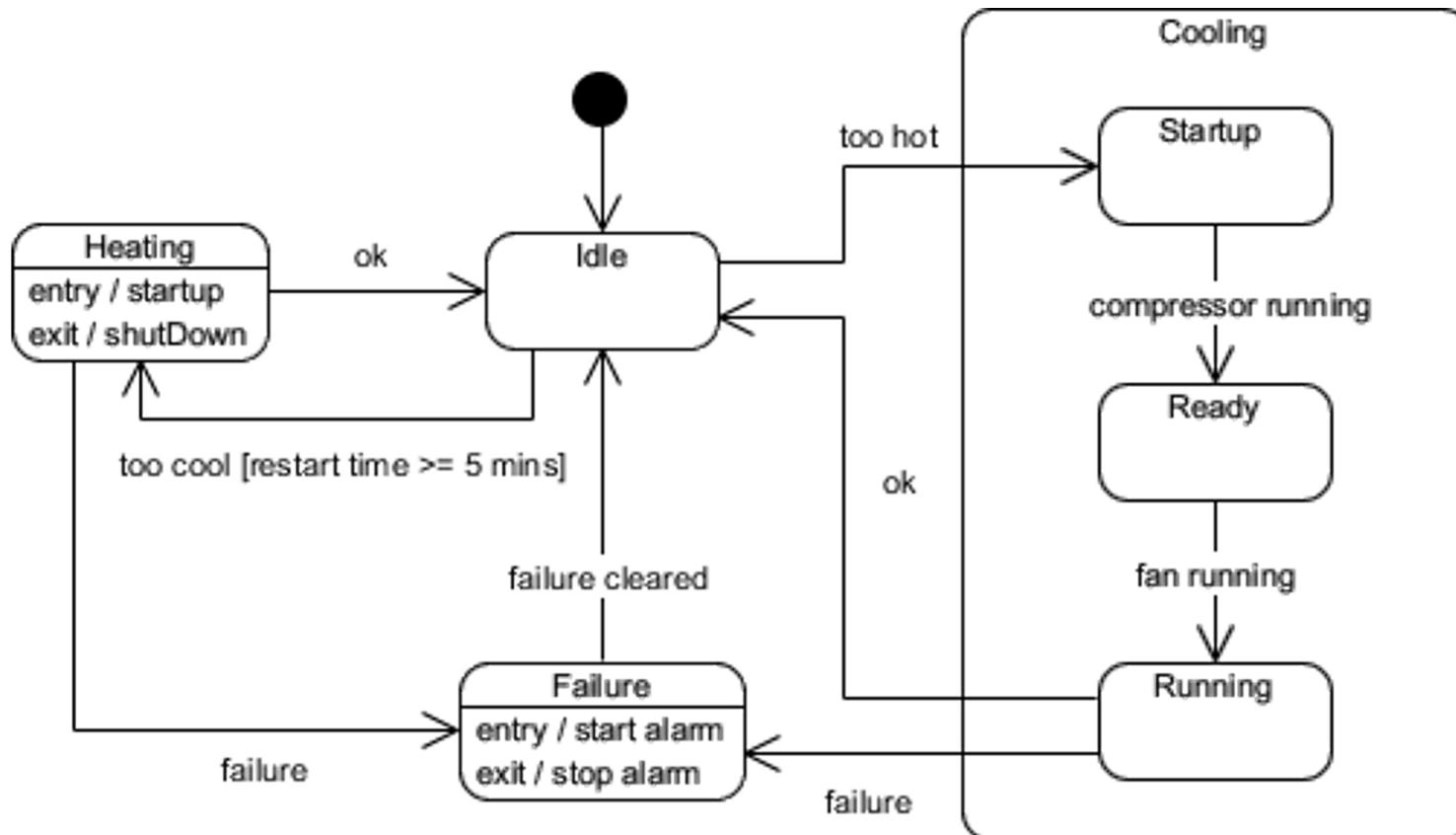
State diagrams

- Example of composite state



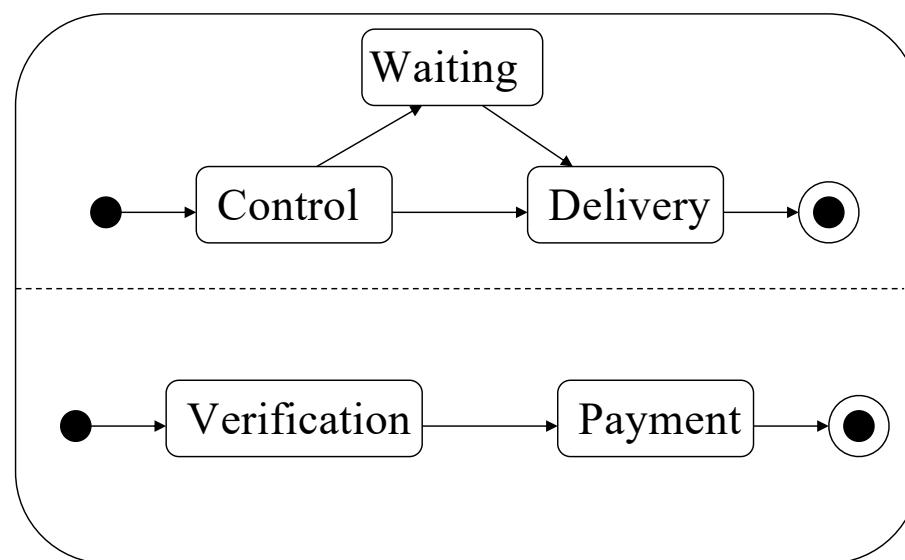
State diagrams

- Example of composite state



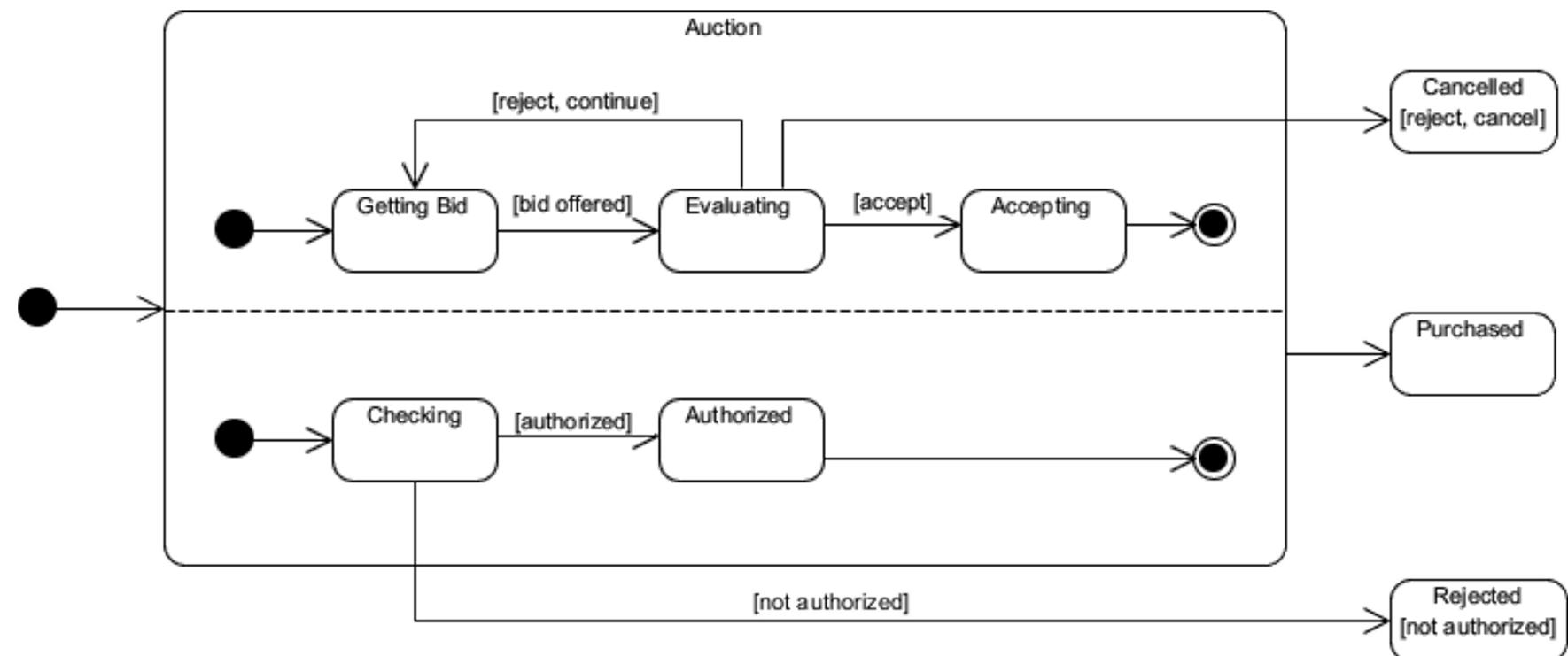
State diagrams

- Parallelism
 - Defining **concurrent state** within a composite state
 - Several states may exist simultaneously within a composite state
 - Example (1)
 - Simultaneous processing of an order and its payment



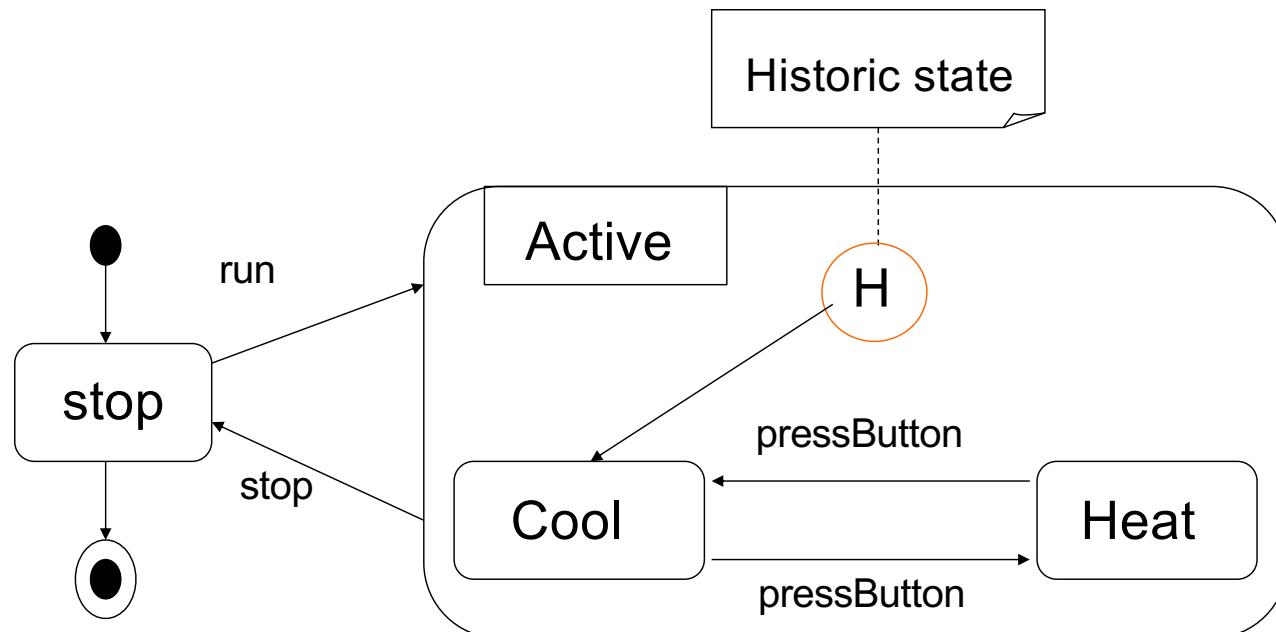
State diagrams

- Parallelism
 - Example (2)
 - Auction Process with two concurrent substates: processing bid and authorizing the payment limit.



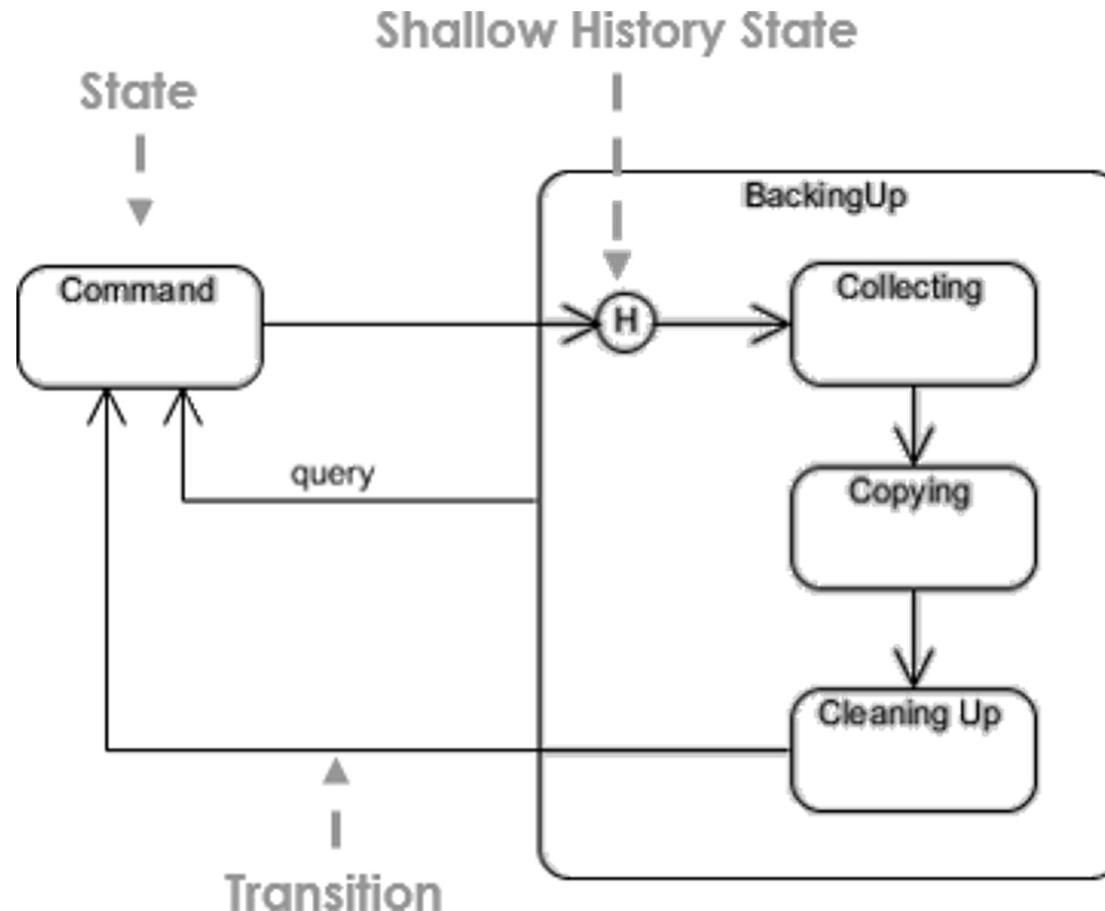
State diagrams

- Historic state
 - Allowing to memorize the current state when exiting a composite state



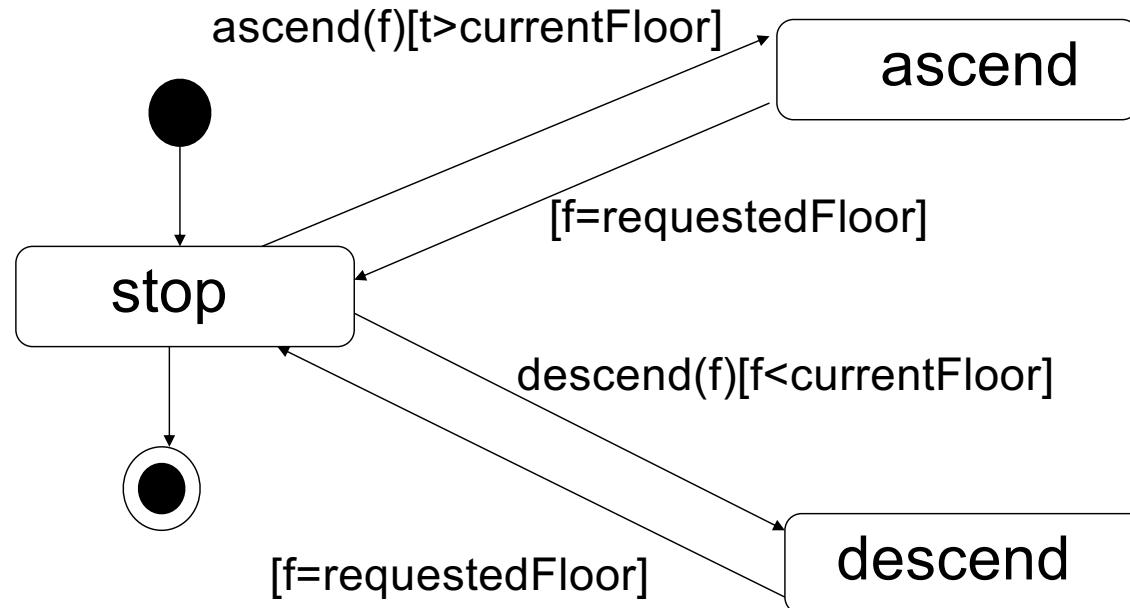
State diagrams

- Historic state
 - Allowing to memorize the current state when exiting a composite state



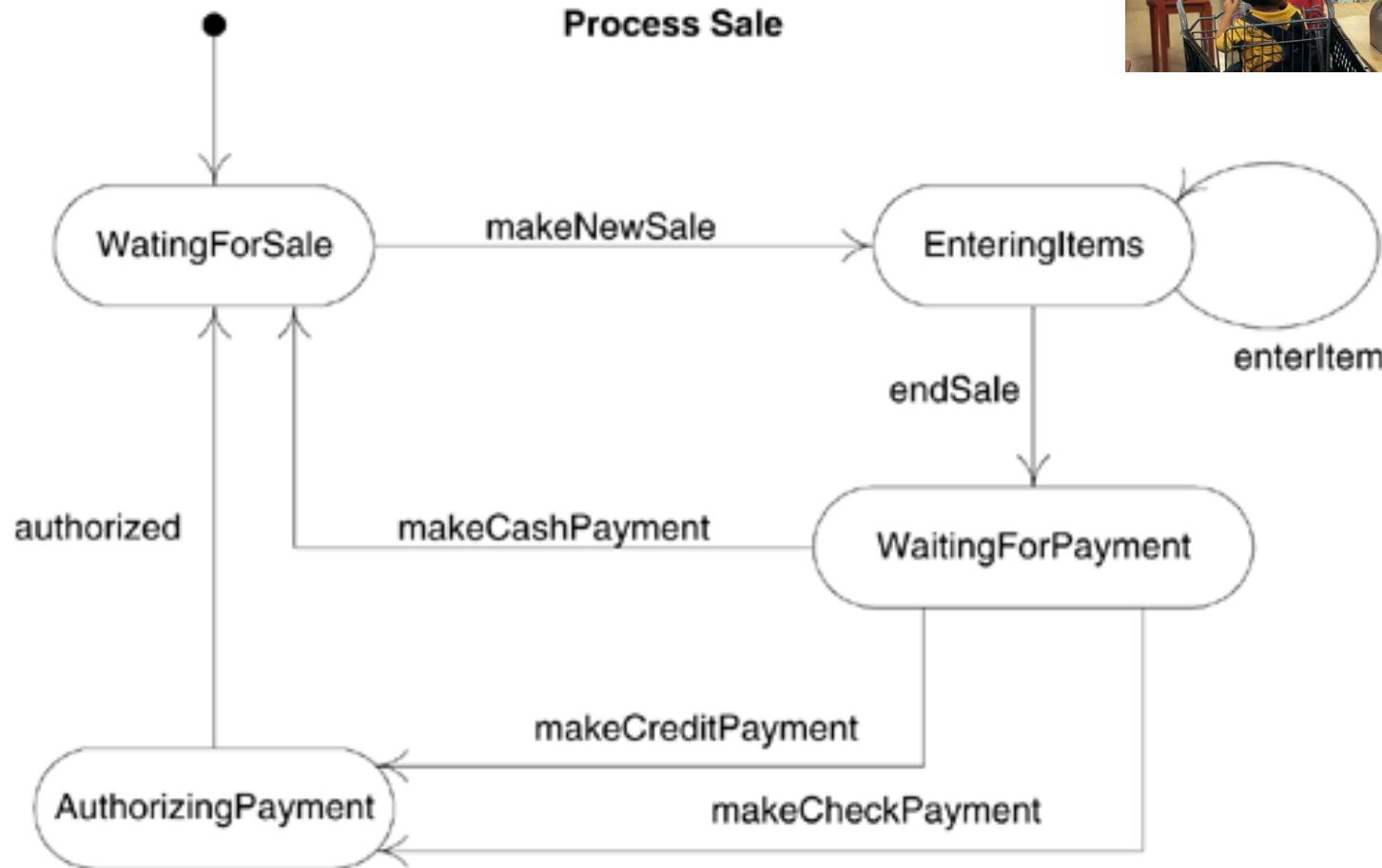
State diagrams

- Example
 - Modelling an elevator's states



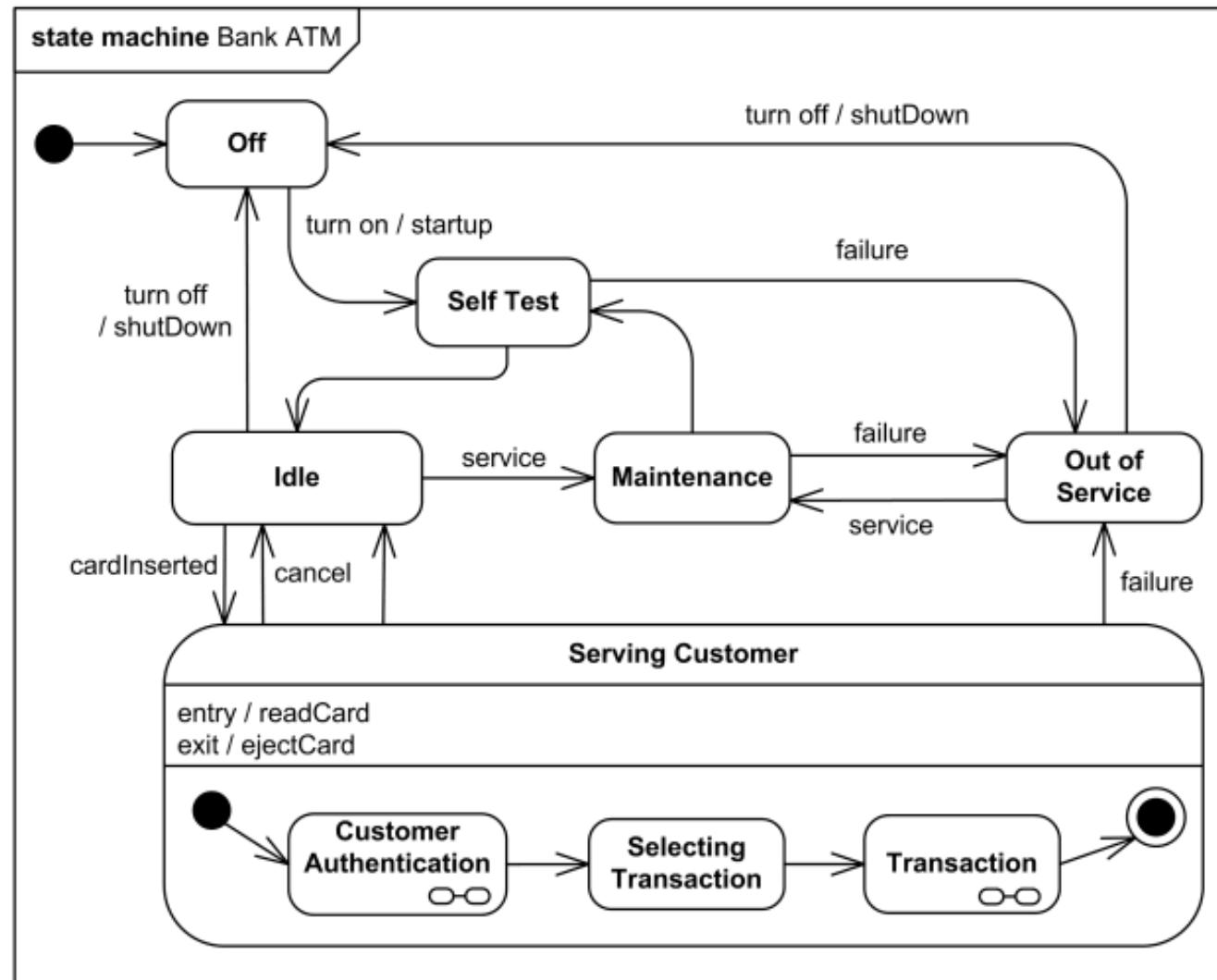
State diagram

- Example
 - Modelling the cash register system



State diagram

- Example
 - Modelling the Bank ATM



Interaction diagrams

- Interaction diagrams are used to model the dynamic aspects of the system
 - An **interaction diagram is associated with a task** performed by the system or its components
 - Interaction diagrams are determined/built based on activity diagrams and use-case diagrams
 - An interaction diagram generally corresponds to **a use-case** or a functionality
 - The interaction diagram shows how objects and actors communicate together to achieve the task
- Specifically, an interaction diagram allows to describe in detail **the algorithms** in the system
- Interaction diagrams can be subsequently used in **the implementation of class methods**

Interaction diagrams

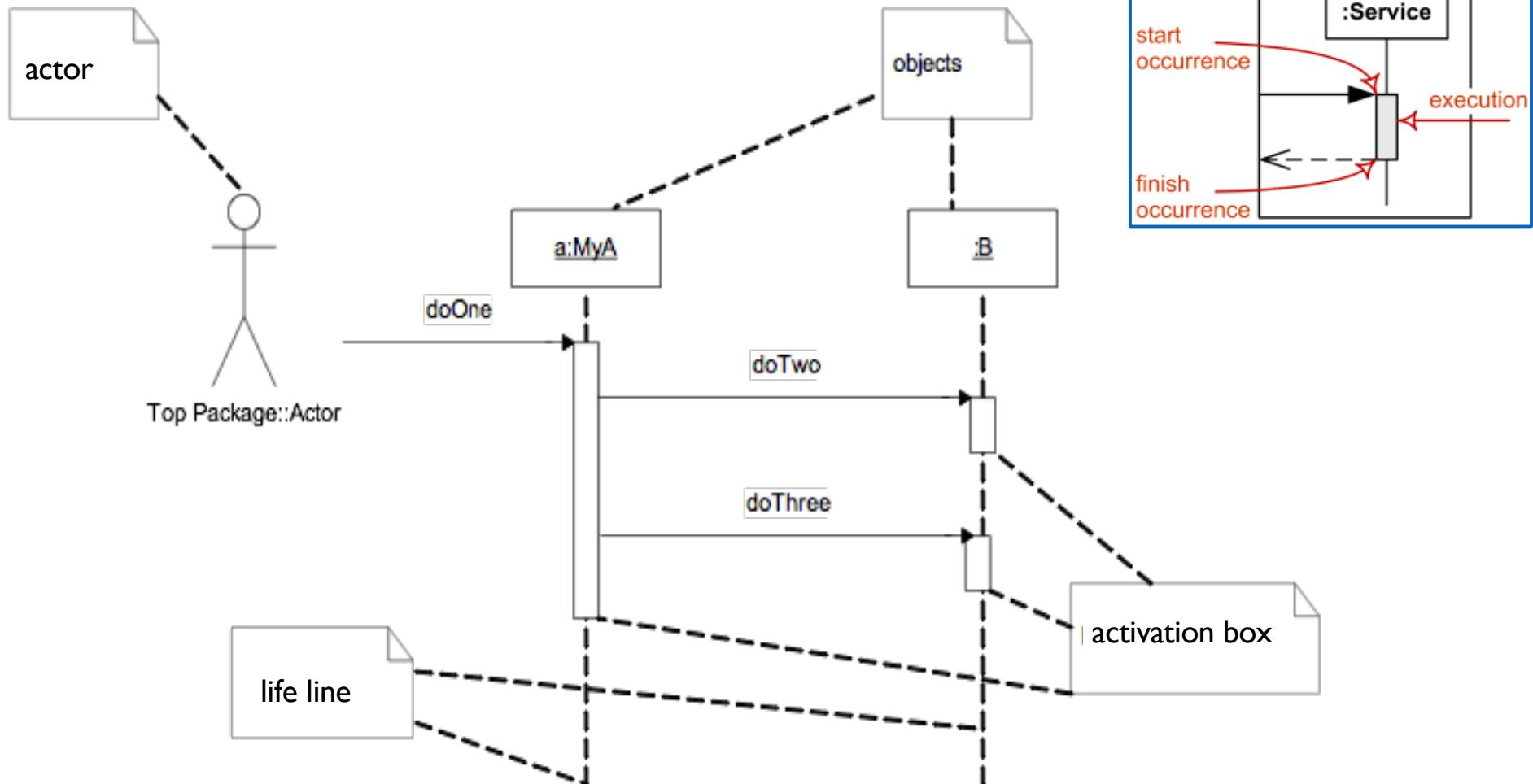
- The essential elements of an interaction diagrams
 - Objects
 - Actors
 - Messages
- Actions between objects and actors are
 - message sendings
 - object creations and destructions
- **Two types of interaction diagrams**
 - **Sequence diagrams**
 - The temporal sequence of interactions
 - **Collaboration diagrams**
 - An instance of class diagram

Sequence diagram

- A **sequence diagram** describes the **temporal sequence** of exchanges of messages between objects and the actor to perform a certain task
 - The **actor** who initiates interactions is usually found on the far left
 - The **objects** are placed horizontally on the diagram
 - The vertical dimension represents time
 - Each object or actor is associated with a **life line** representing the time where the object or actor is
 - An **activation box** represents the object activation period

Sequence diagram

□ Notation

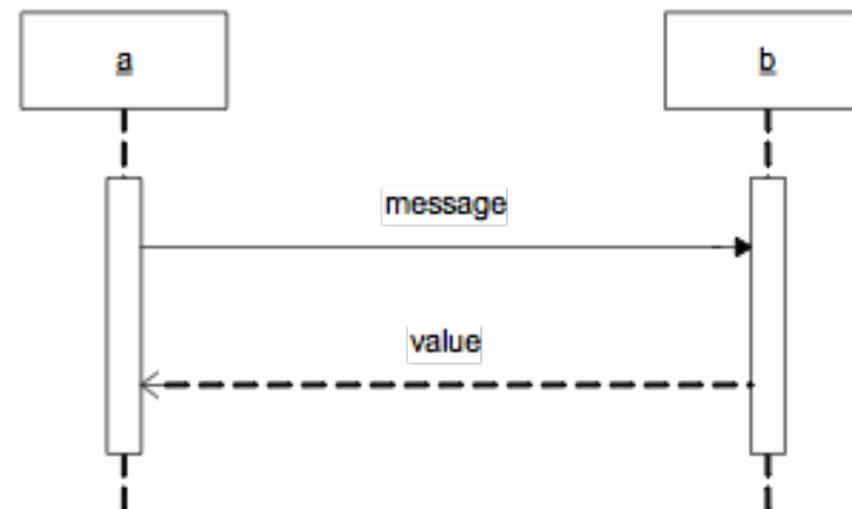


Sequence diagram

- Messages
 - Message is the medium of communication between objects
 - The general form of message
 - [**guard**] **message**(**parameters**)
- **guard**: a condition must be satisfied in order to send the message
- **message**: the identifier of the sent message
- **parameters**: a list of parameter values
- Note: guard and parameters can be omitted

Sequence diagram

- The return values
 - Sending a message to an object cause **the execution of a method** of this object
 - This method can optionally return a value
 - The return values may be omitted or be explicitly described
 - either as the following form
[guard]value := message(parameters)
 - or by a return message that represents graphically

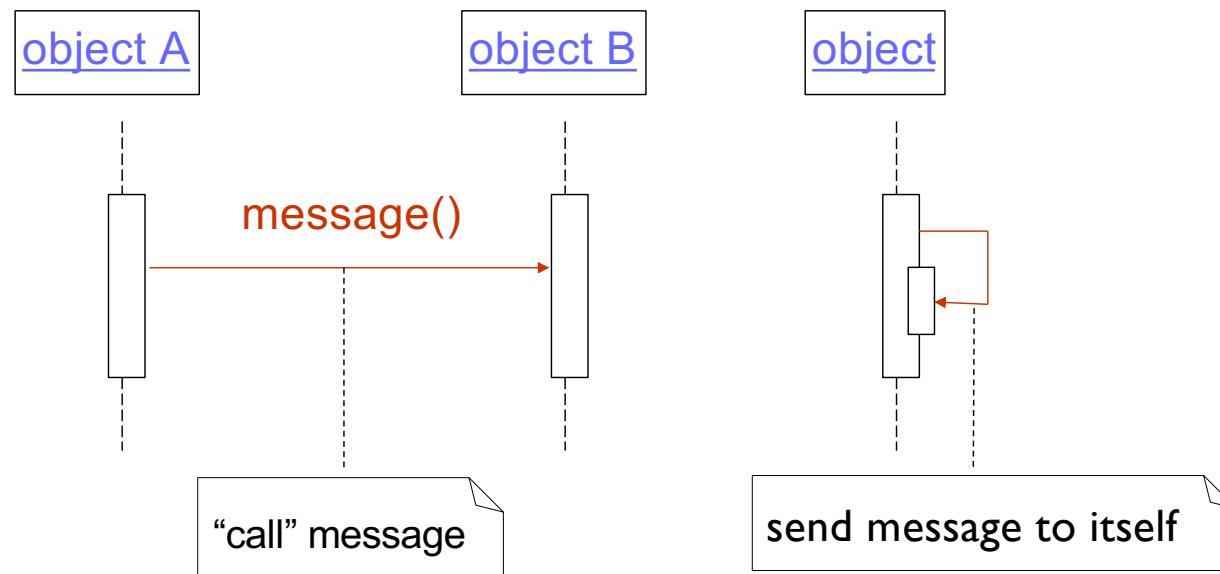


Sequence diagram

- Types of message
 - “call” message
 - “return” message
 - “send” message gửi đi nhưng không chờ giá trị trả về
 - “create” message
 - “destroy” message

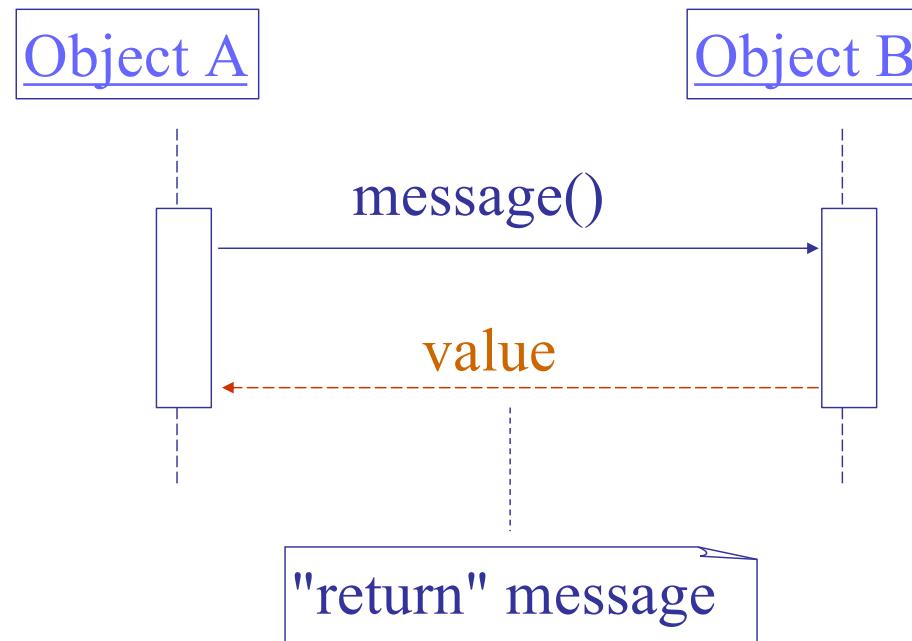
Sequence diagram

- “call” message
 - A “call” message invokes an operation/method of the object
 - A “call” message is a **synchronous message**: the object that sends the message must wait for the termination of the execution of the message before doing other tasks
 - An object can send message to itself
 - Notation



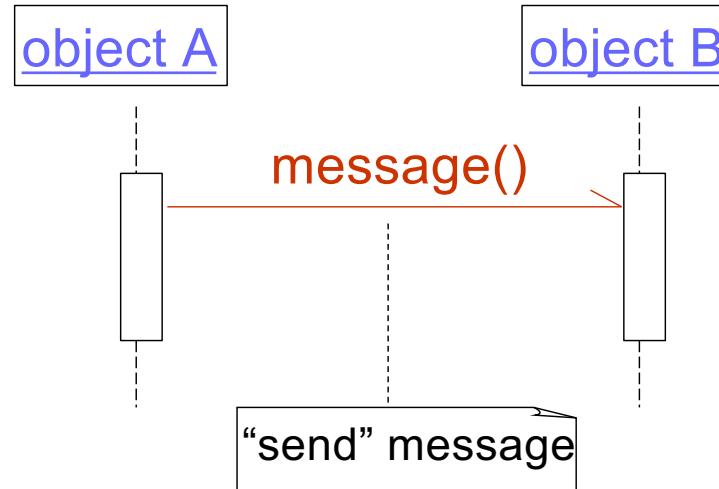
Sequence diagram

- The “return” message returns a value for the calling object
- Notation



Sequence diagram

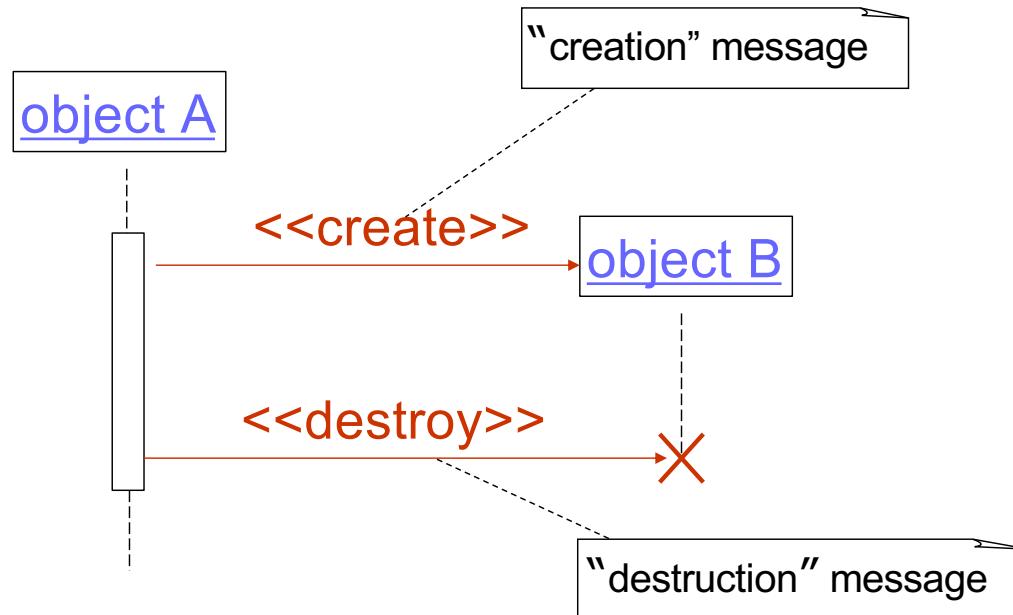
- “send” message
 - A “send” message sends a signal to an object
 - A “send” message is an **asynchronous message**: once the object sends the message, it expects nothing and continues to do other tasks
 - Notation



- Asynchronous message is often used in multi-threaded environment
 - For example, *Thread.start()*, *Runnable.run()* in Java

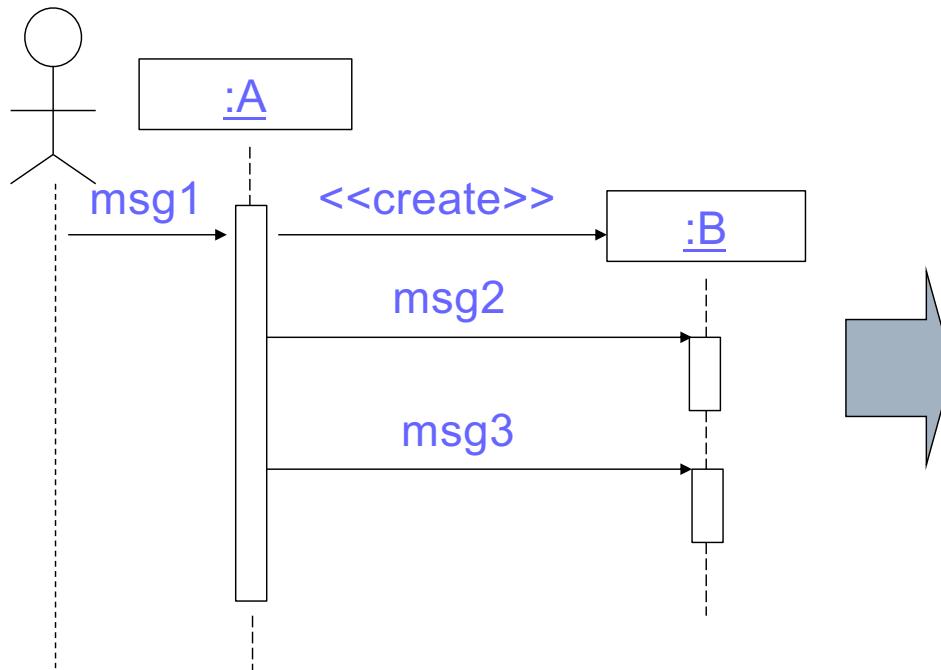
Sequence diagram

- “creation” message
 - invokes the creation method of object (constructor)
- “destruction” message
 - invokes the destruction message of message (destructor)
- Notation



Sequence diagram

- Example
 - The sequence diagram and the corresponding code

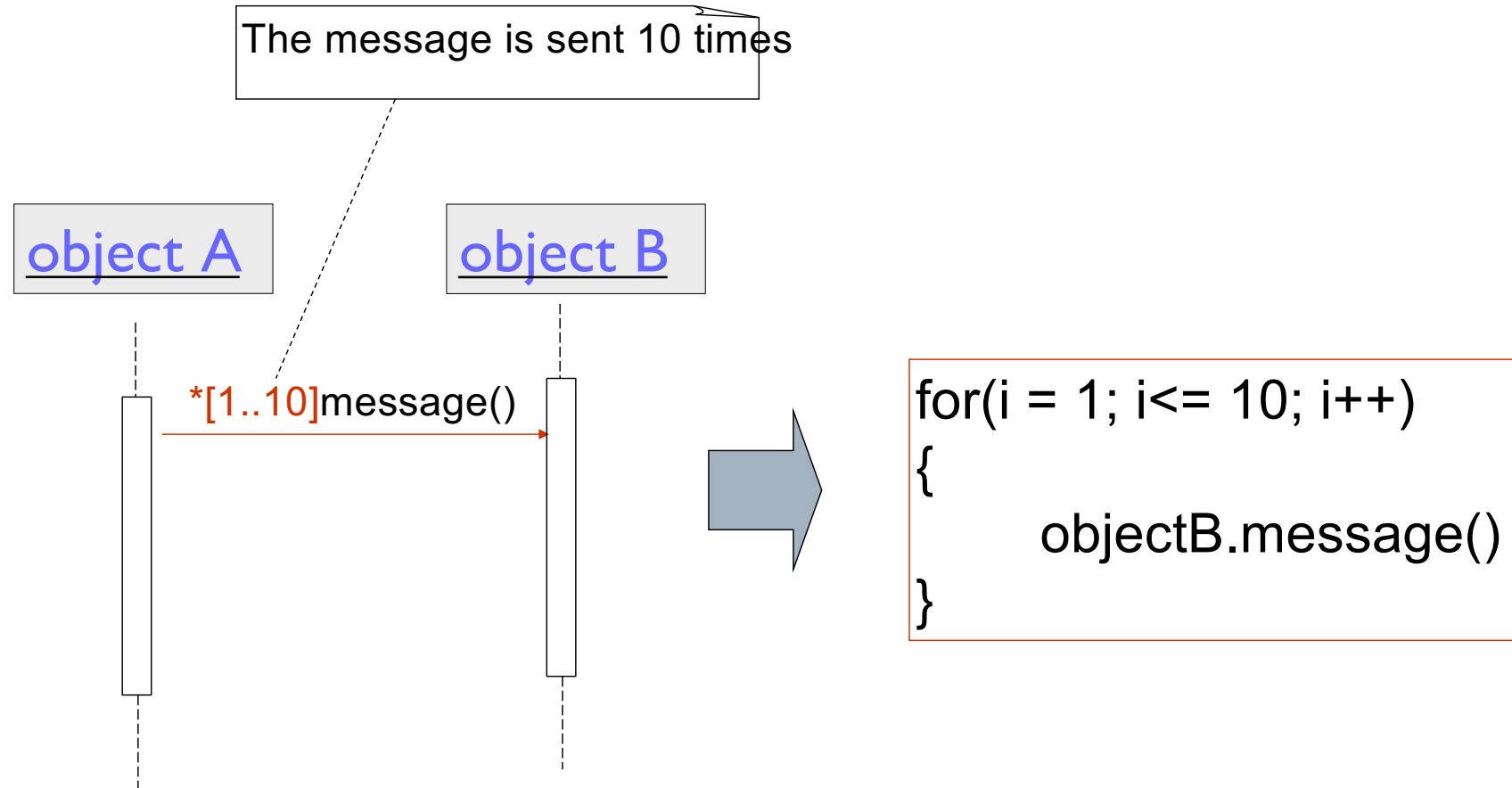


```
public class A
{
    private B objB;
    public void msg1()
    {
        objB = new B();
        objB.msg2();
        objB.msg3();
    }
}

public class B
{
    ...
    public void msg2() { ... }
    public void msg3() { ... }
}
```

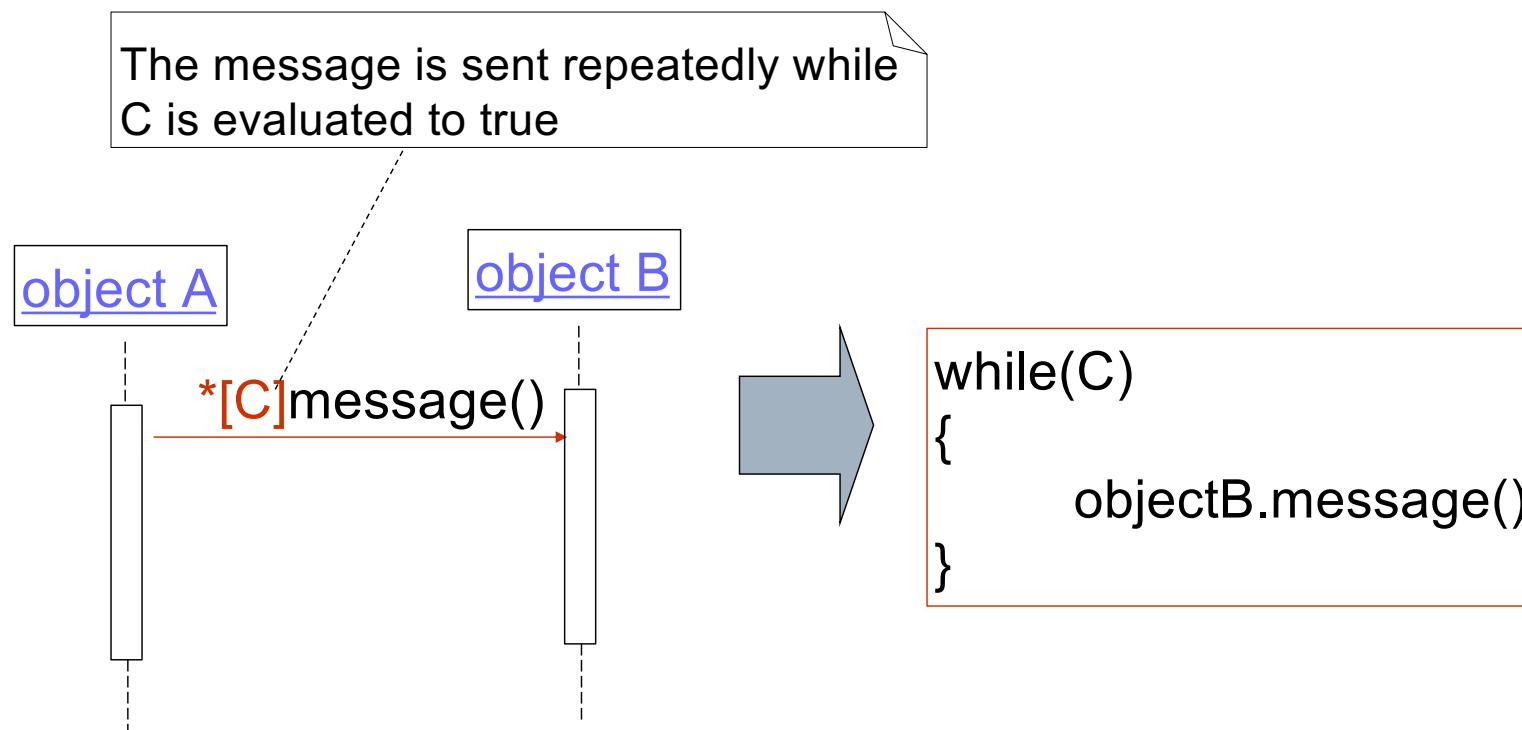
Sequence diagram

- A message can be **sent iteratively**
- Example



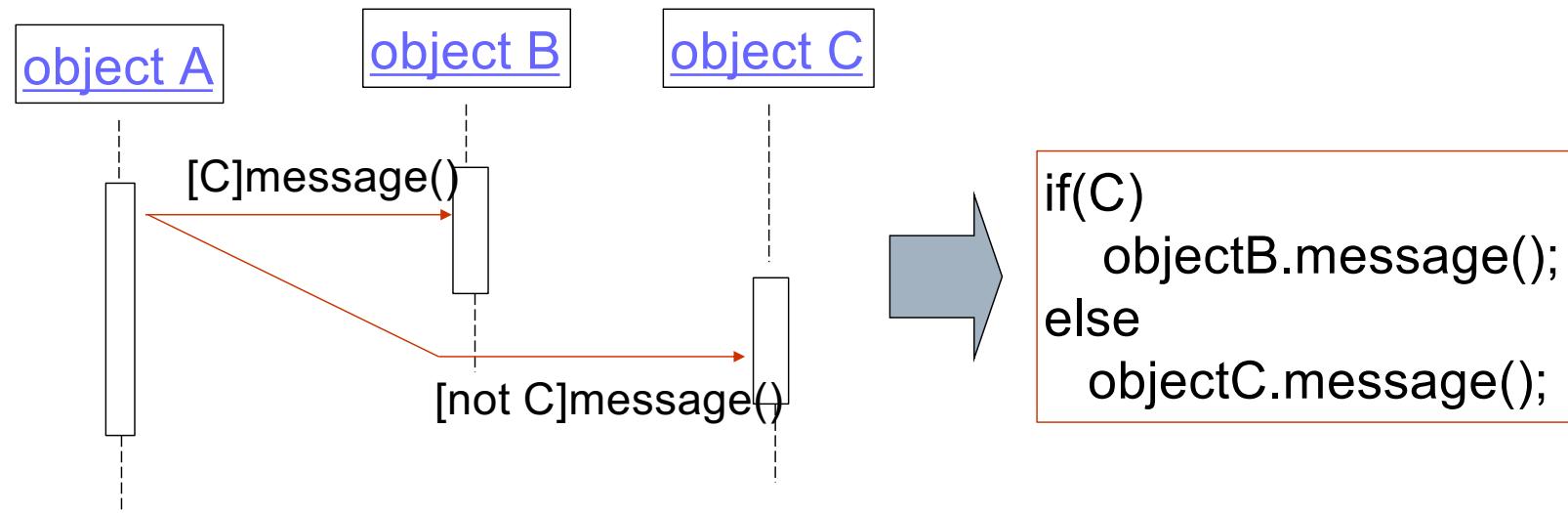
Sequence diagram

- A message can be sent iteratively based on a condition
- Example



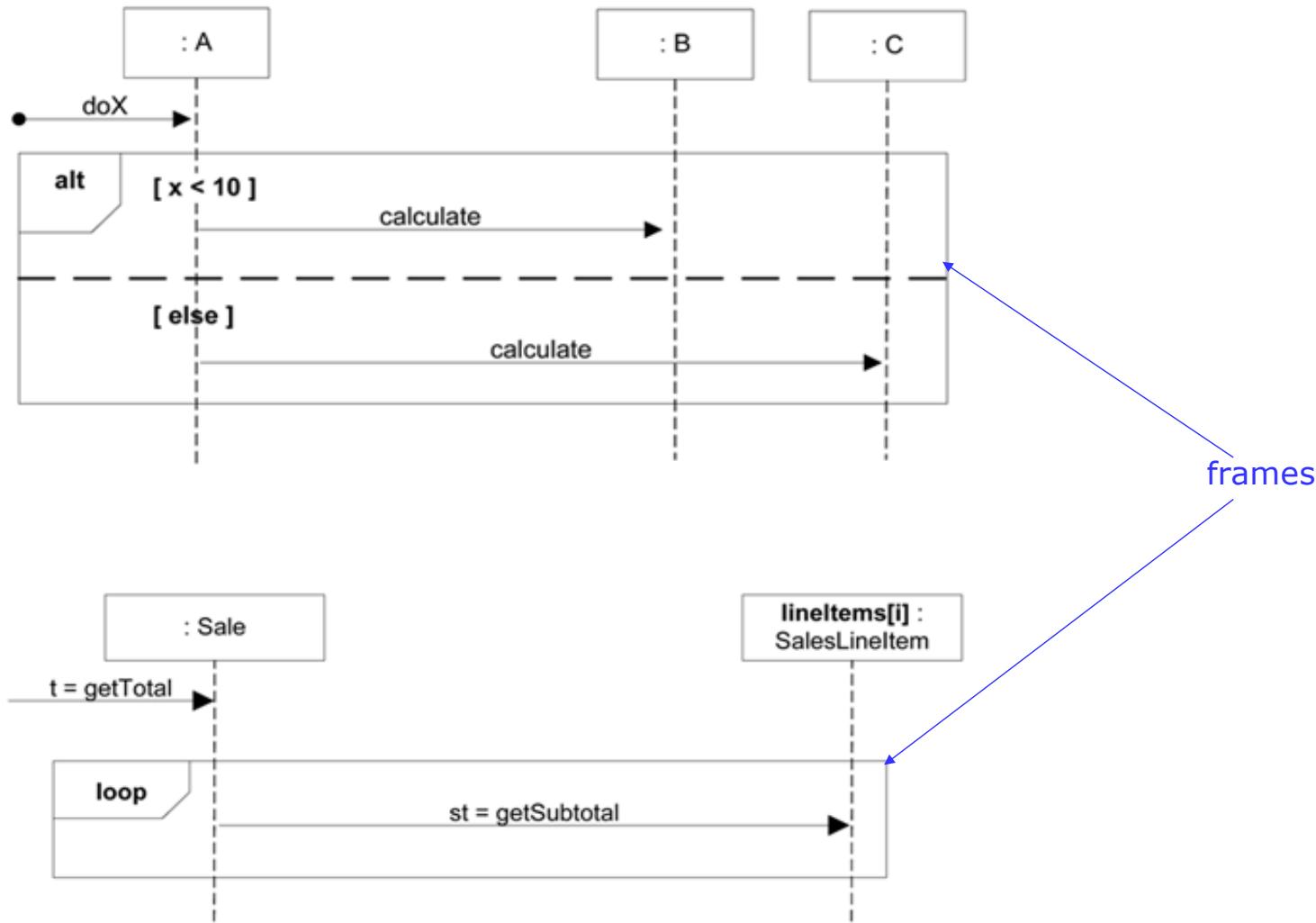
Sequence diagram

- The sending of a message can depend on a **decision**
- Example



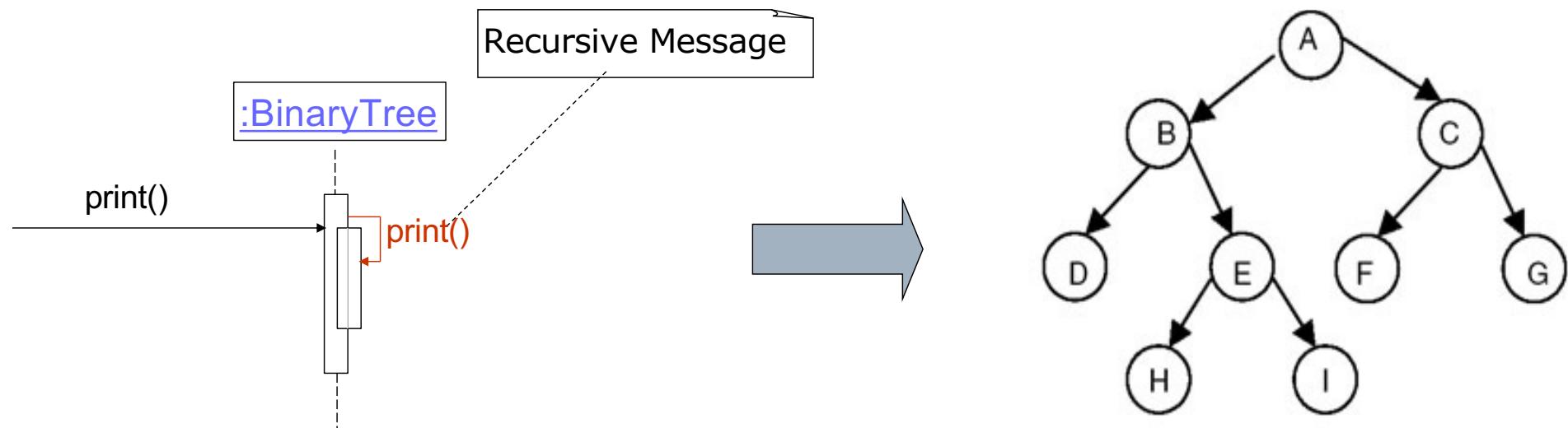
Sequence diagram

- Note: UML 2.x notations allow the use of frames to represent the conditions or iterations



Sequence diagram

- A message can be called **recursively**
- Notation



Inorder : DBHEIAFCG

Preorder : ABDEHICFG

Postorder : DHIEBFGCA

Sequence diagram

Modelling a polymorphic message

Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation

Payment

Pay by Credit or Debit card:      

Card Number: Please enter a valid card number

Card Type: Please select a card type

Expiry Date: Please select an expiry date

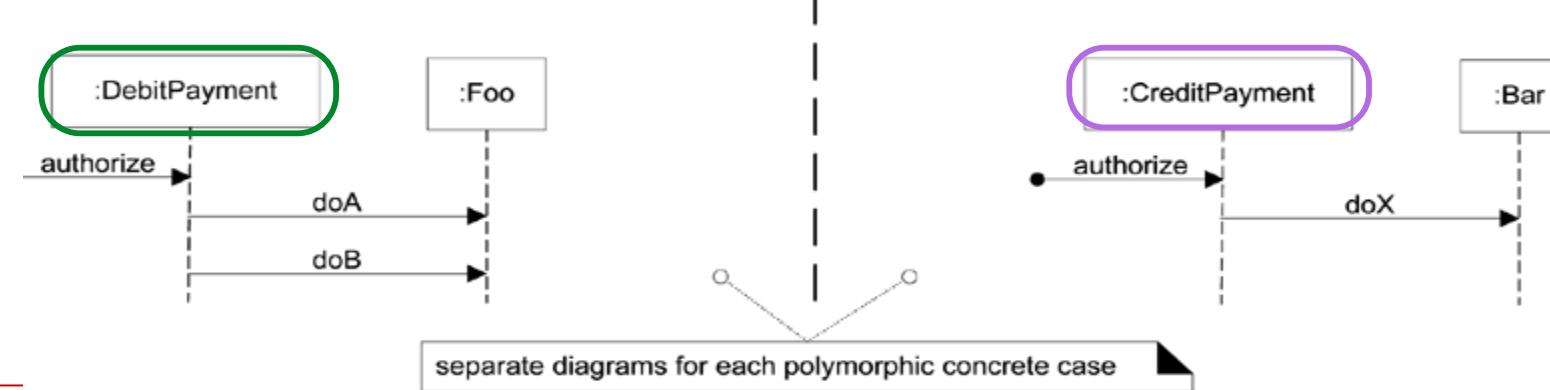
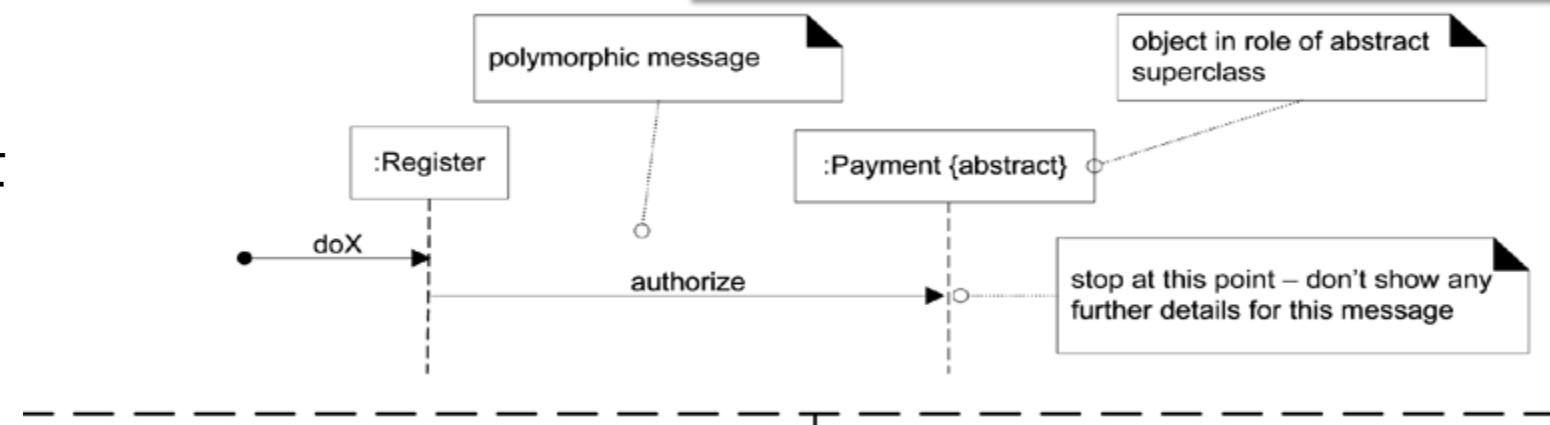
Security Code (CVV): [What is this?](#) Please enter a valid numeric security code (CVV)

Cardholder's Name: Please enter a valid cardholder's name

Postcode/Zip Code:

Email: Please enter a valid email

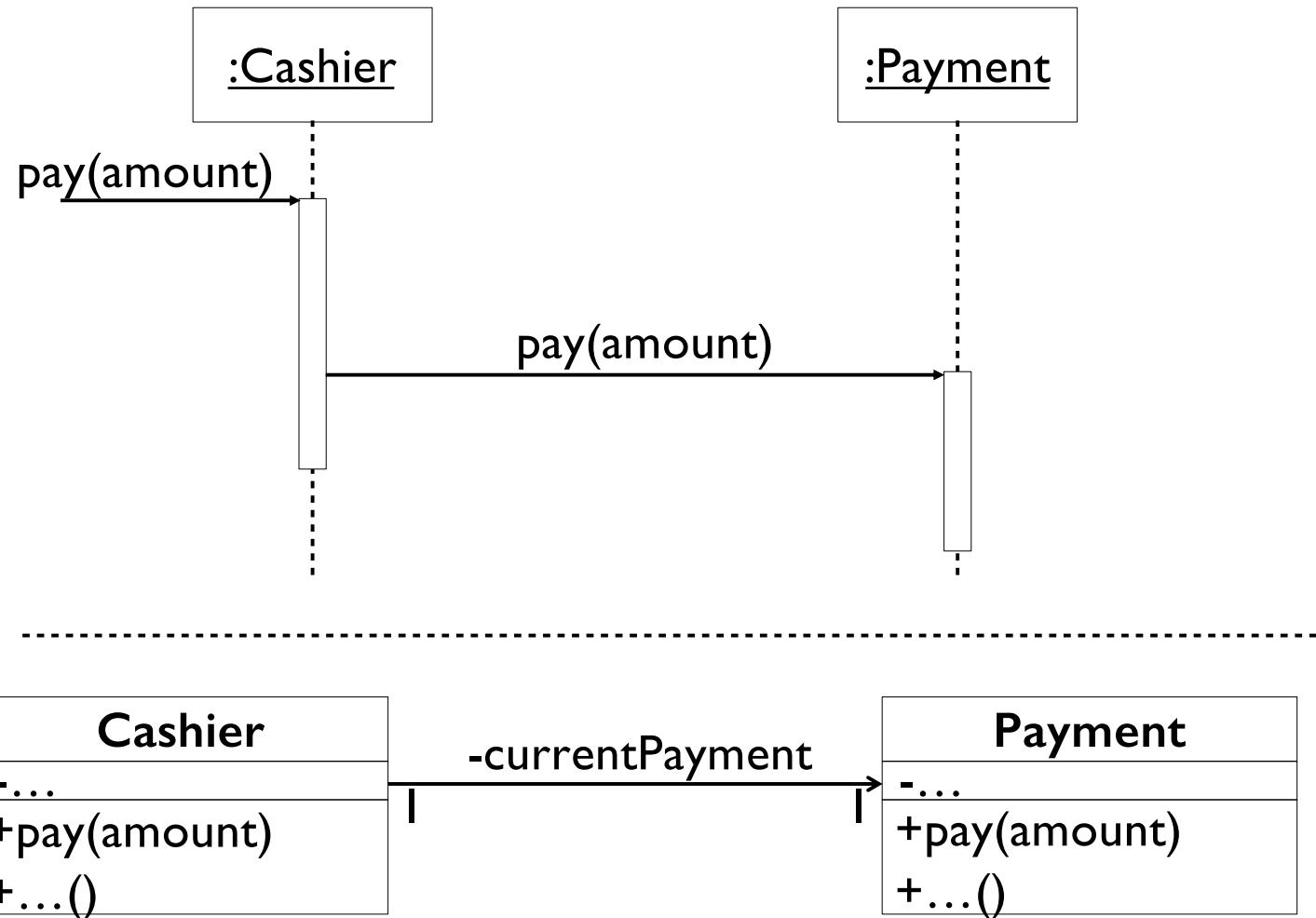
Save my card details for next time



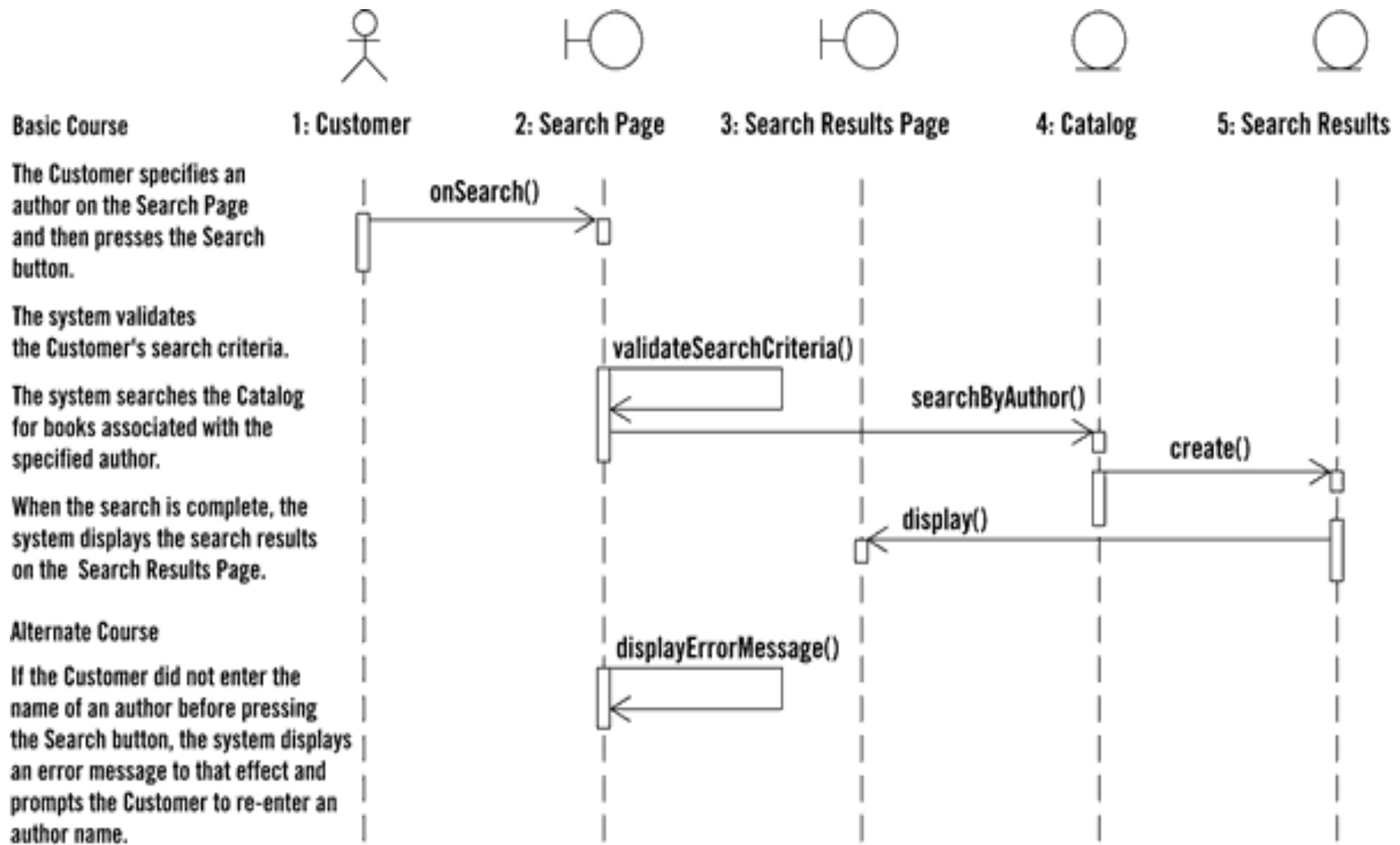
chia sơ đồ thành
2 phần để mô tả
cụ thể hơn

Sequence diagram

- Relationship between class diagram and sequence diagram



Sequence diagram from use-case



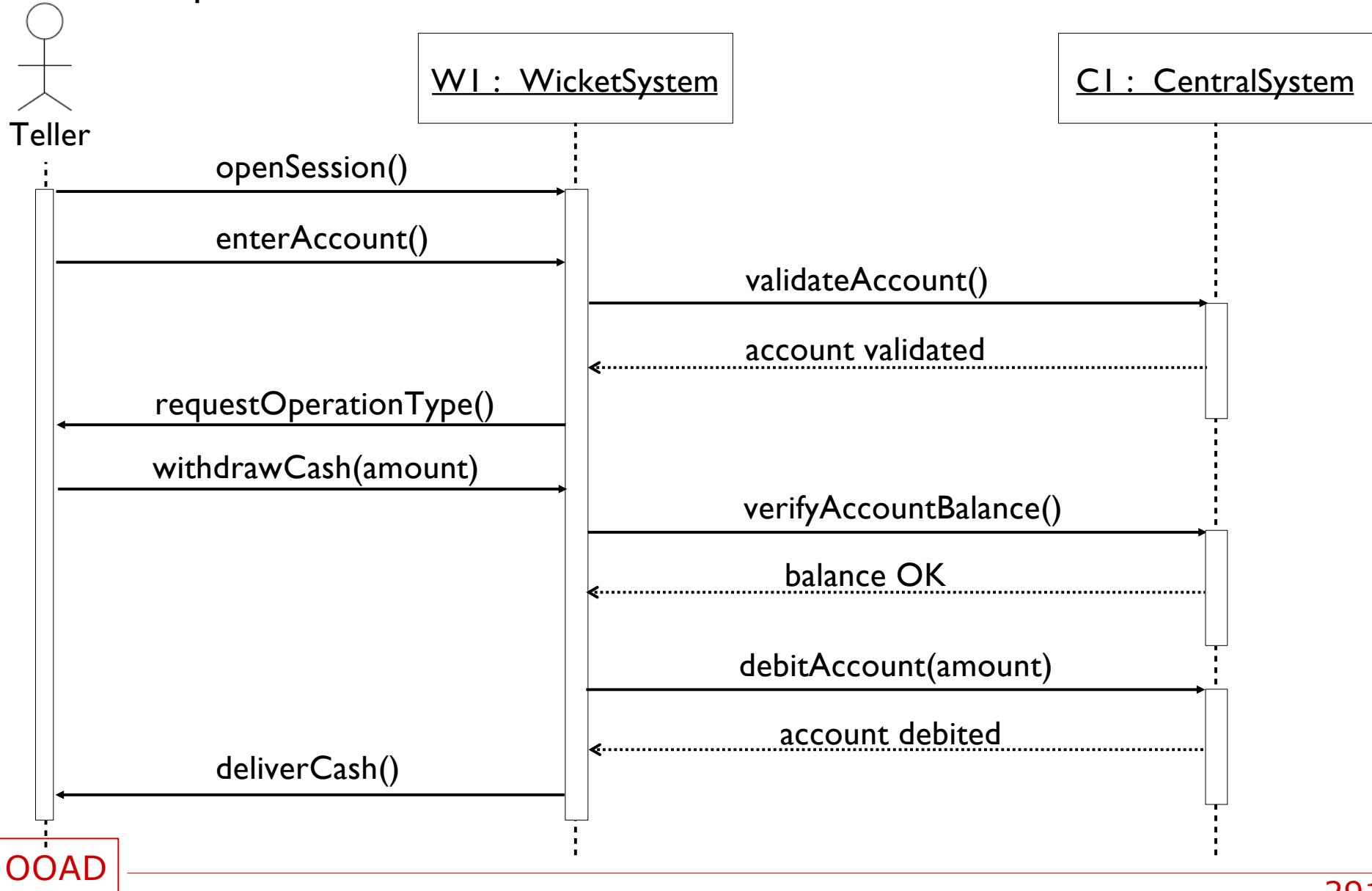
Sequence diagram

- Example: Cash withdrawal at the bank



Sequence diagram

- Example: Cash withdrawal at the bank



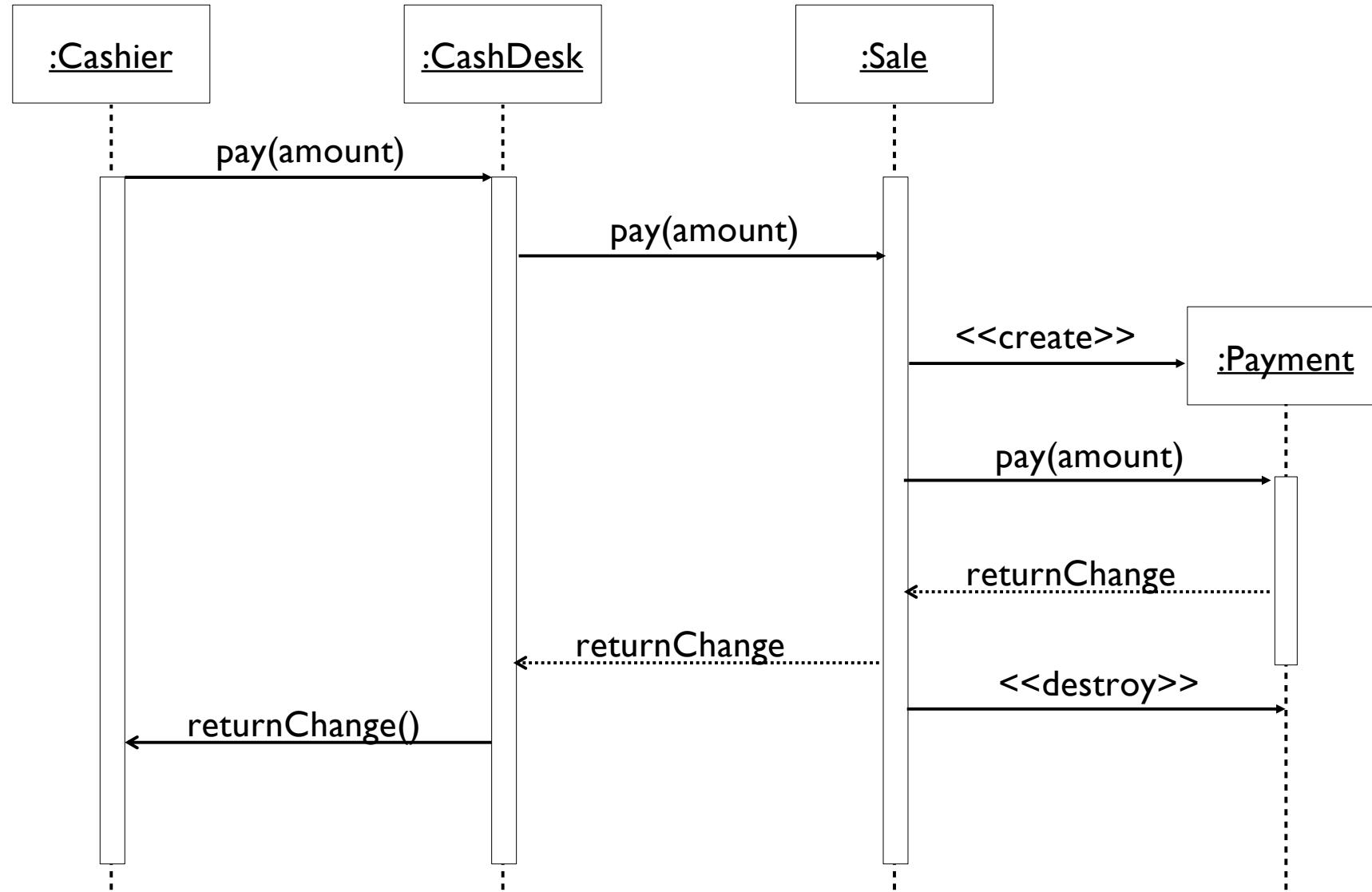
Sequence diagram

- Example: Use-case “cash payment”



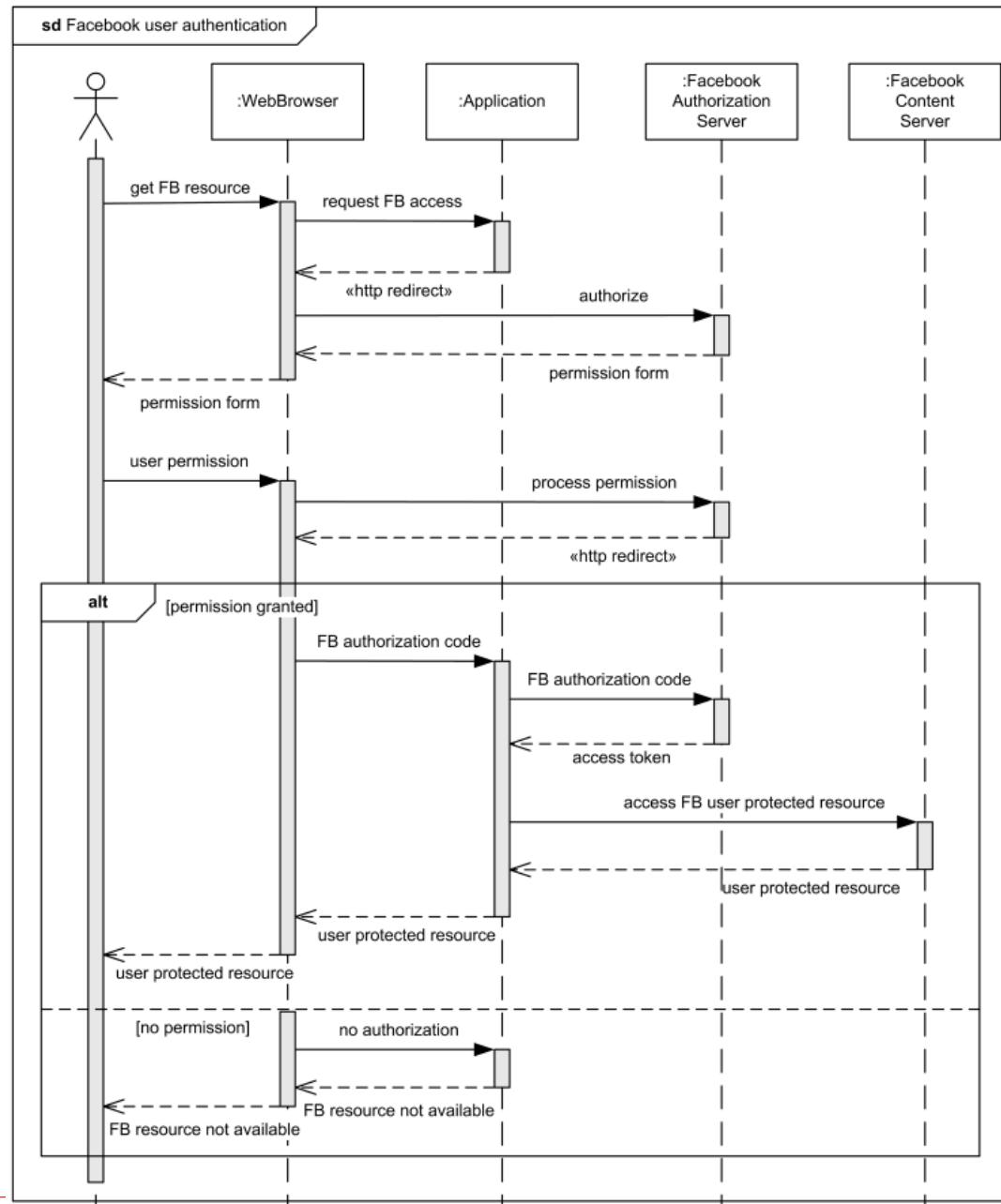
Sequence diagram

- Example: Use-case “cash payment”



Sequence diagram

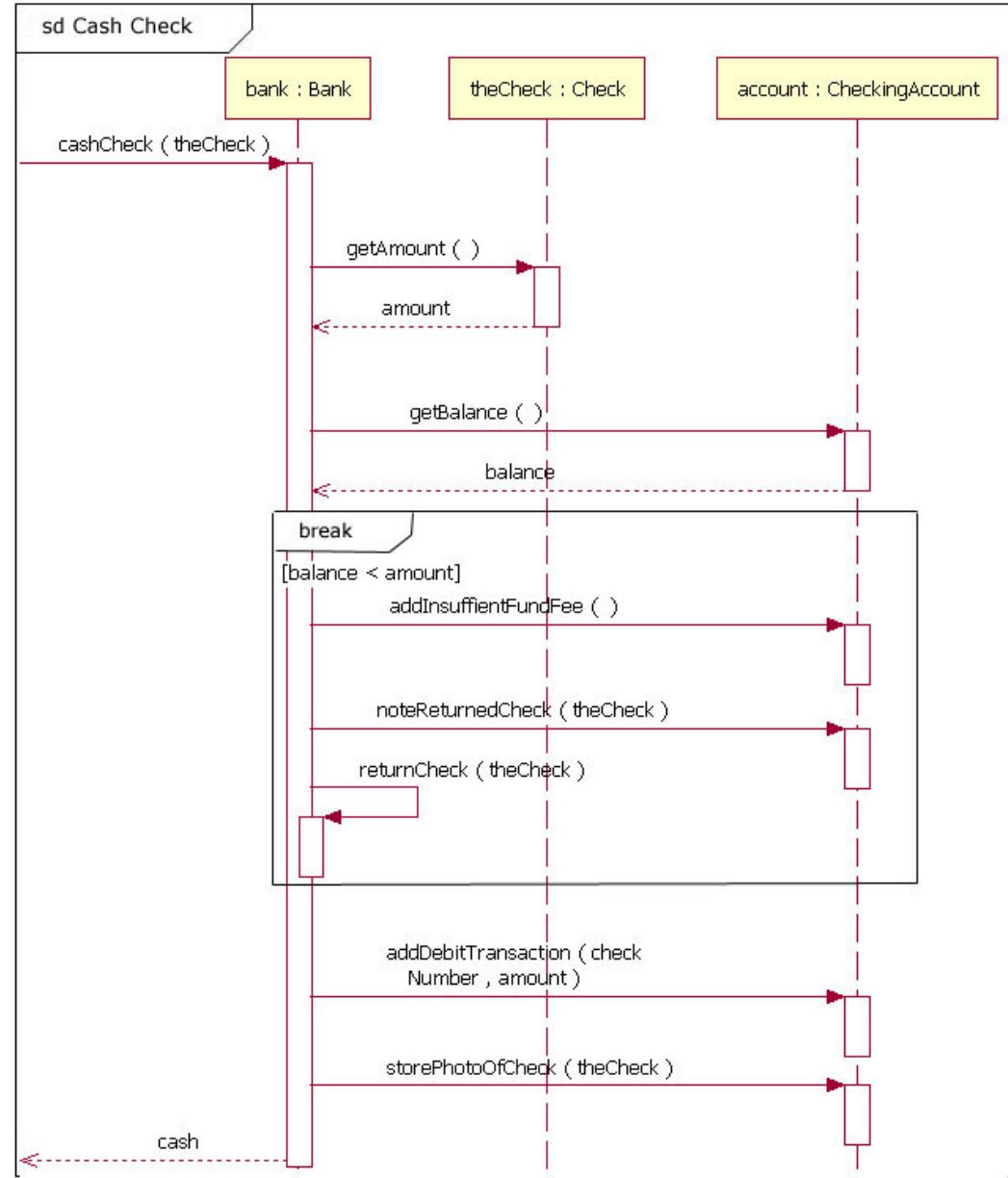
- Example:
Facebook
Web
User
Authentication



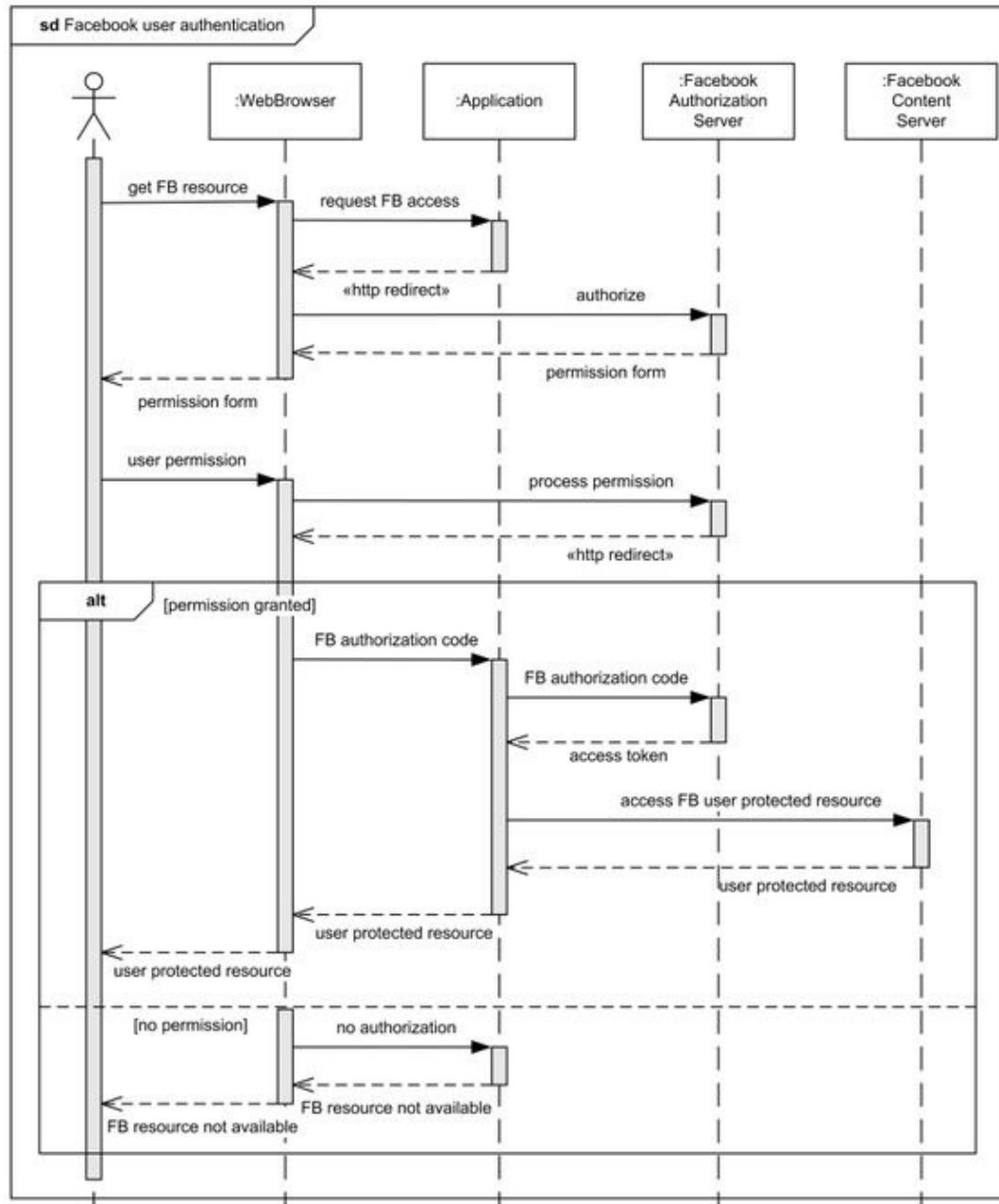
OOAD

Sequence diagram

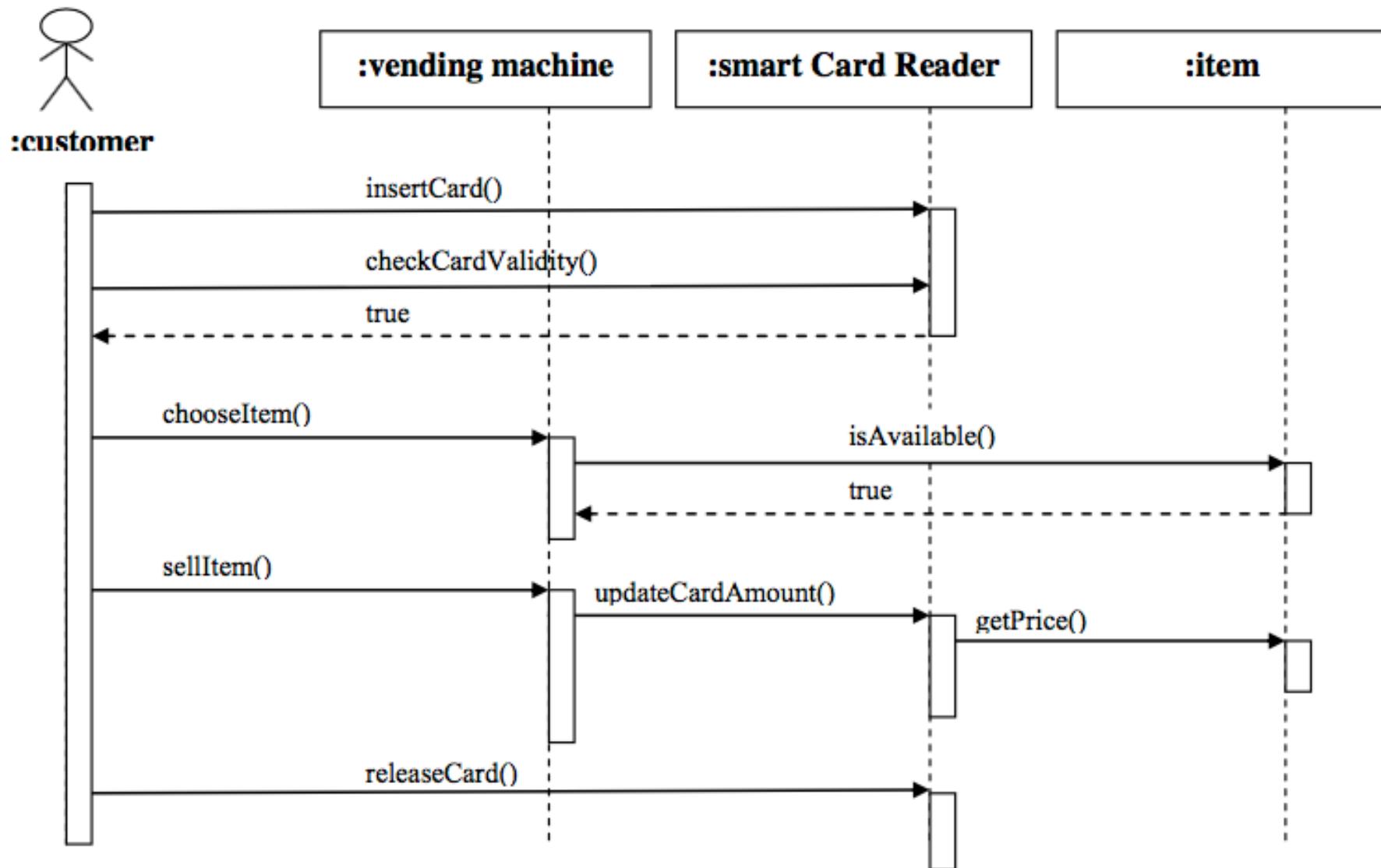
- Example:
Cash Check

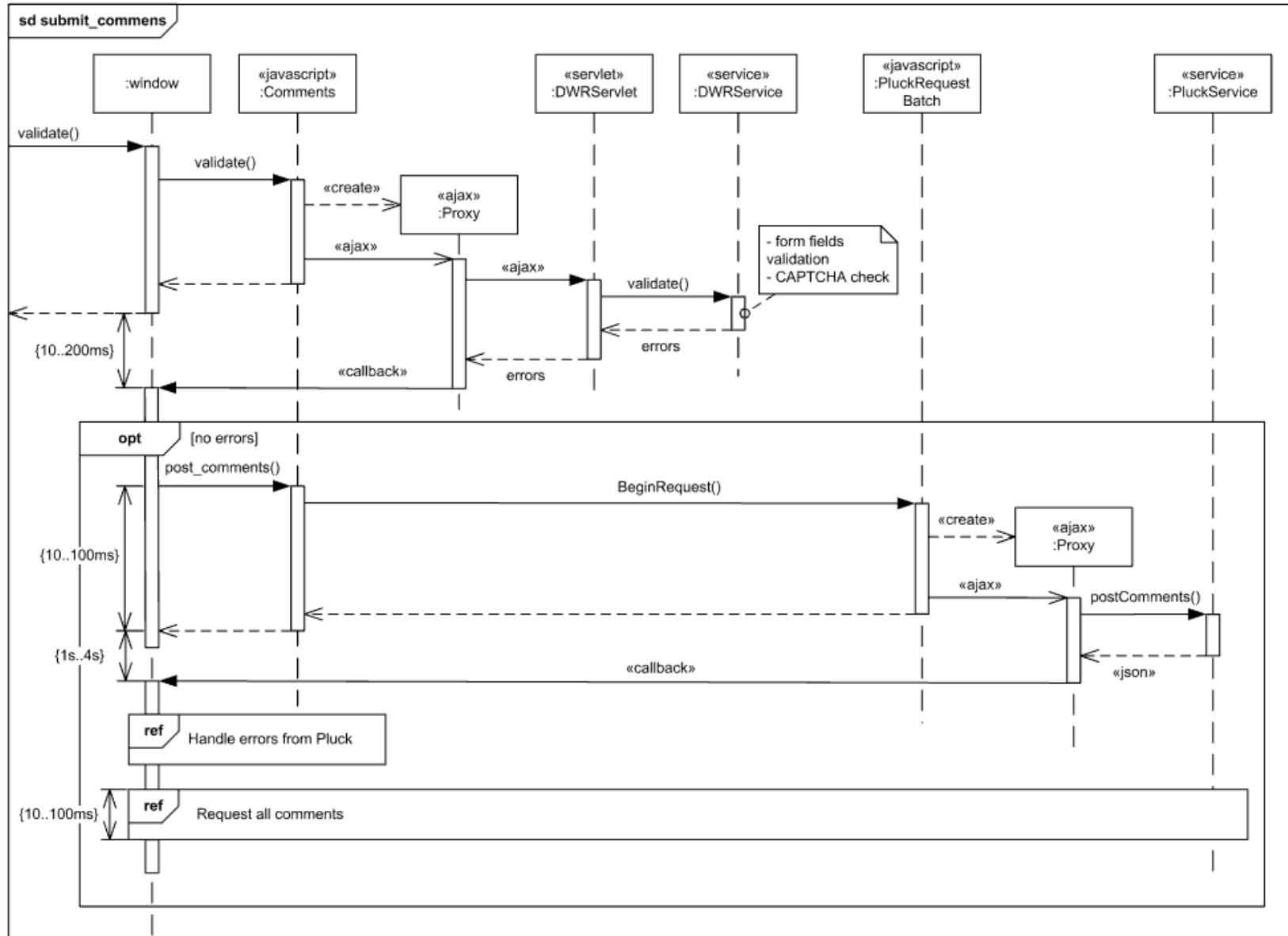


OOAD



OOAD





OOAD

Why not just code it?

- Sequence diagrams can be somewhat close to the code level.
So why not just code that algorithm rather than drawing it as a sequence diagram?
 - a good sequence diagram is still a bit above the level of the real code (not EVERY line of code is drawn on the diagram)
 - sequence diagrams are language-agnostic (can be implemented in many different languages)
 - non-coders can do sequence diagrams
 - easier to do sequence diagrams as a team
 - can see many objects/classes at a time on same page (visual bandwidth)

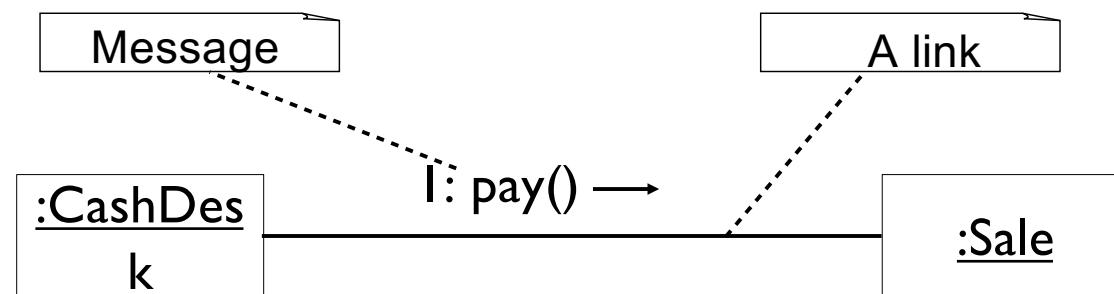
Collaboration/Communication diagram

- A collaboration diagram describes the interaction between objects
 - A collaboration diagram is a graph whose
 - nodes represent object
 - edges represent the communication between objects
 - The temporal ordering of messages is represented by a **numbering** of messages
 - Collaboration diagram is an extension of class diagram

Collaboration diagram

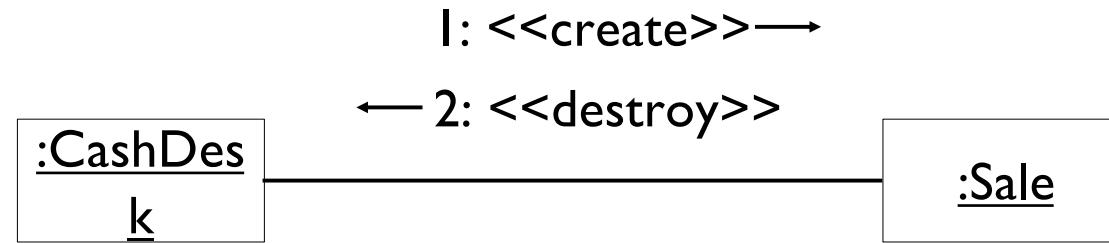
- Links
 - A link shows the sending of a message from an object to another object
 - Formally, a link is an instance of an association

- Messages
 - Each message between objects is presented by an expression of message and an arrow showing the direction of the message

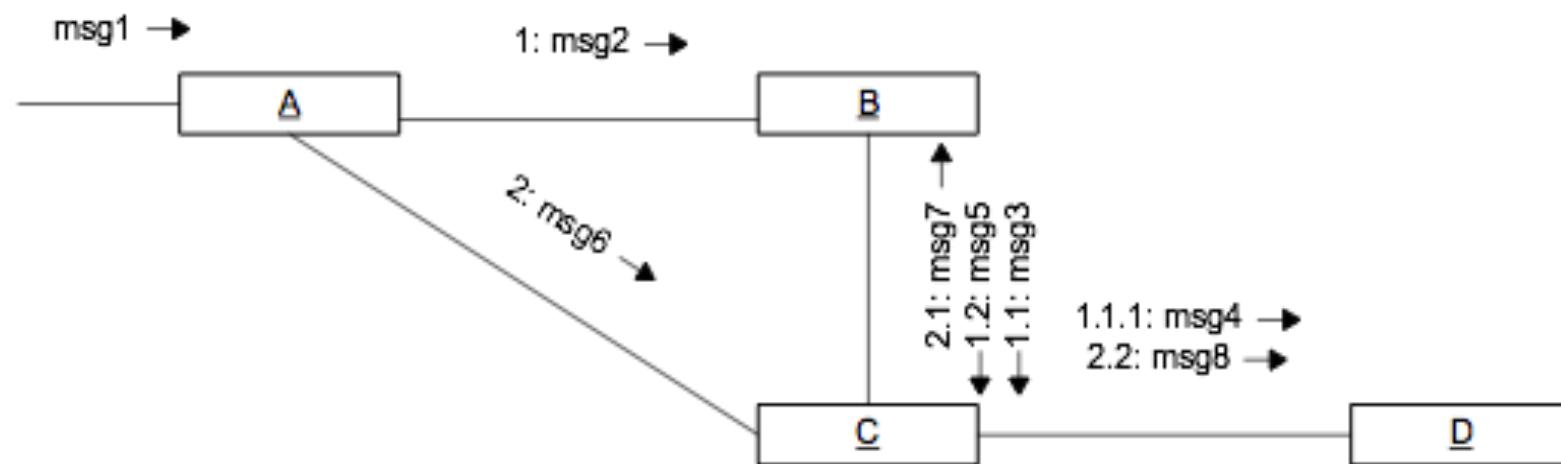


Collaboration diagram

- “creation” message and “destruction” message

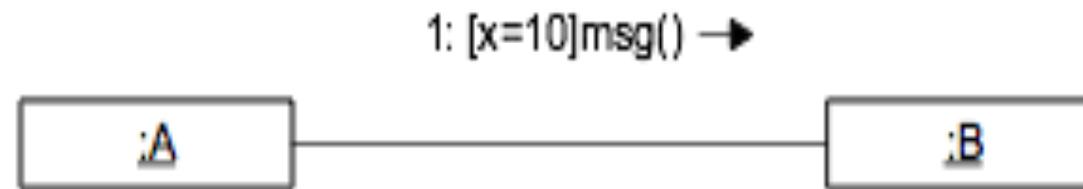


- Message numbering

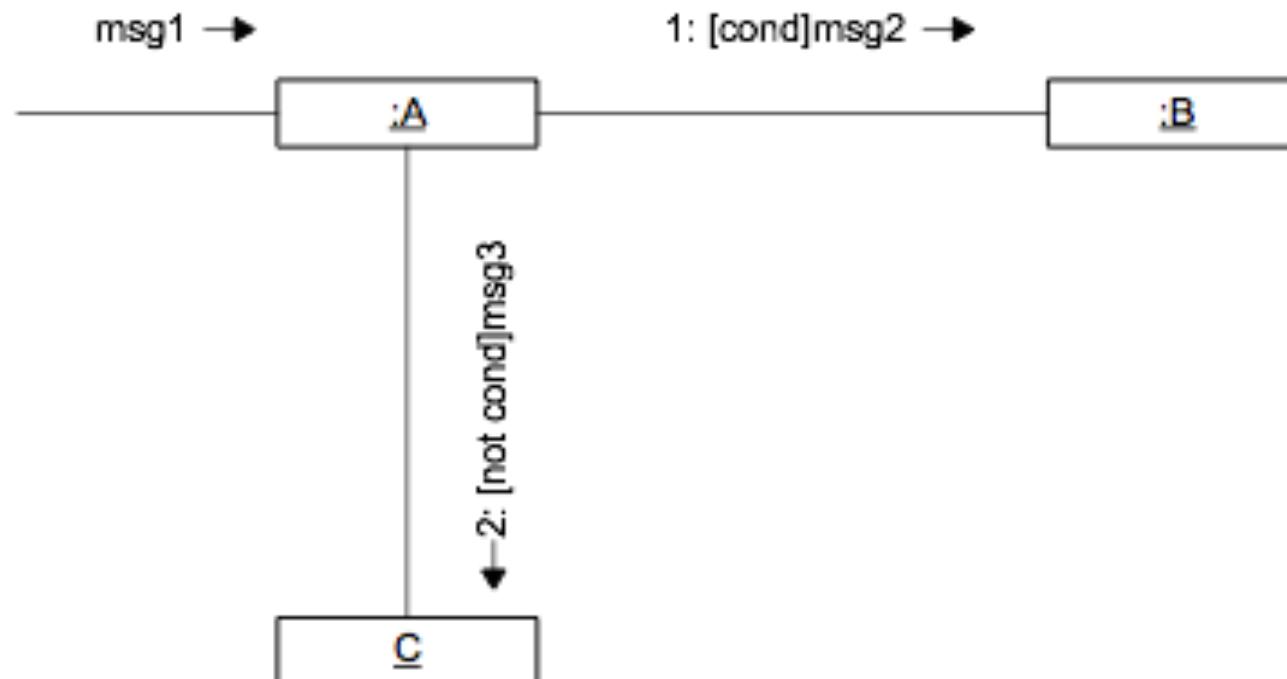


Collaboration diagram

- Conditional message

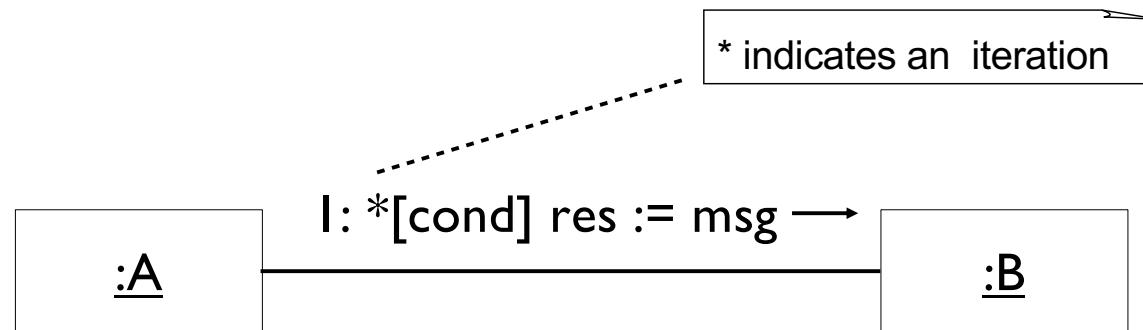


- Modelling a decision



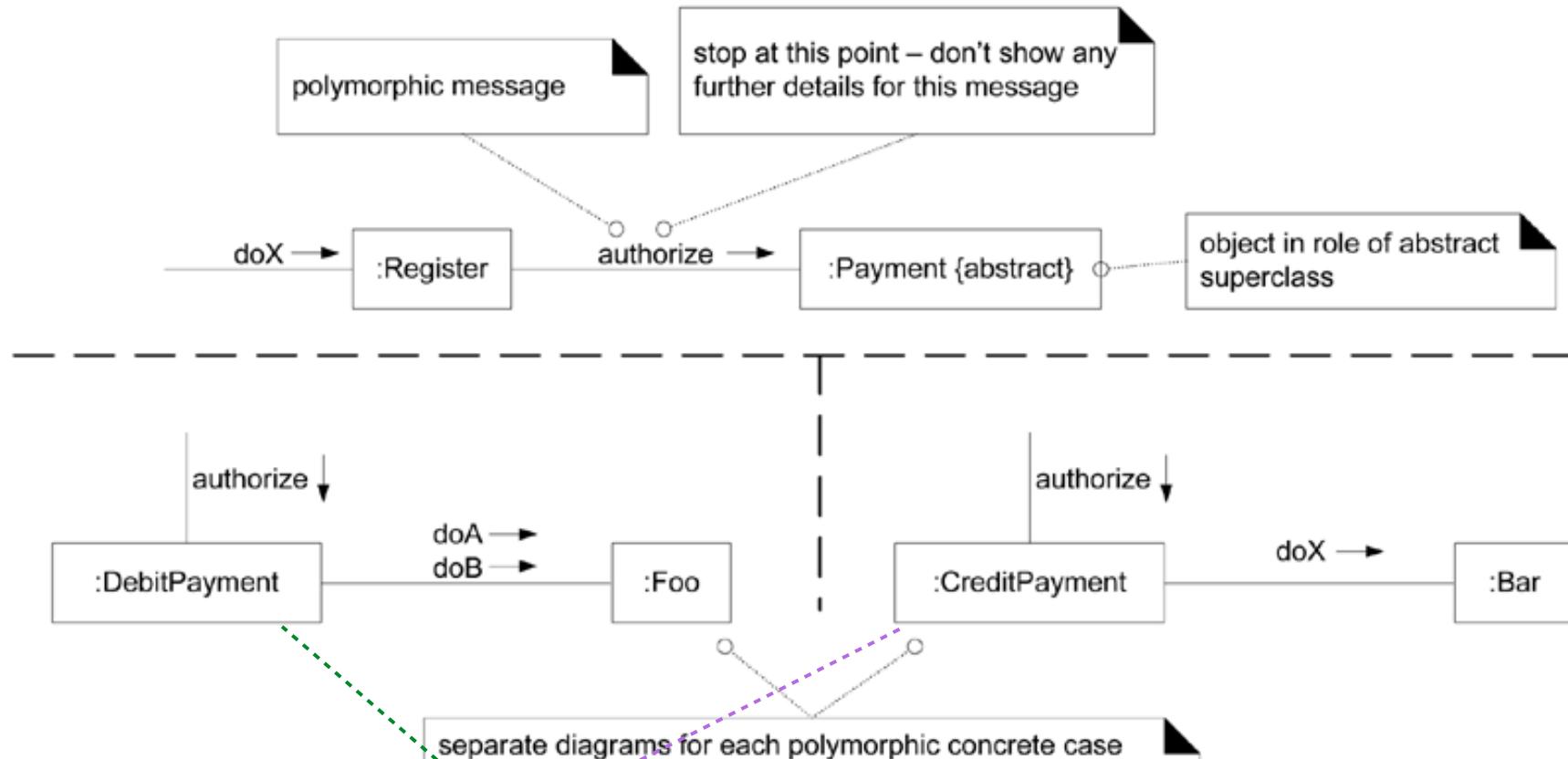
Collaboration diagram

- Modelling an iteration



Collaboration diagram

□ Modelling a polymorphic message



Payment

Pay by Credit or Debit card:

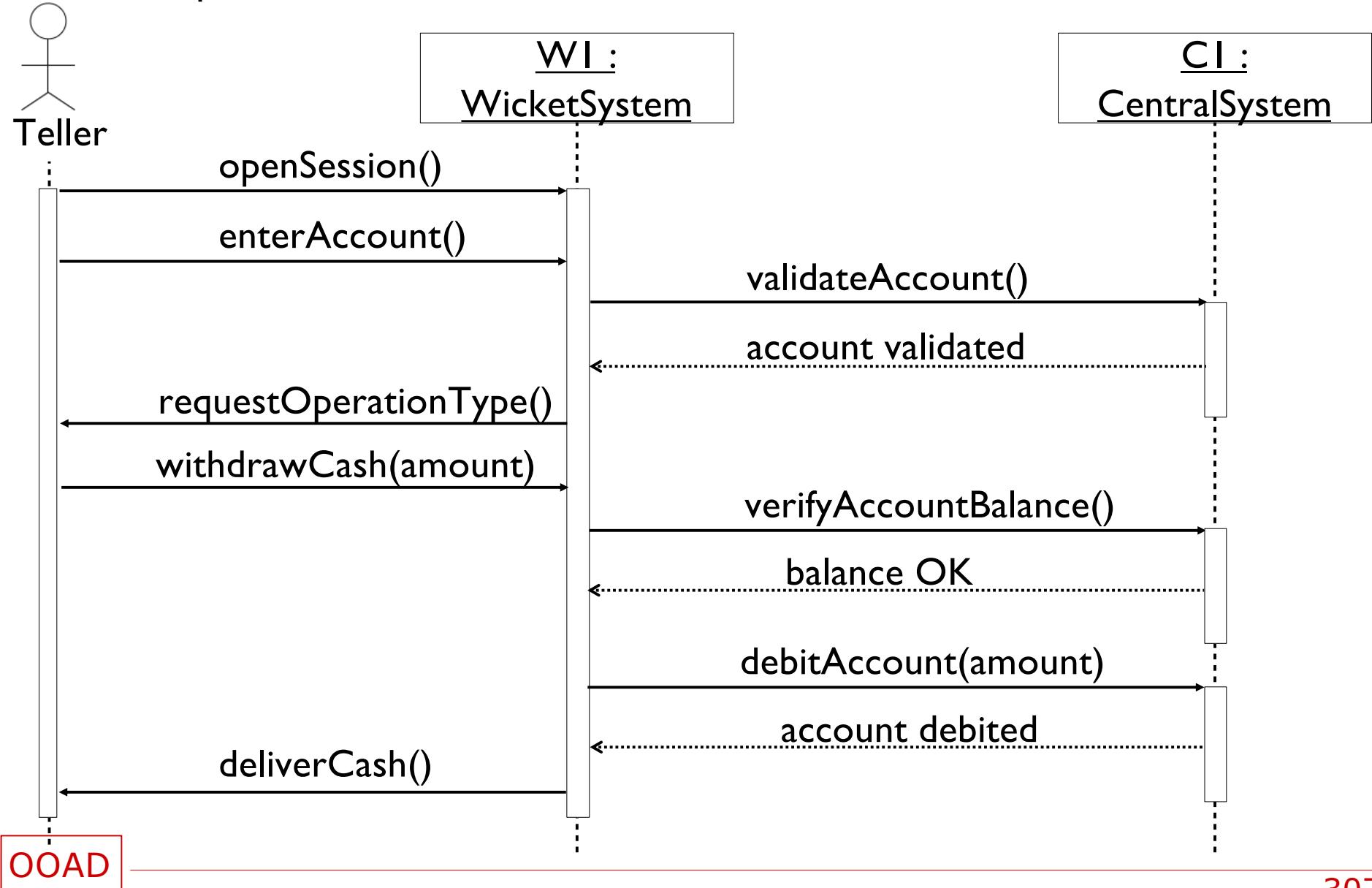
Card Number:	<input type="text"/>	<small>Please enter a valid card number</small>
Card Type:	<input type="button" value="Select card type"/>	<small>Please select a card type</small>
Expiry Date:	<input type="text"/> / <input type="text"/>	<small>Please select an expiry date</small>
Security Code (CVV):	<input type="text"/>	<small>Please enter a valid numeric security code (CVV)</small>
Cardholder's Name:	<input type="text"/>	<small>Please enter a valid cardholder's name</small>
Postcode/Zip Code:	<input type="text"/>	<small>Please enter a valid email</small>

Cash withdrawal at the bank



Sequence diagram

- Example: Cash withdrawal at the bank



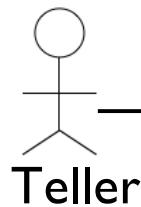
Collaboration diagram

- Example: cash withdrawal in the bank

1: openSession()

2: enterAccount()

6: withdrawCash(amount)



Teller

WI:

WicketSystem

5: requestOperationType()

11: deliverCash()

4: account validated

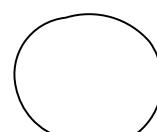
8: balance OK

10: account debited

3: validateAccount()

7: verifyAccountBalance()

9: debitAccount(amount)



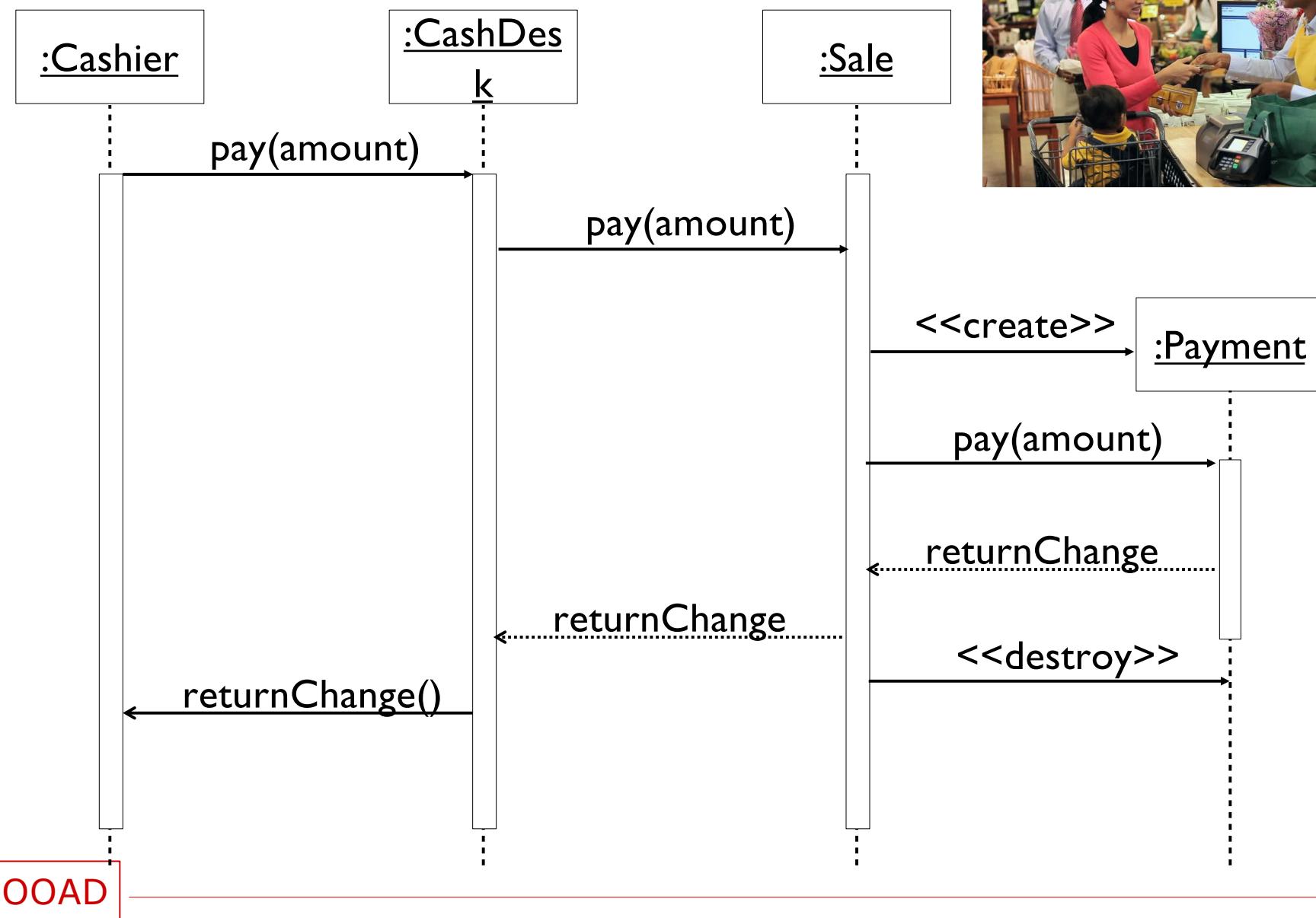
CI:

CentralSystem

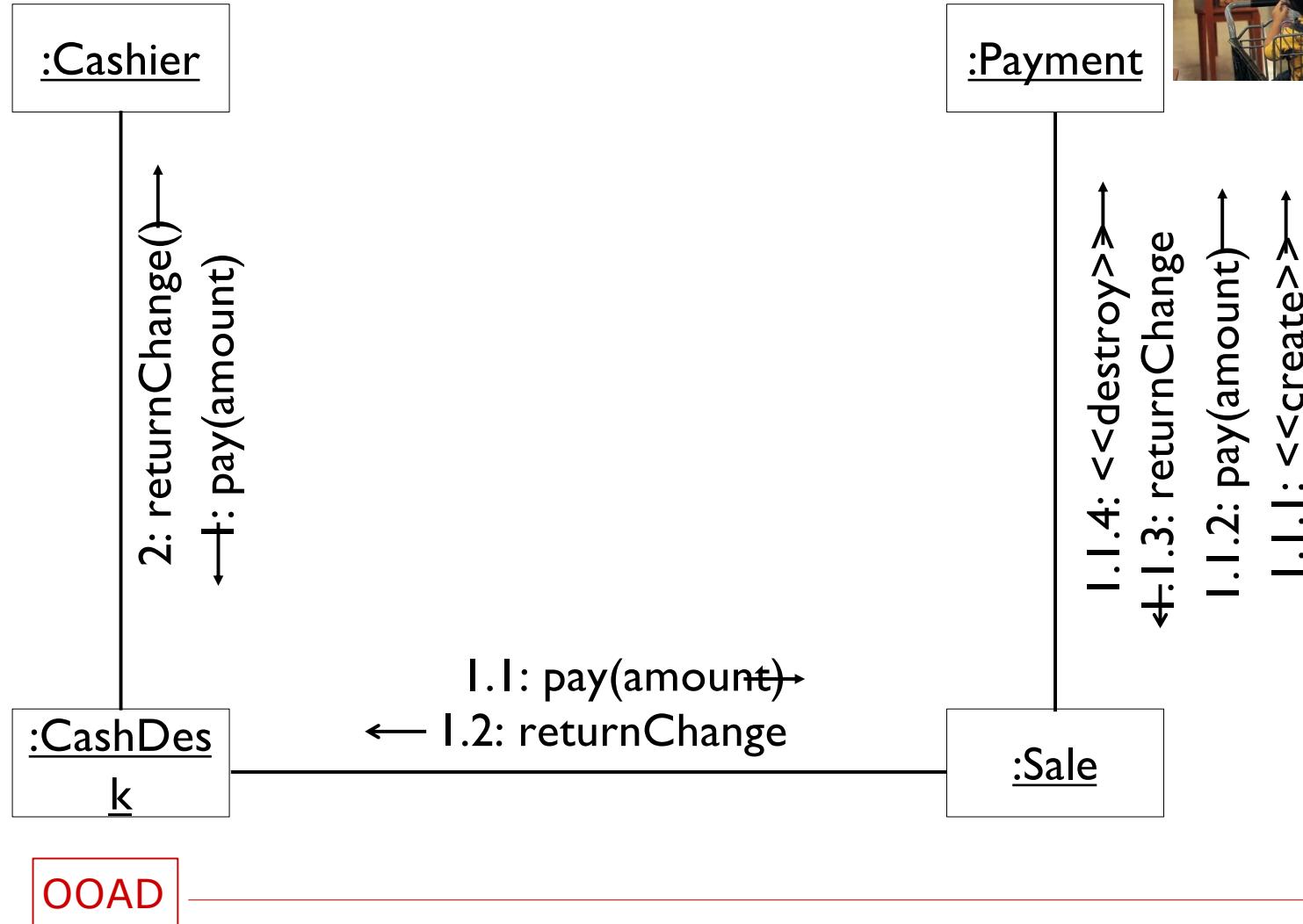
Use-case “cash payment”



Sequence diagram



Collaboration diagram

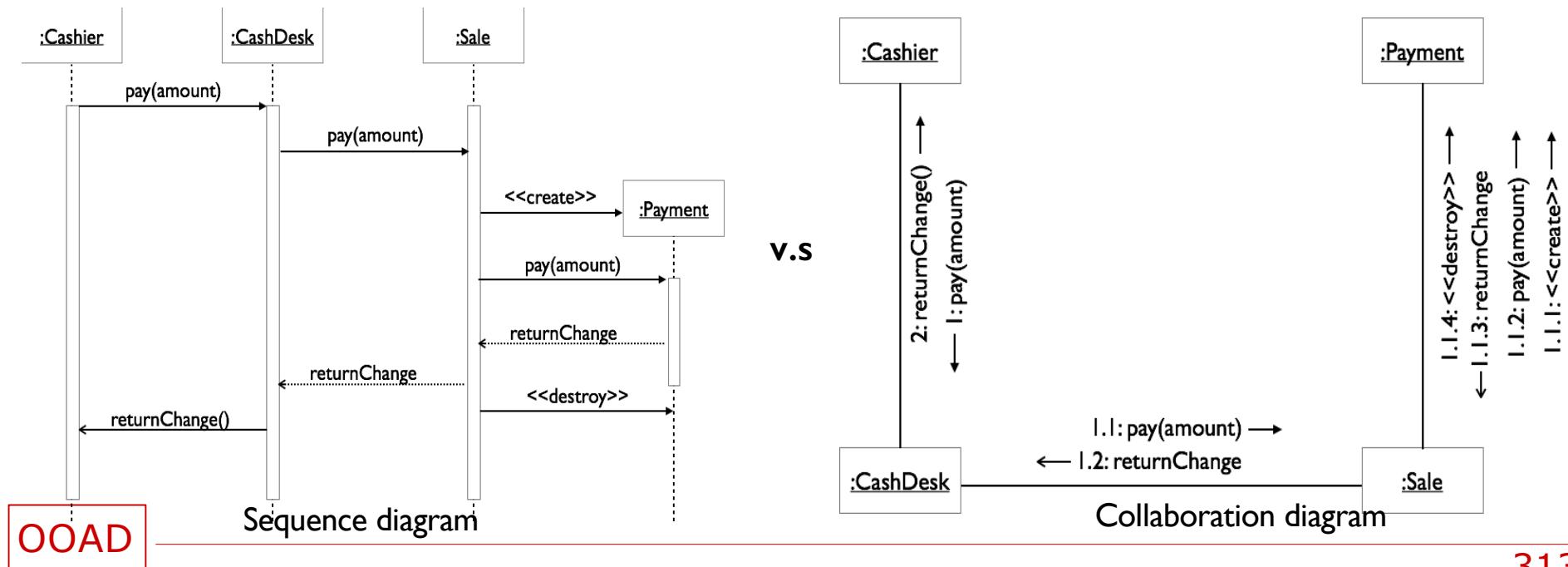


Sequence diagram v.s. Collaboration diagram

- Both sequence diagram and collaboration diagram are alternate representations of an interaction
- Sequence diagram
 - is a graphical view of a scenario
 - shows object interaction in a time-based sequence of what happens first, what happens next
 - establishes the roles of objects and help provide essential information to determine class responsibilities and interfaces
 - is normally associated with a use-case
- Collaboration diagram
 - shows how object associate with each other (objects, links and messages)
 - provides the structural relationships between objects

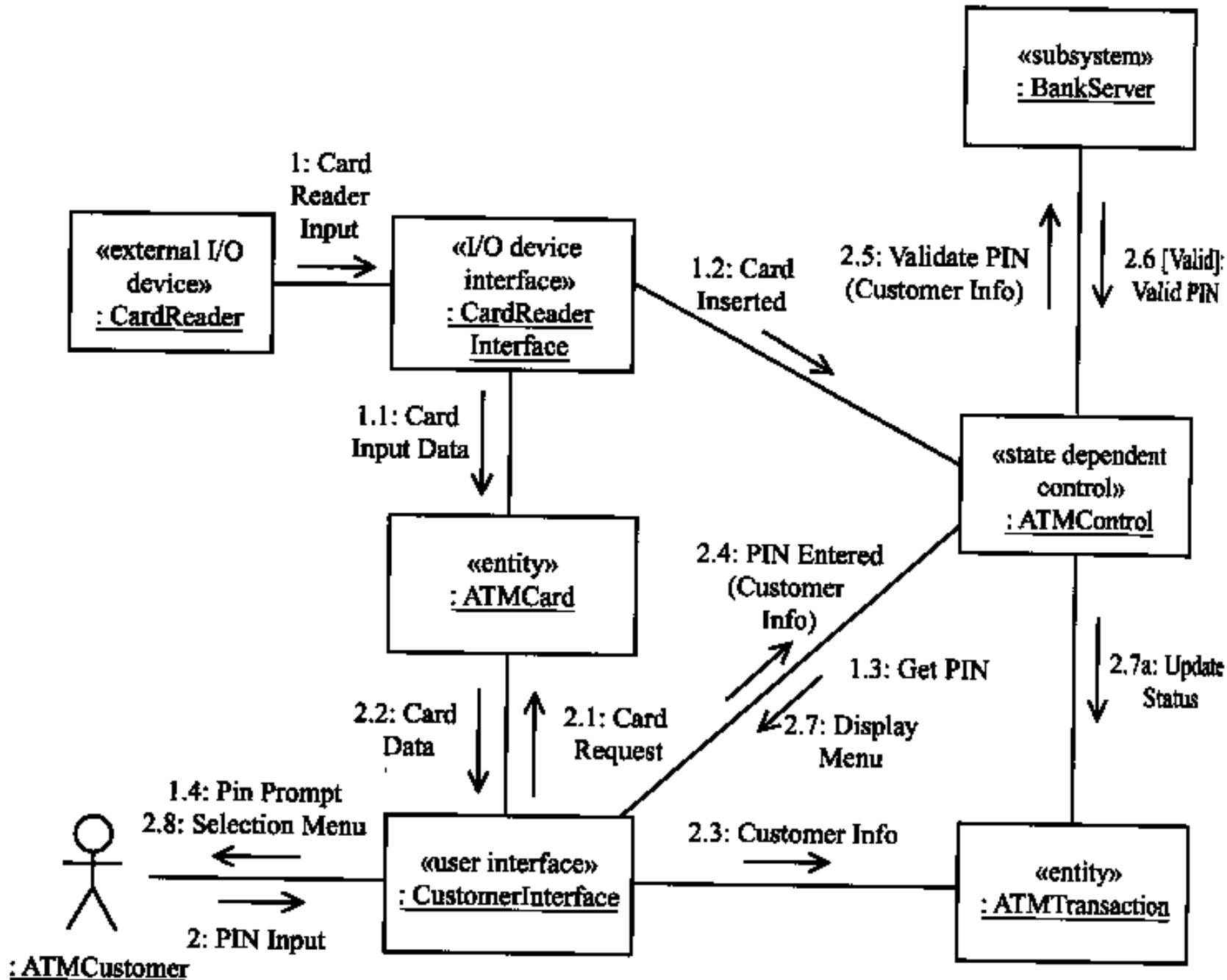
Sequence diagram v.s. Collaboration diagram

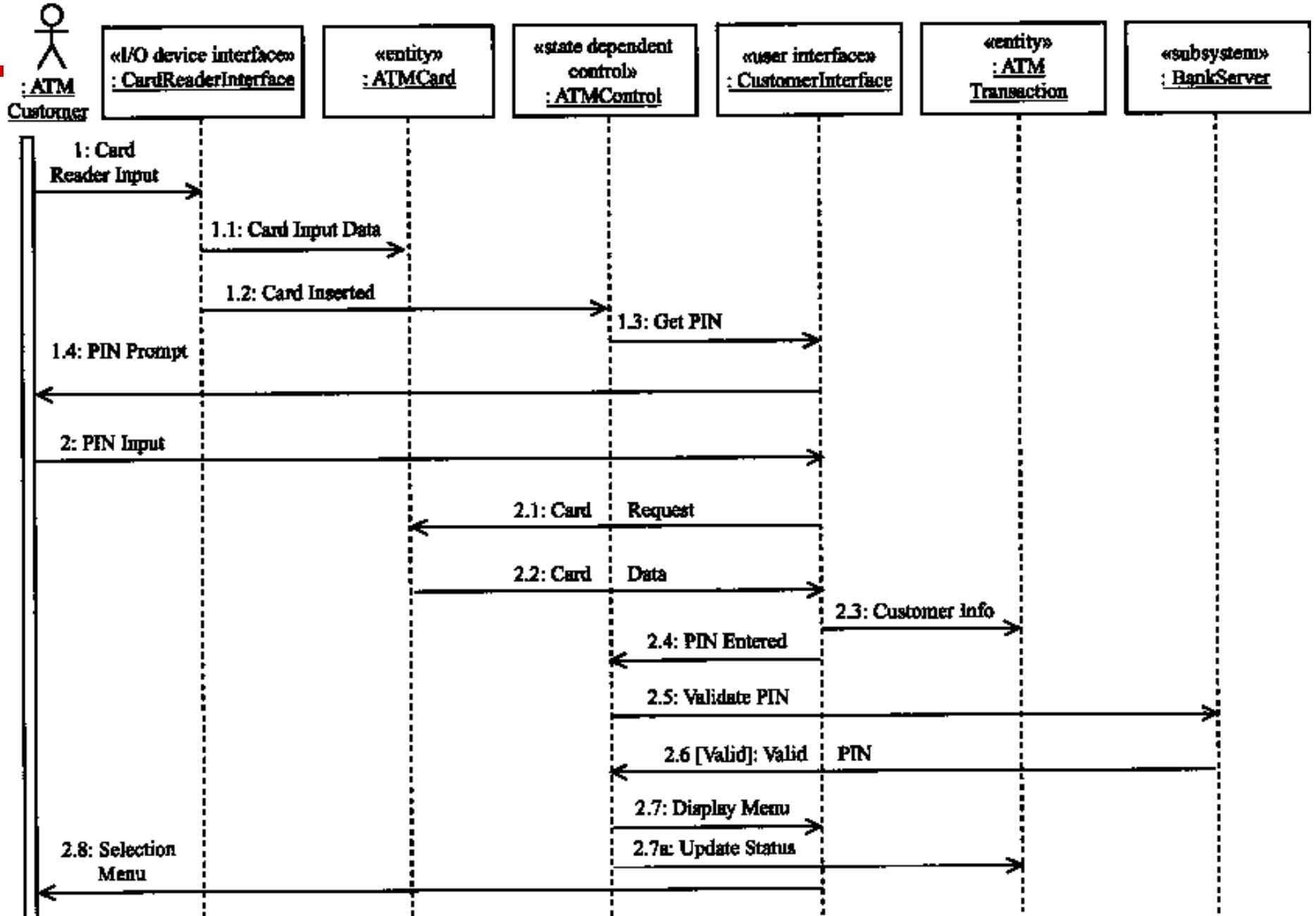
- Sequence diagram
 - Clearly shows the temporal ordering of messages
 - Consumes space
- Collaboration diagram
 - Is preferable when the interaction is deduced from the class diagram
 - Consumes less space
 - Is difficult to see the sequence of messages



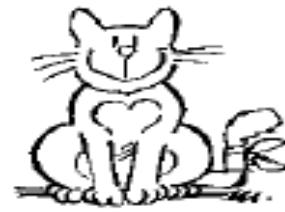
- Let's do a sequence diagram for the following casual use case, *Add Calendar Appointment* :

- The scenario begins when the user chooses to add a new appointment in the UI. The UI notices which part of the calendar is active and pops up an Add Appointment window for that date and time.
- The user enters the necessary information about the appointment's name, location, start and end times. The UI will prevent the user from entering an appointment that has invalid information, such as an empty name or negative duration. The calendar records the new appointment in the user's list of appointments. Any reminder selected by the user is added to the list of reminders.
- If the user already has an appointment at that time, the user is shown a warning message and asked to choose an available time or replace the previous appointment. If the user enters an appointment with the same name and duration as an existing group meeting, the calendar asks the user whether he/she intended to join that group meeting instead. If so, the user is added to that group meeting's list of participants.





Fun example



:Cat



:Policeman

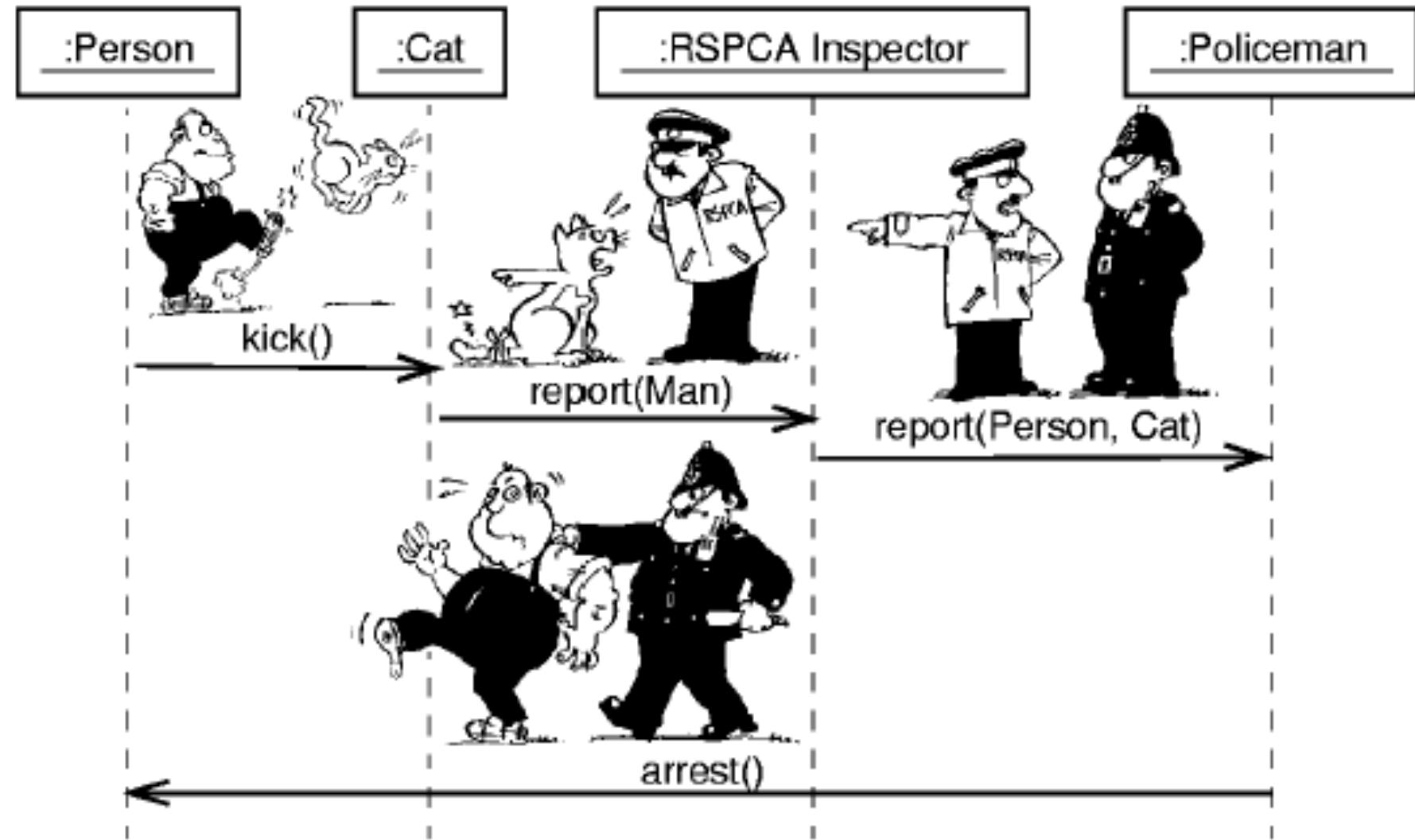


:Person

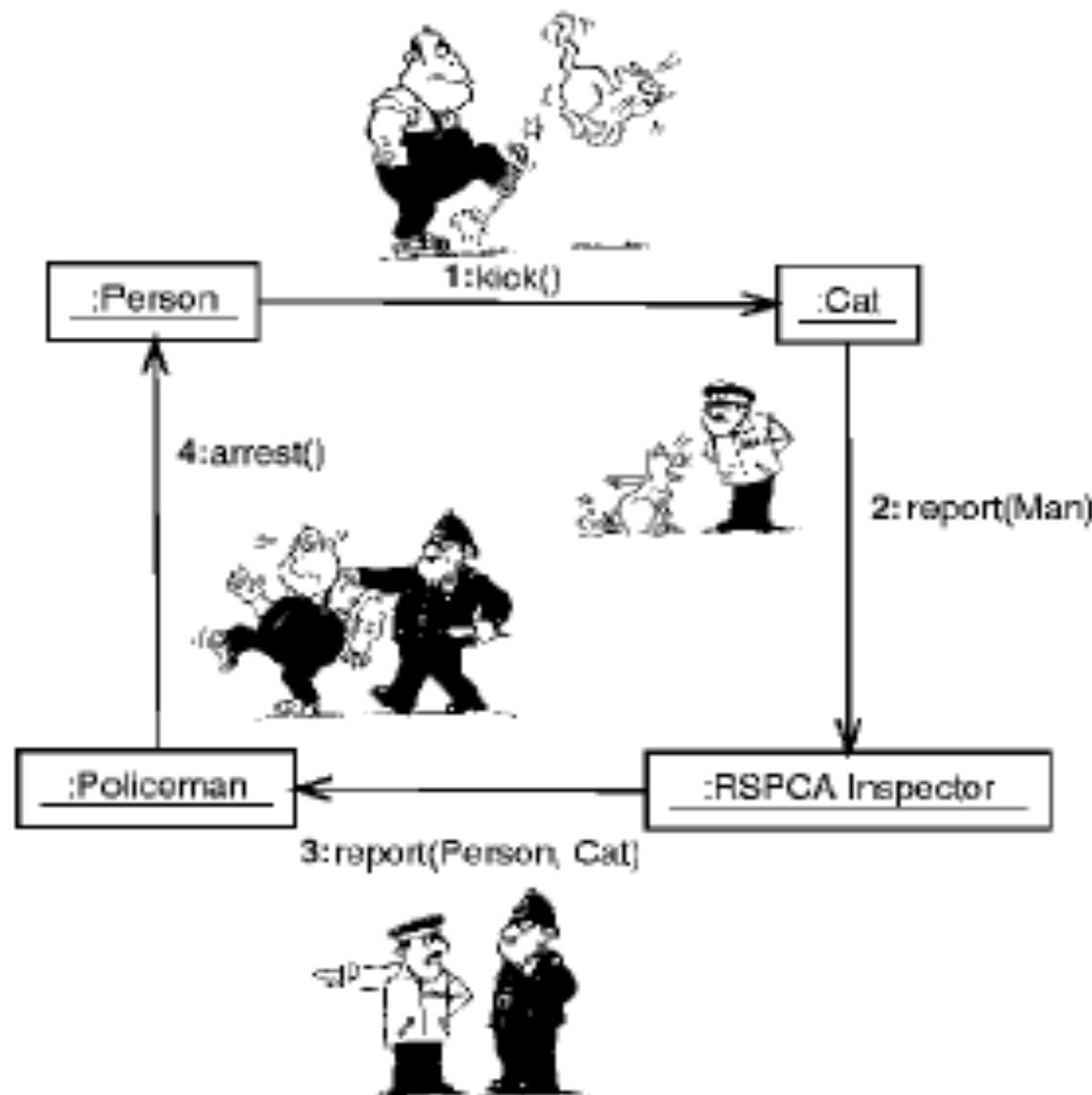


:RSPCA Inspector

Fun example: Sequence diagram

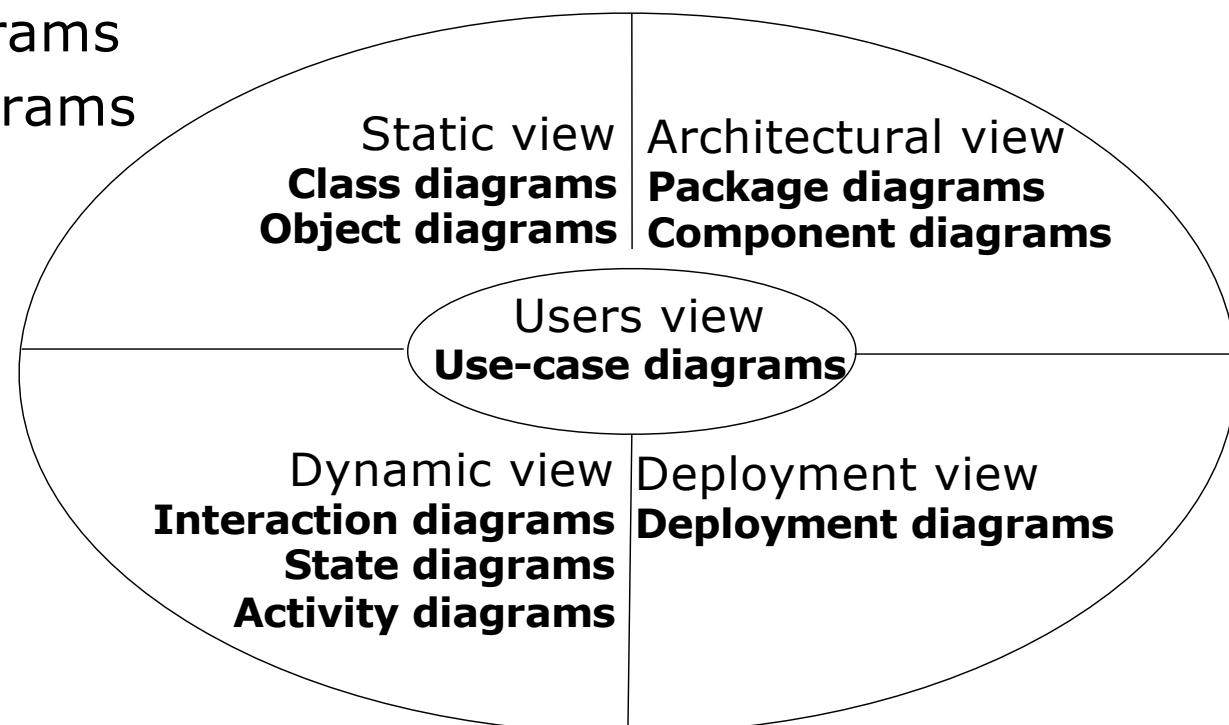


Fun example: Collaboration diagram



Architectural & Deployment modelling

- Package diagrams
- Component diagrams
- Deployment diagrams

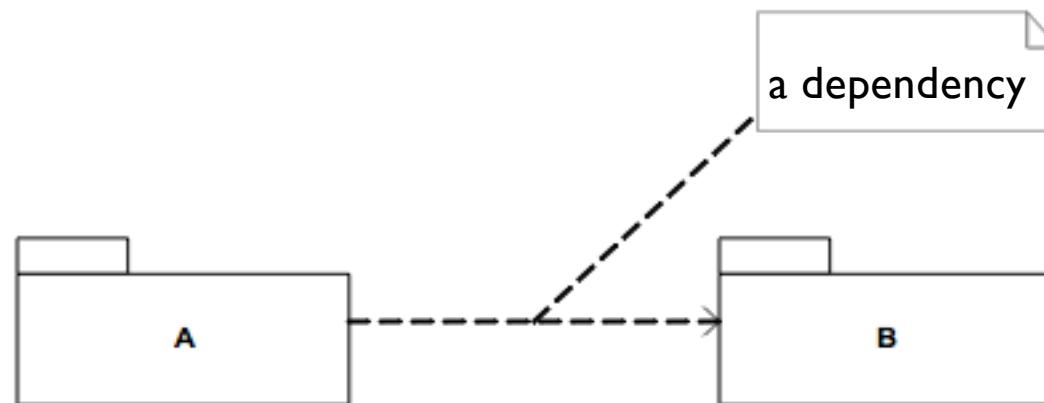


Package diagrams

- A package allows to group related elements
 - Several related classes are grouped together into a package
 - Several related packages are grouped into another package
- Notation

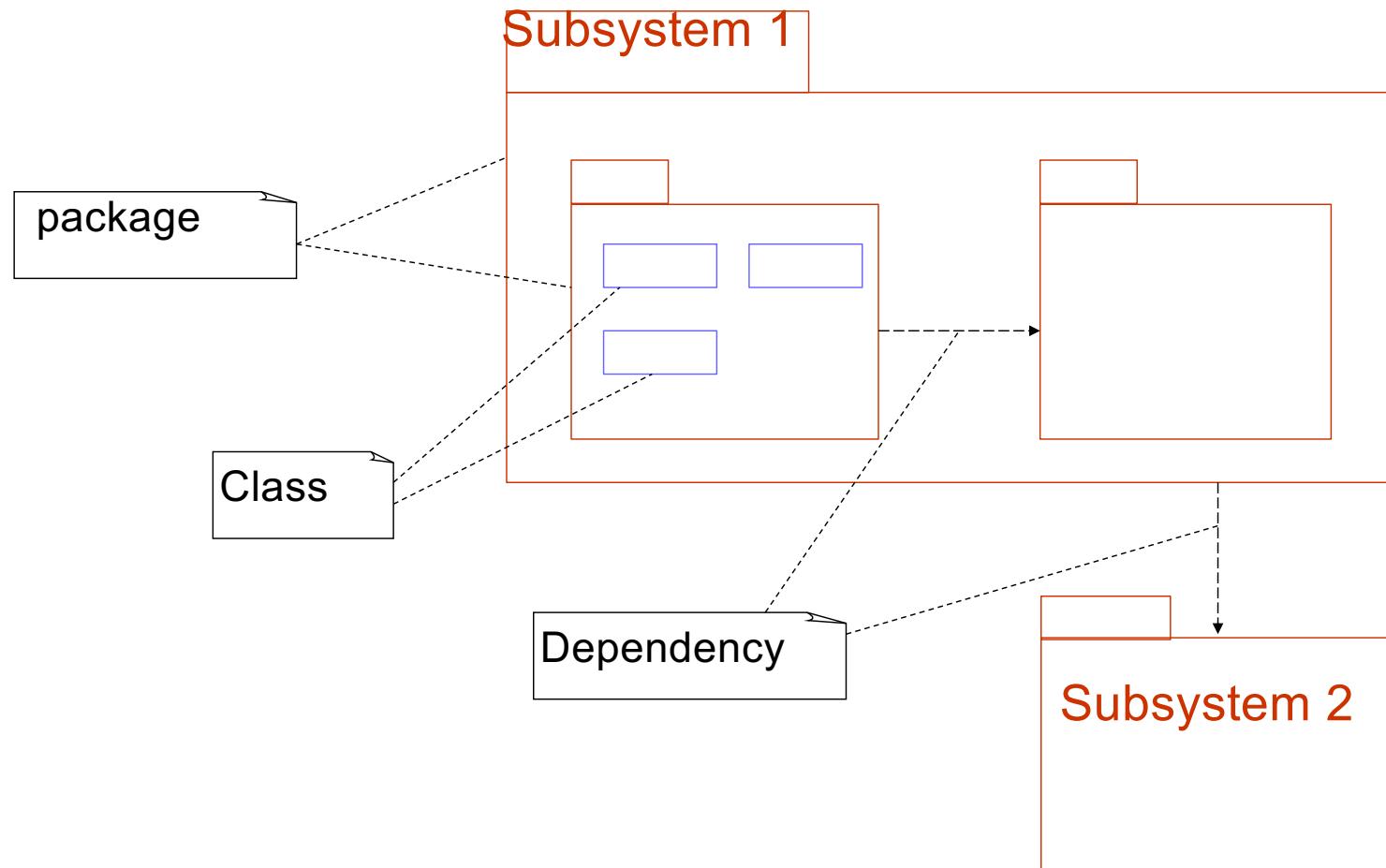


- Dependency
 - A package may depend on another package
 - For example, a package refers to an element of another package
 - Notation



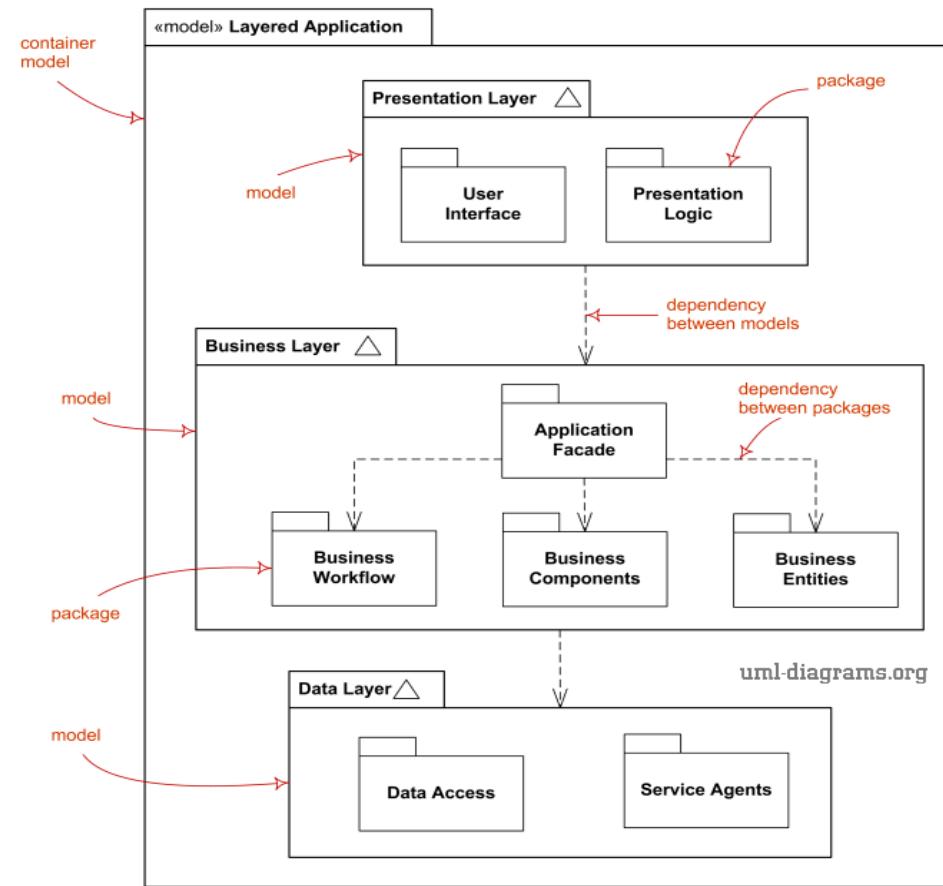
Package diagrams

- Example



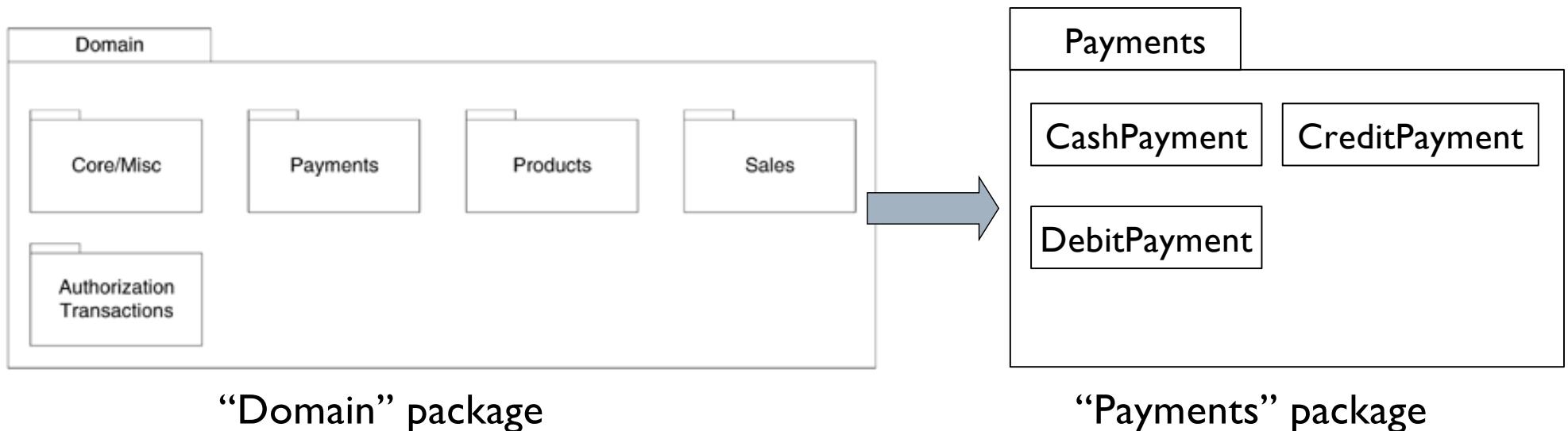
Package diagram

- Why packages?
 - Easy to manage, understand and manipulate
 - Reduce complexity
 - Iterative development: different developers, teams work simultaneously on different packages
- Example



Package diagrams

- Organizing principles of packages
 - **Functional cohesion**
 - Classes/interfaces that are grouped are strongly associated in terms of purpose, service, collaboration, function
 - Example: all elements of the “payment” package are related to the payment of the products



Package diagrams

- Organizing principles of packages
 - Functional cohesion
 - **The cohesion of a package is quantified by**

$$C = \frac{\text{Number Of Internal Relationships}}{\text{Number Of Elements}}$$

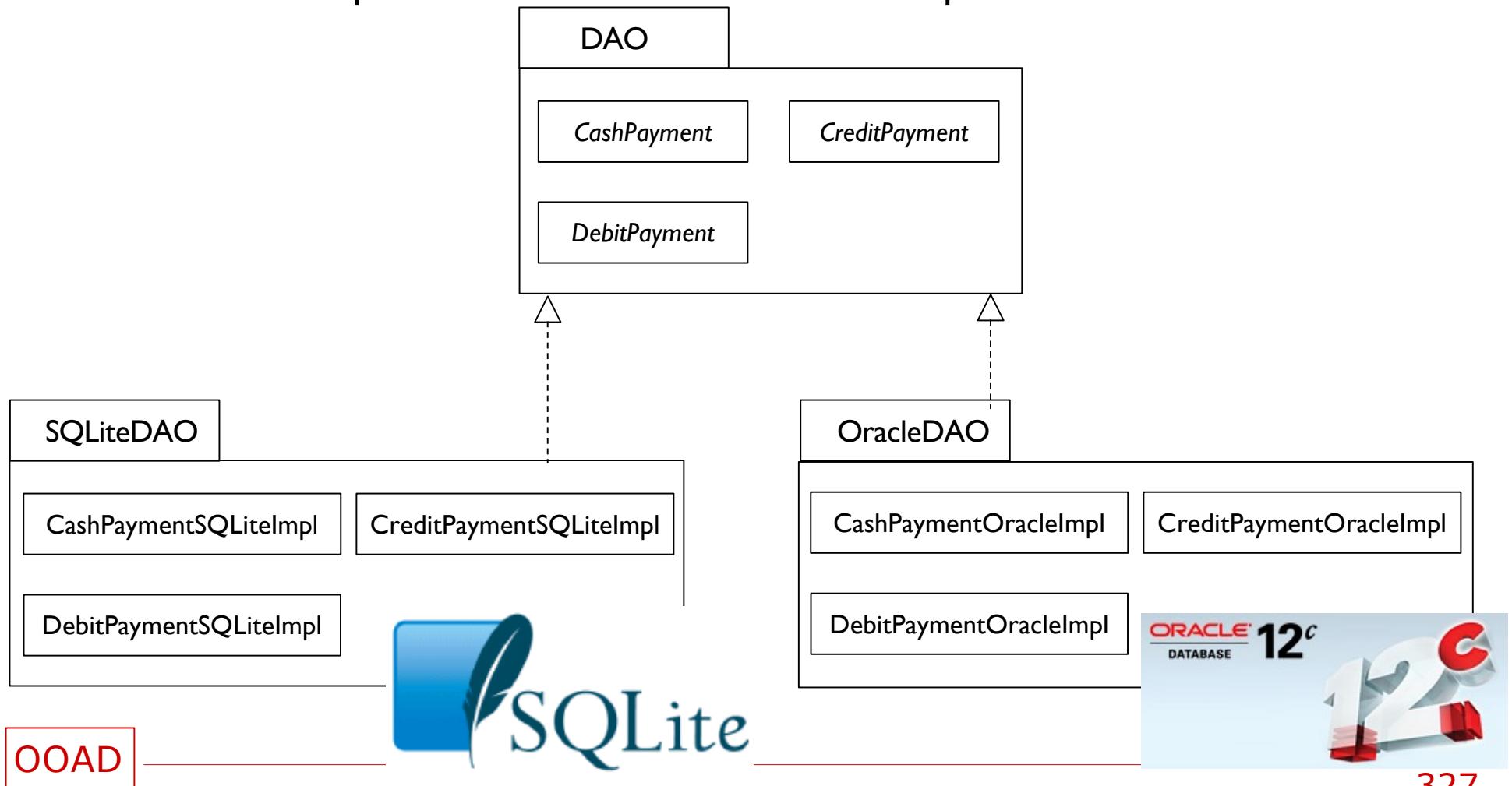
- A small value of C can say that
 - The package contains too many non-related items, is not well organised
 - The package contains no-related items with the intent of the designer
 - for example a “tools” packages
 - The package contains a subset of highly cohesive elements, but the whole is not cohesive

Package diagrams

- Organizing principles of packages (continued)

■ Package of interface

- Related interfaces are placed in a package
- The implementation classes are separated

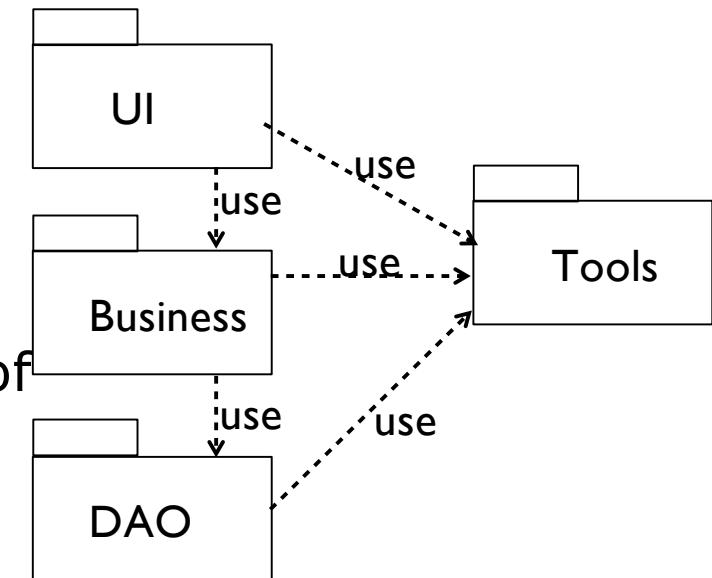


Package diagrams

- Organizing principles of packages (continued)
 - **Package of unstable elements**
 - Stable elements are grouped in a package
 - Unstable elements are grouped together in a package
 - These are items that are often modified and redistributed
 - To reduce the impact on the stable elements
 - Example: a package includes twenty classes of which ten are often modified and redistributed. It is best to separate them into two packages: one includes ten stable classes, the other is unstable.

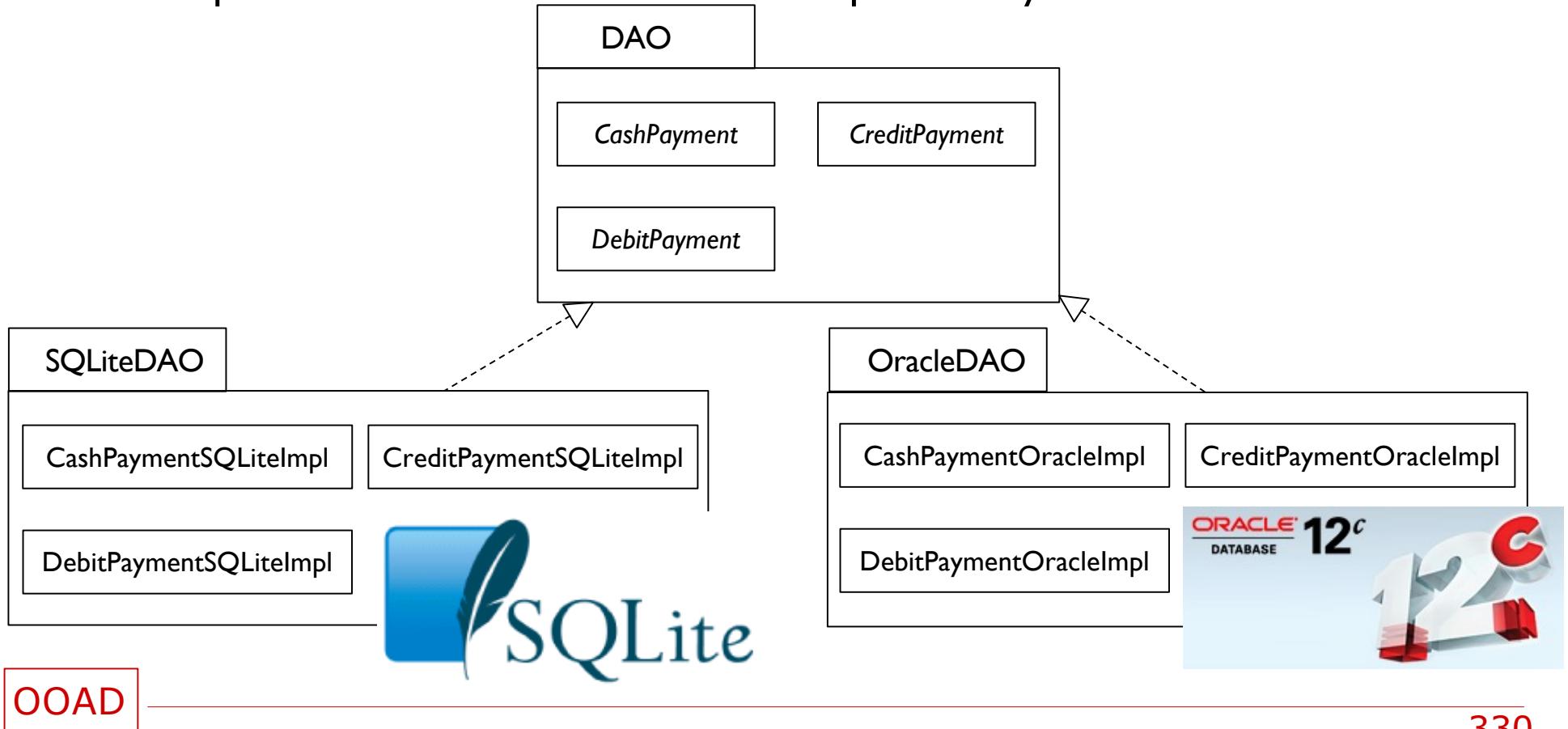
Package diagrams

- Organizing principles of packages (continued)
 - The more dependent the more stable
 - The more dependent a package has, the more stable it should be
 - Since changes on these packages has big impact on other packages
 - For example, a “tools” package needs to be stable.
 - Some ways to improve the stability of a package
 - It contains only interfaces or abstract classes
 - It doesn’t depend on other packages, or depends only on very stable packages
 - It contains stable code (well implemented)



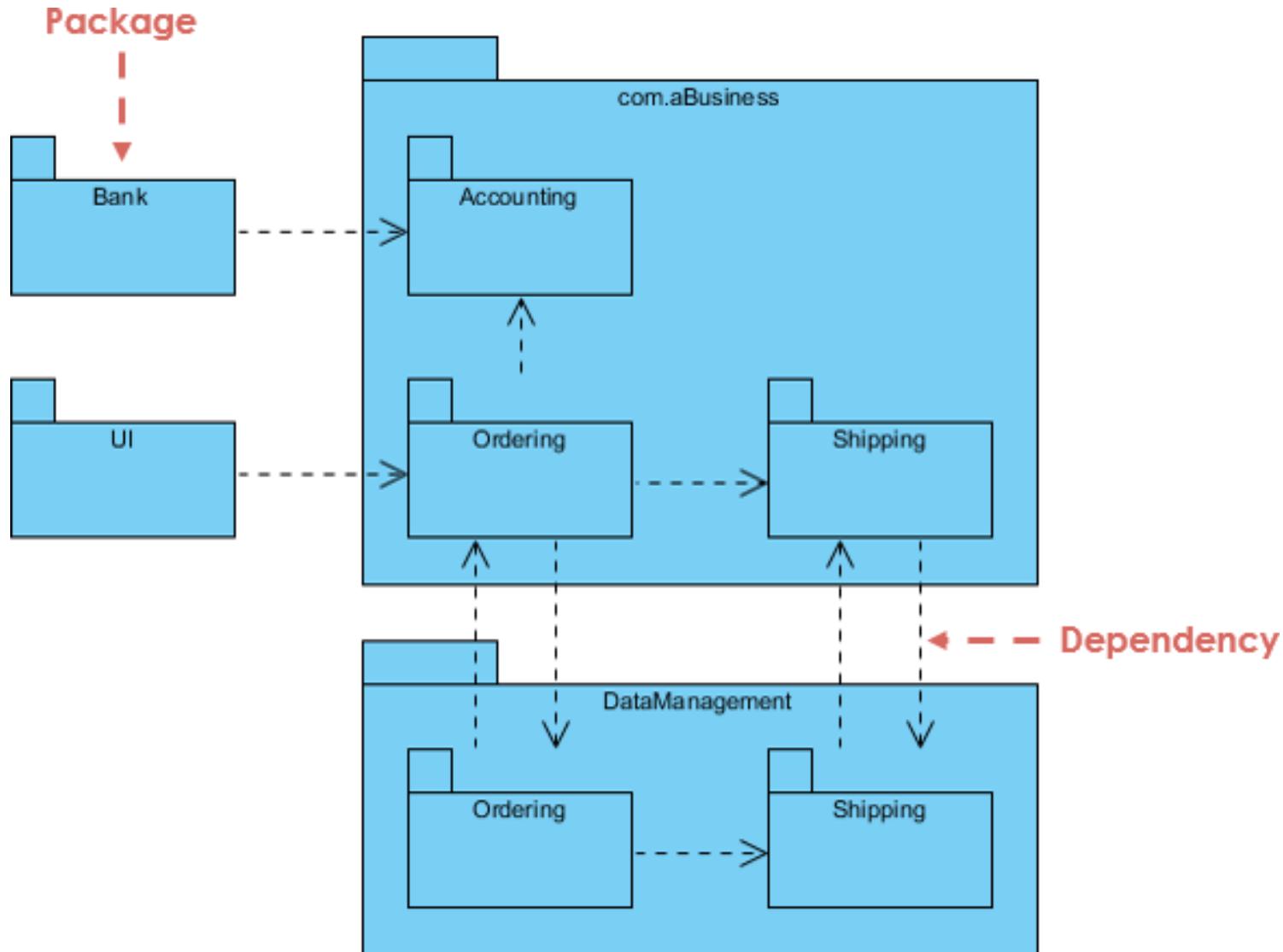
Package diagrams

- Organizing principles of packages (continued)
 - **Package of independent elements**
 - The elements that are used independently or in different contexts are separated into different packages
 - Example: SQLite & Oracle for development/testing & production environments respectively



Package diagrams

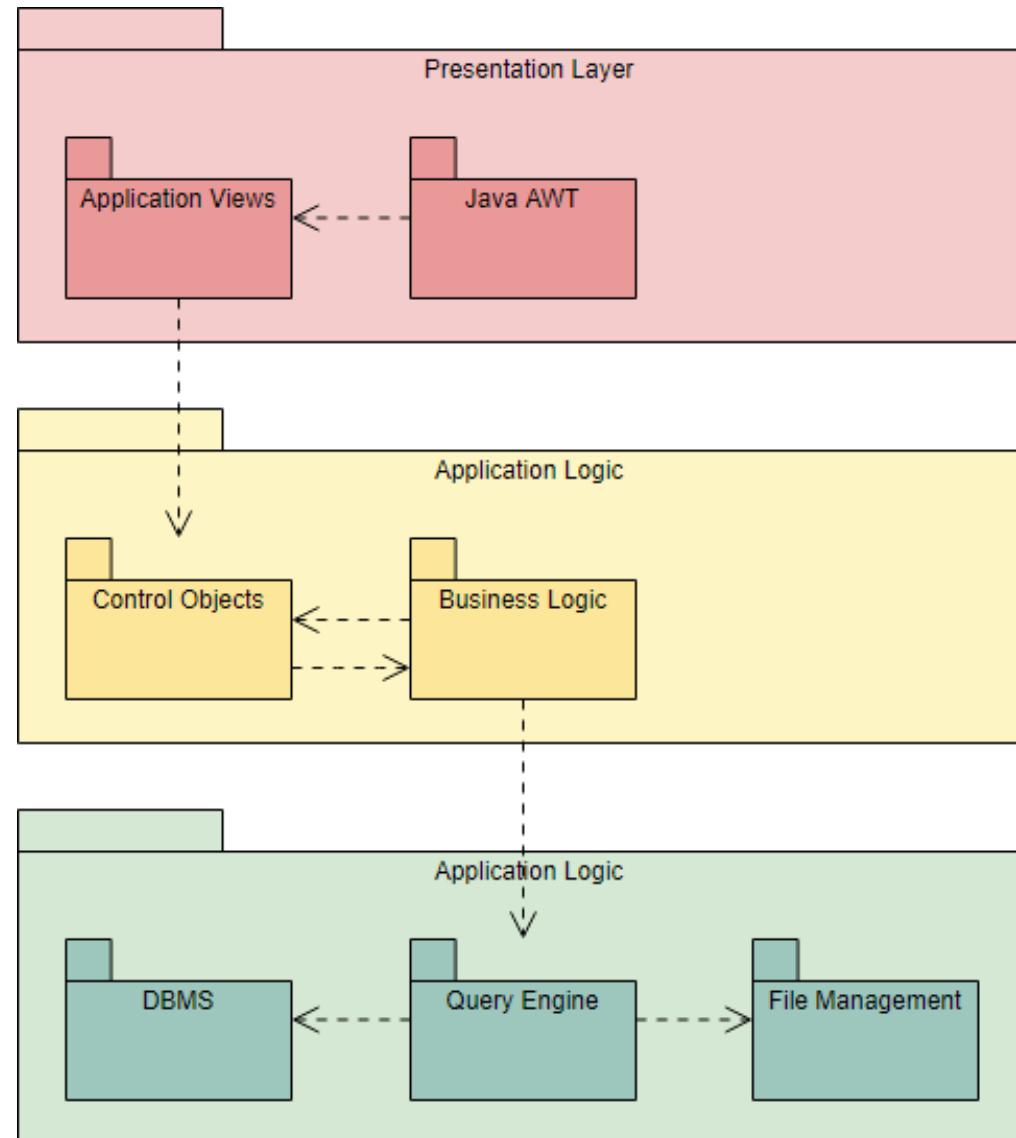
□ Examples



Package diagrams

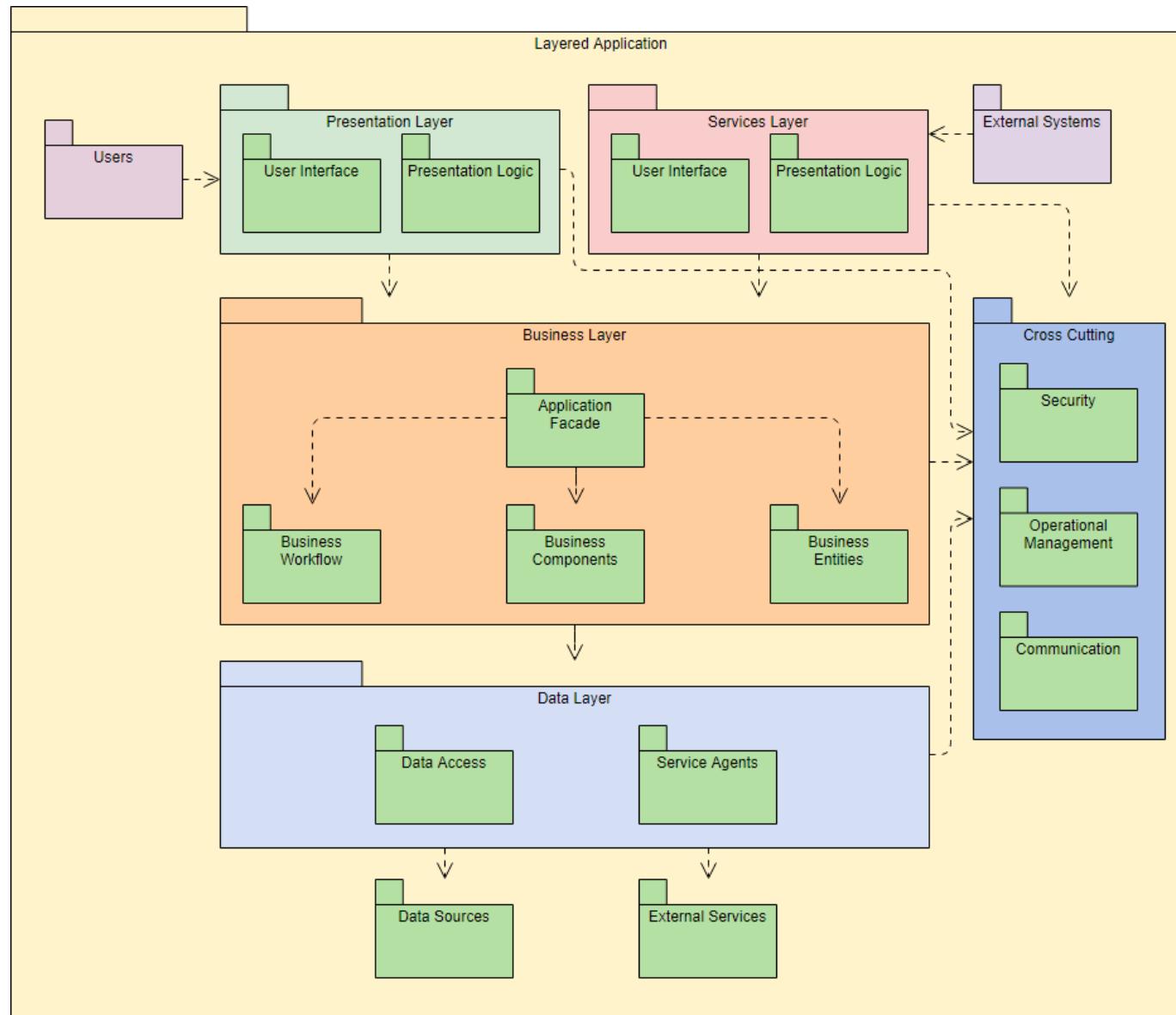
□ Examples: MVC Structure

□



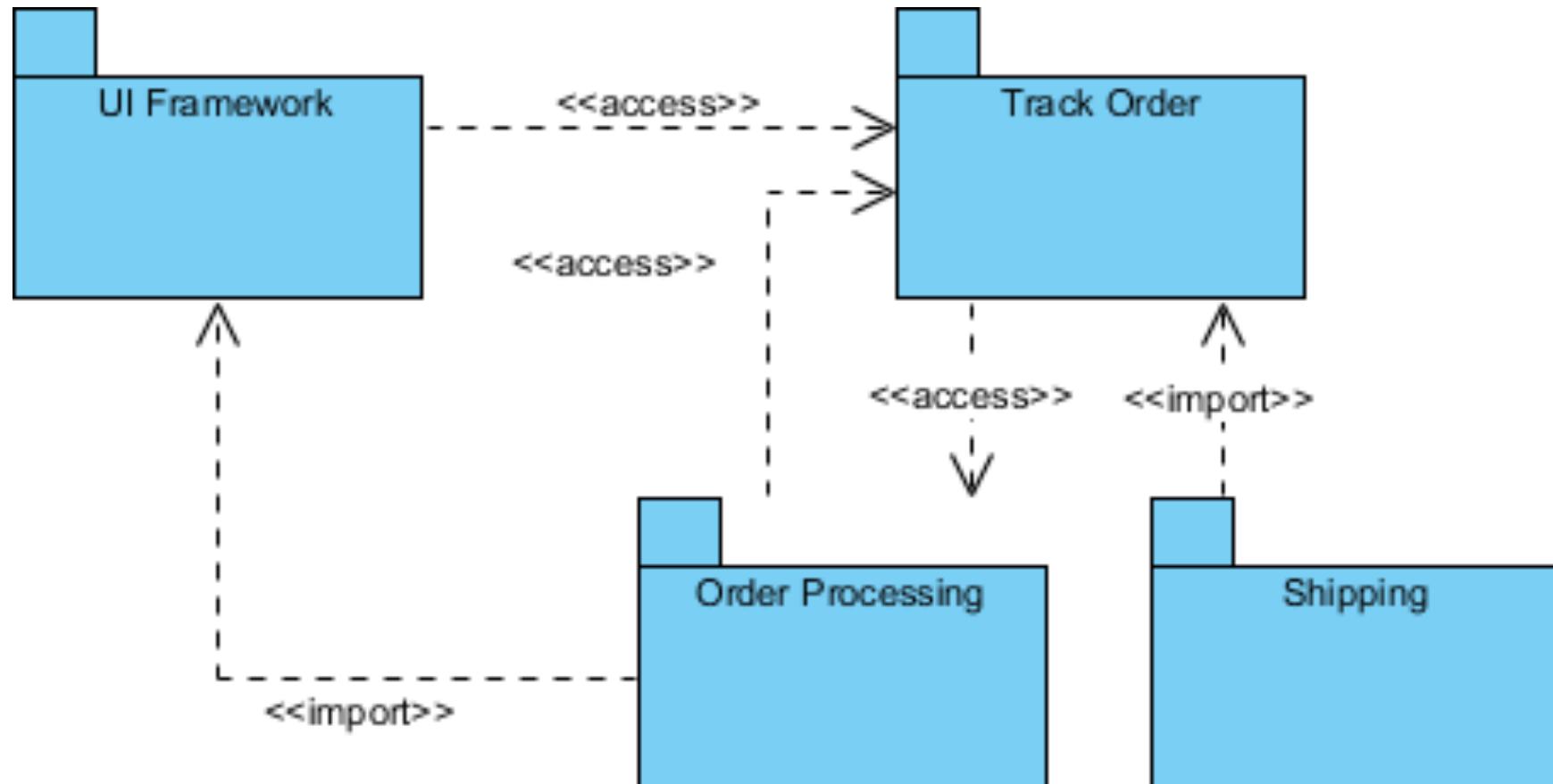
Package diagrams

□ Examples: Layering Structure



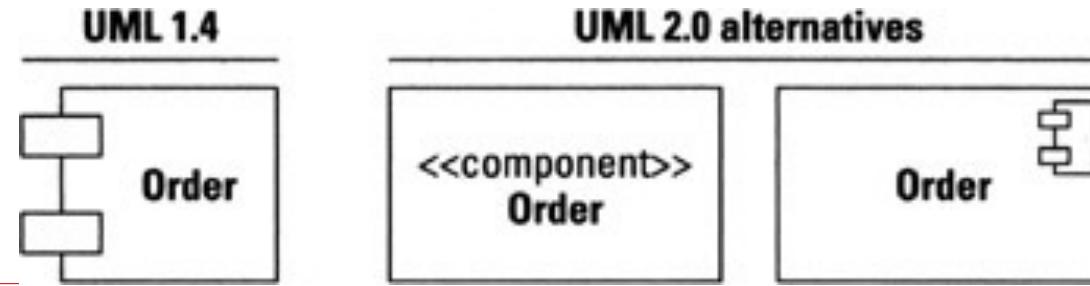
Package diagrams

- Examples: Order Processing System



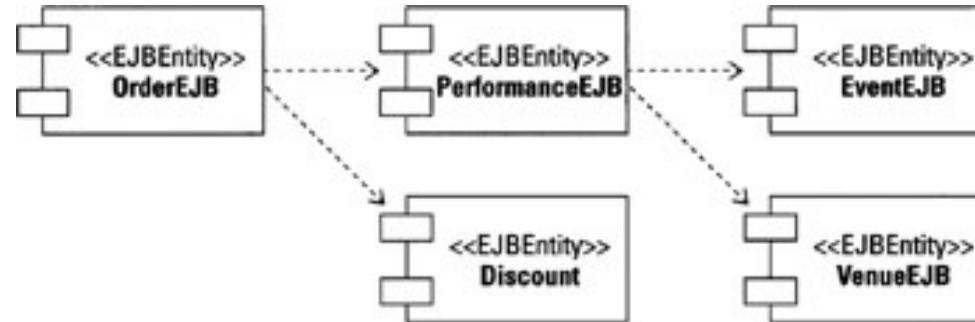
Component diagrams

- A component diagram models the **architectural view** of the system, the physical implementation of the system
 - Classes describe the logical organization, while *Components* describe the physical implementations
- Component diagrams define physical software modules and their relationships to one another
- *Components* may represent anything from a single class to applications, subsystems, and systems
- *Artifacts* that *implement* components
- The artifacts may be any type of code that can reside in any type of memory-source code, binary files, scripts, executable files, databases, or applications
- *Dependencies* represent the types of relationships that exist between components on a Component diagram
- Notation

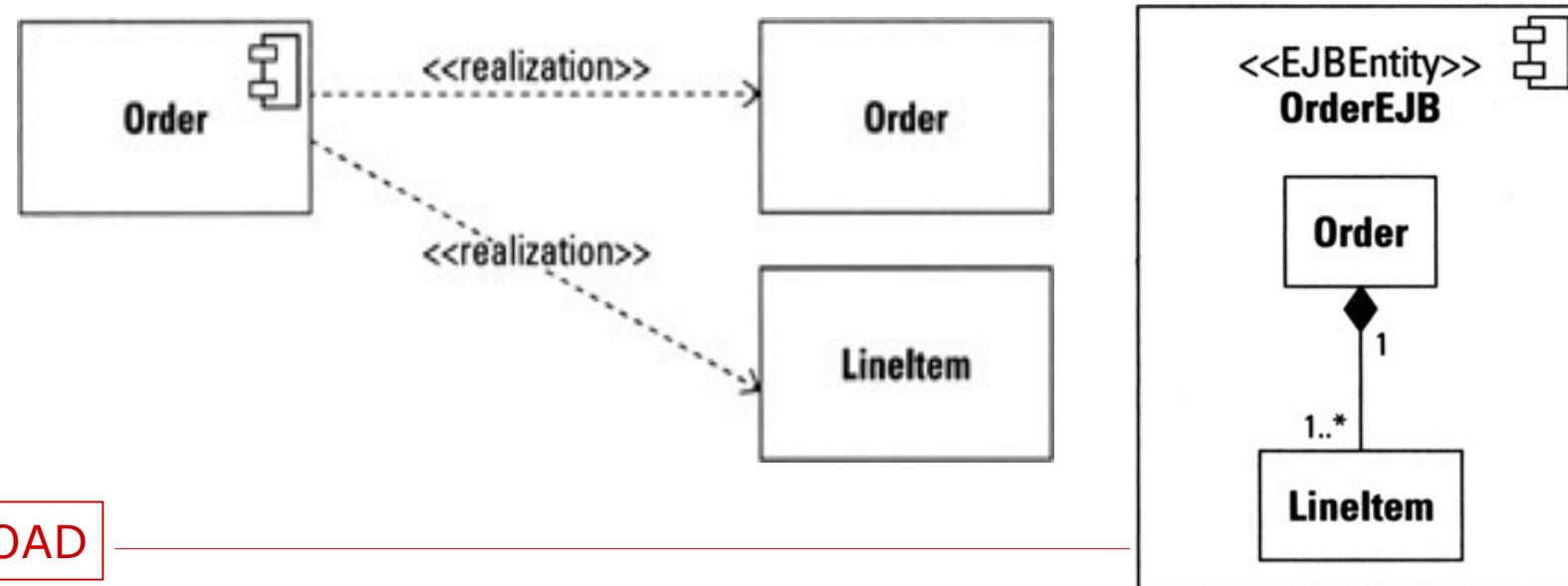


Component diagrams

- Modeling dependencies

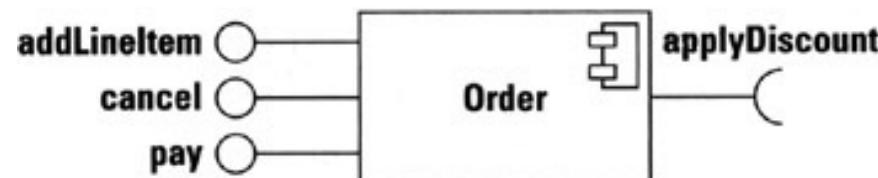
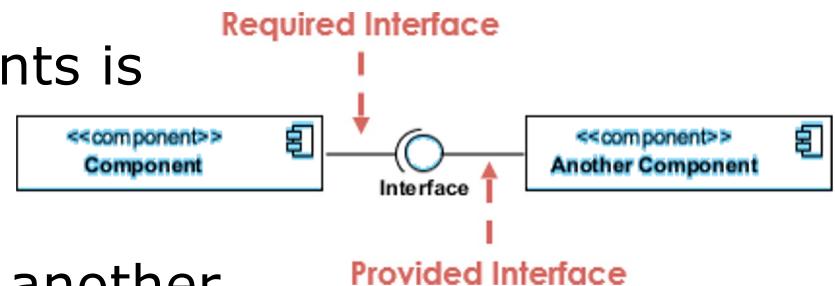


- A component is an abstraction, a representation of a requirement for the physical software.
- *Realization* refers to any implementation of a requirement.
- Component can contain the realizations (implementations)



Component diagrams

- A fundamental feature of components is the ability to define interfaces
- Interfaces come in two types:
 - *Provided interface*: defines how another component must ask for access to a provided service
 - *Required interface*: defines an interface exactly what it needs
- Example: Order component *provides* the services "add line items to the order", "cancel the order", and "pay for the order"; and *requires* discount (from other component)

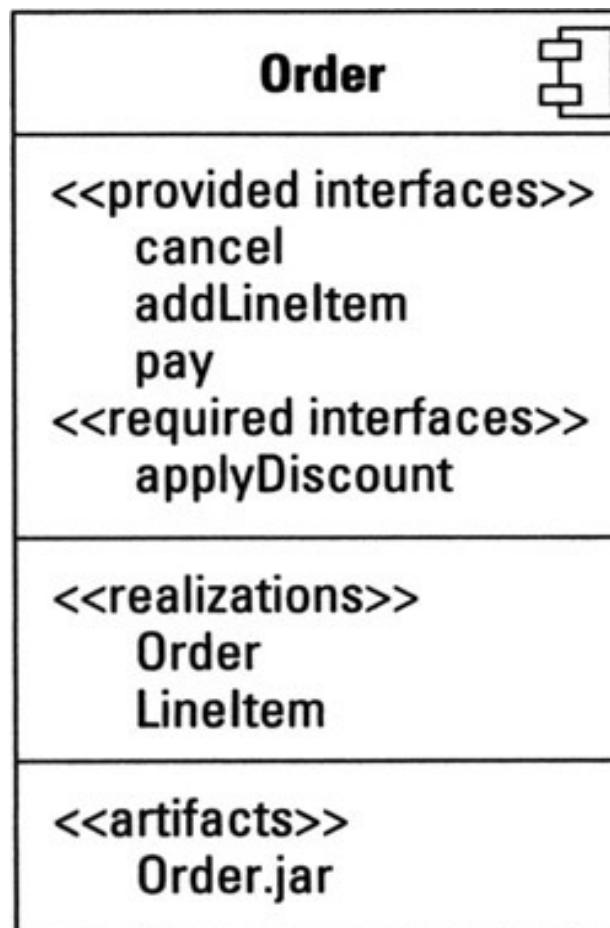


- Connecting required and provided interfaces to form a partnership between components



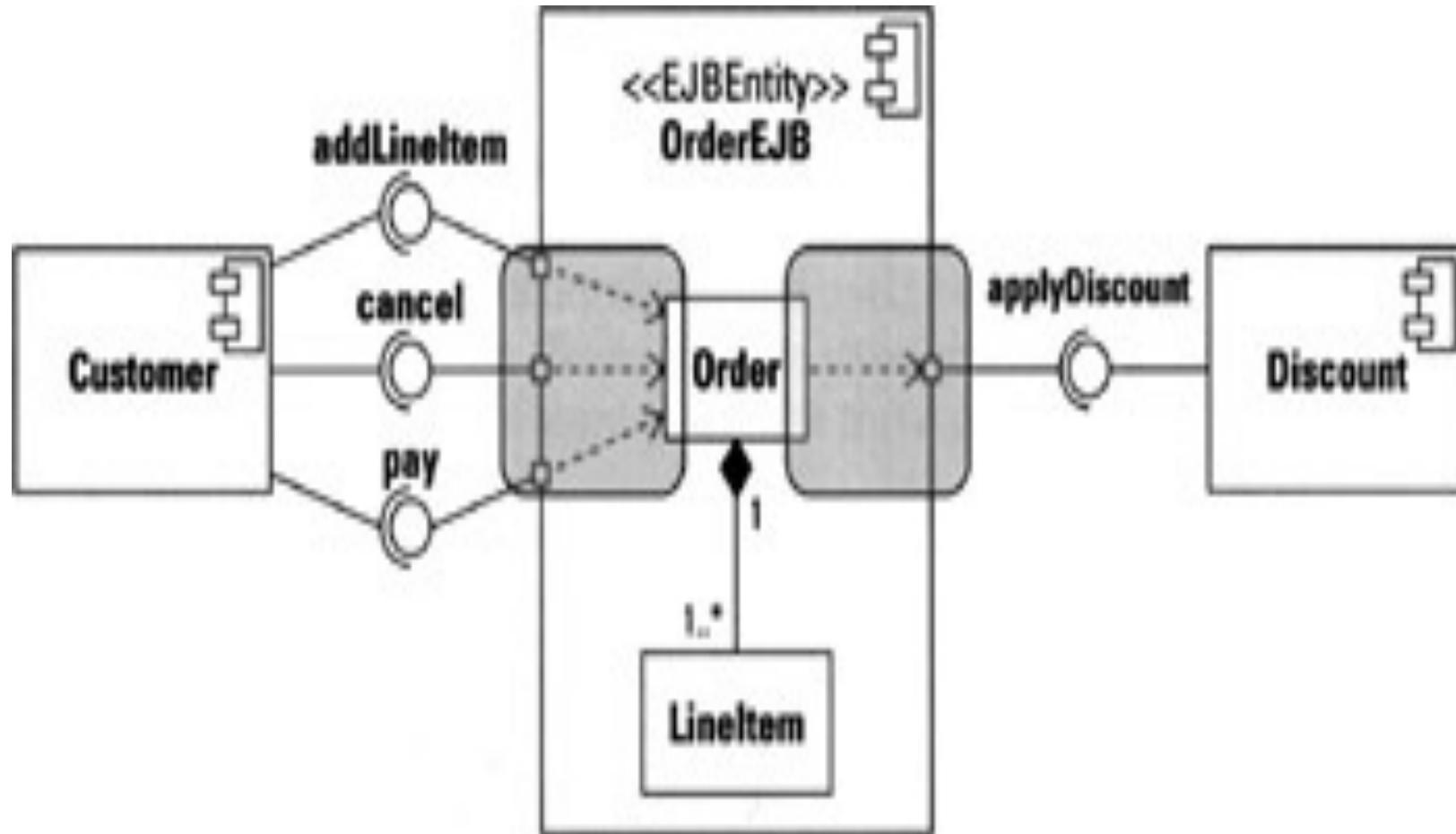
Component diagrams

- UML 2.0 provides a way to represent all of the information defined so far for a component, including interfaces, realizations, and artifacts
 - white box view



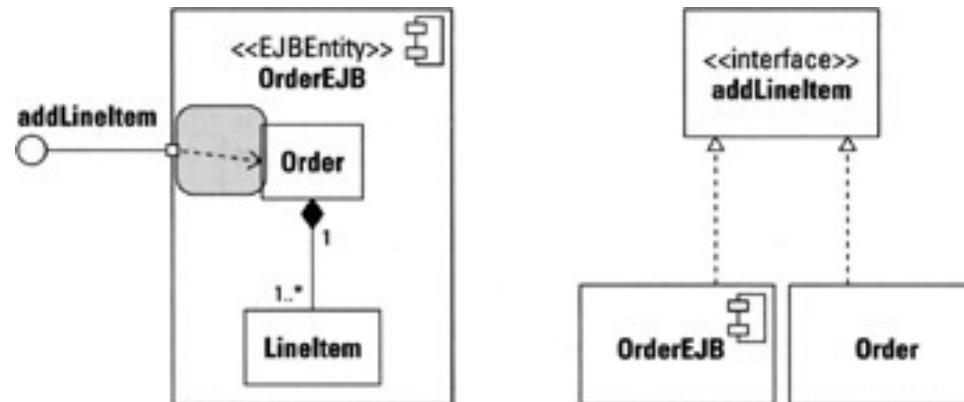
Components diagrams

- **Ports** map the interfaces of the component to the realizations that support them
- A port appears as a small square in the edge of the component

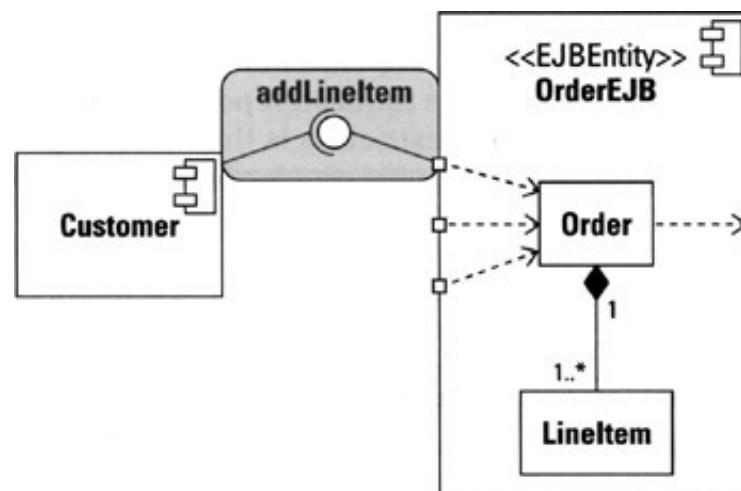


Component diagrams

- A **connector** is a **link** that enables communication between two or more components
 - Two types of connection: delegation connector & assembly connector
- A *delegation connector* maps a request from one port to either another port or to a realization that provides the implementation for the request

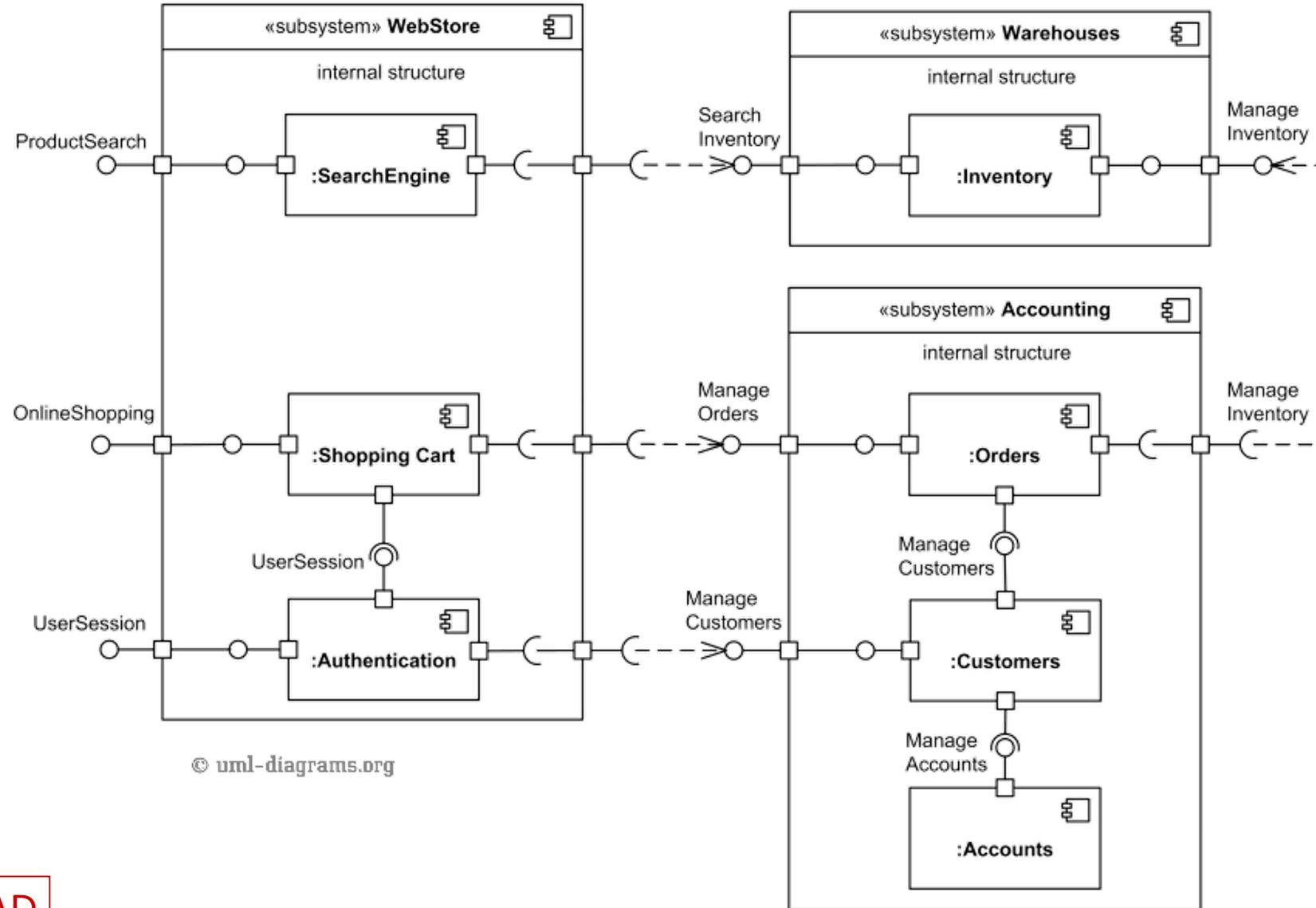


- An *assembly connector* is used to map a required interface to a provided interface



Component diagrams

- Example: Online shopping example with three related subsystems - Webstore, Warehouses and Accounting

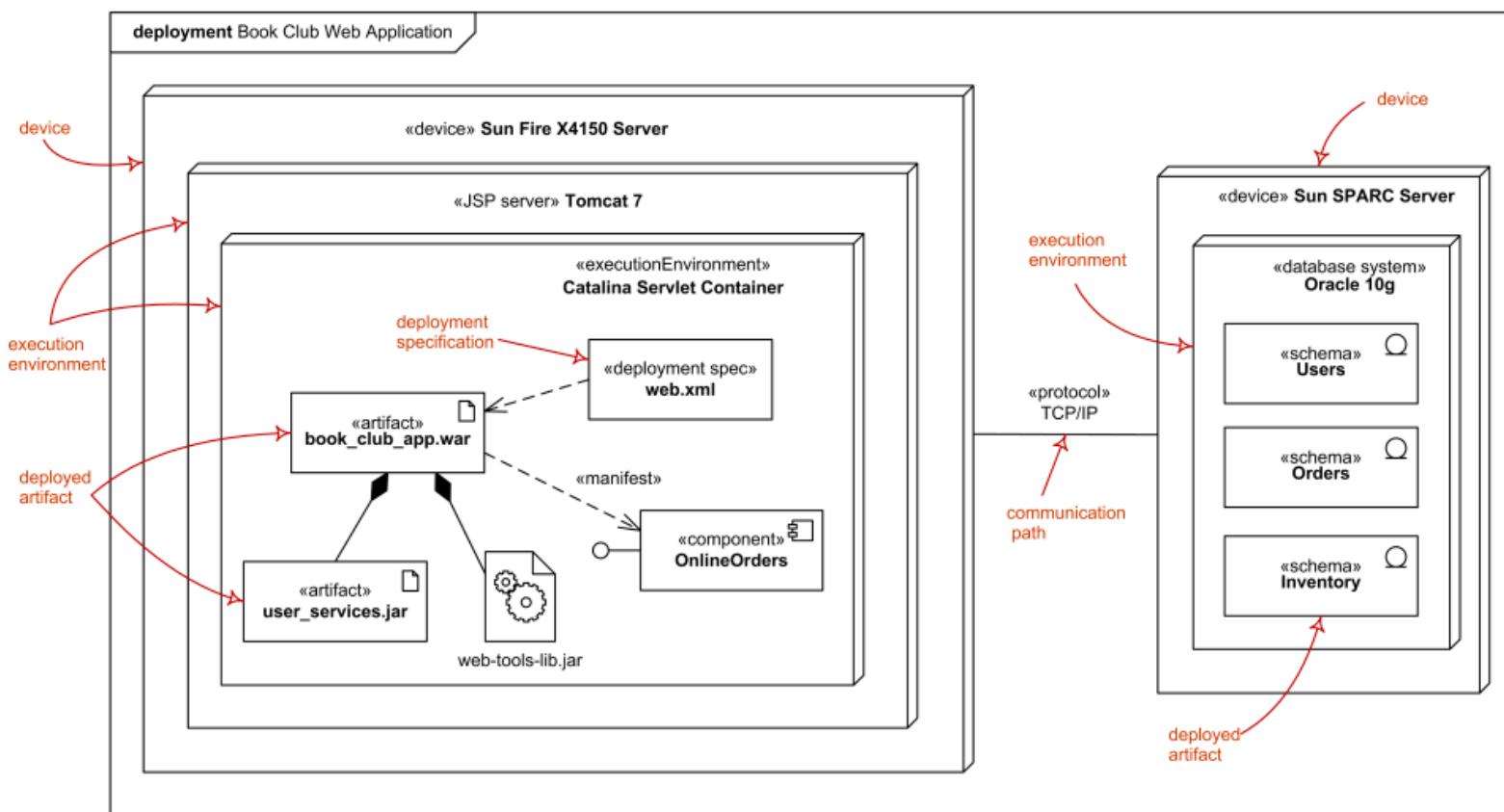


Deployment diagrams

- **Deployment diagram** shows the architectural of a system as deployment (distribution) of software artifacts to deployment targets
 - **Artifacts** represent concrete elements in physical world such as executable files, libraries, archives, database schemas, configuration files, ...
 - Deployment target is usually represented by a **node** which is either hardware device or software execution environment. **Nodes** could be connected
- Deployments diagrams could describe architecture at **specification level** (also call type level) or at **instance level** (similar to class diagrams and object diagrams).

Deployment diagrams

- **Specification level** (also call type level) deployment diagram shows some overview of deployment of artefacts to deployment targets, **without referencing specific instances of artefacts or nodes**.

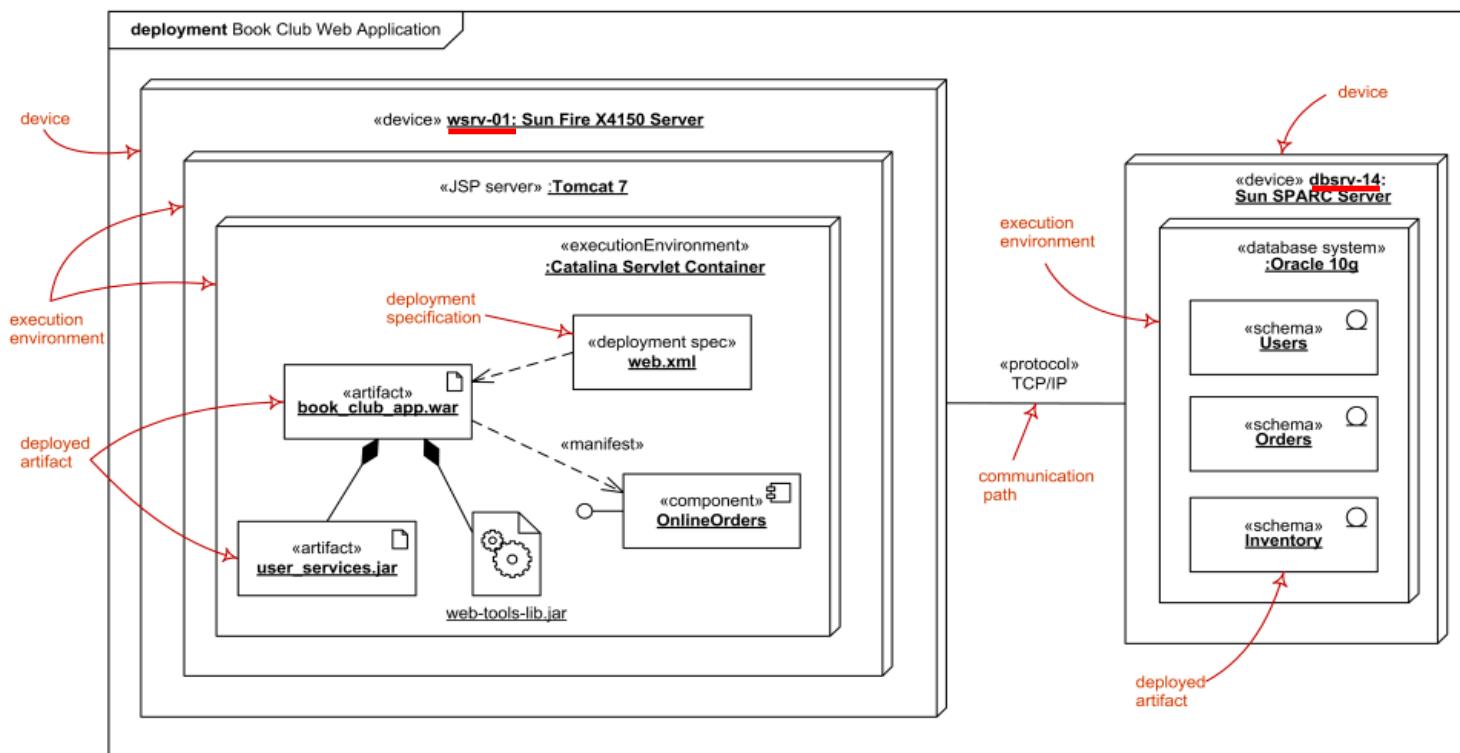


Specification level deployment diagram -

web application deployed to Tomcat JSP server and database schemas-to database system

Deployment diagrams

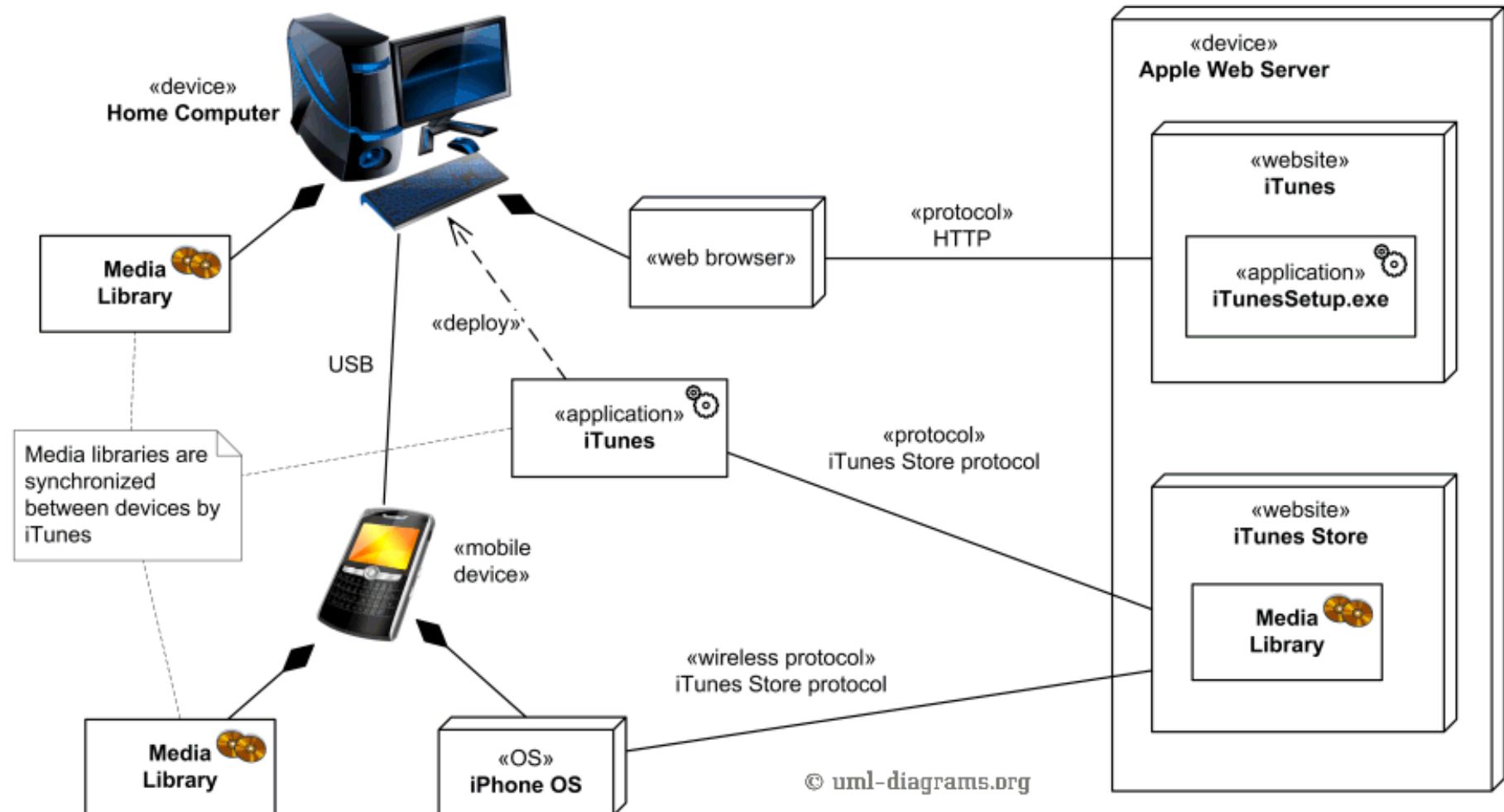
- **Instance level** deployment diagram shows deployment of instances of artefacts to **specific instances of deployment targets**. It could be useful for example to show the differences in deployments to development, staging or production environments with the names/ids specific to deployment services or devices



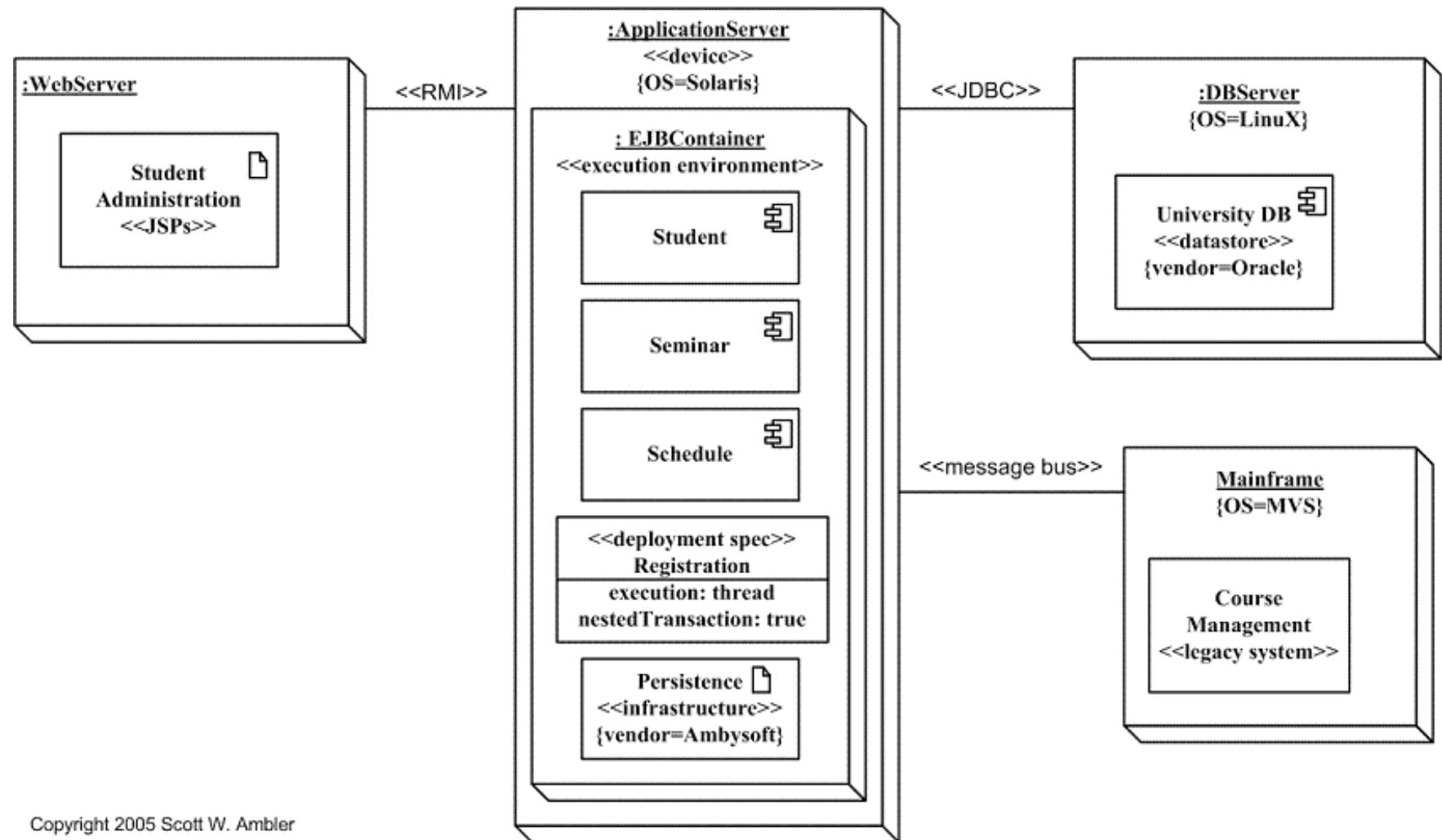
Instance level deployment diagram - web application deployed to Tomcat JSP server and database schemas - to database system

Deployment diagrams - Example

- Deployment diagram for Apple iTunes application



Deployment diagrams - Example

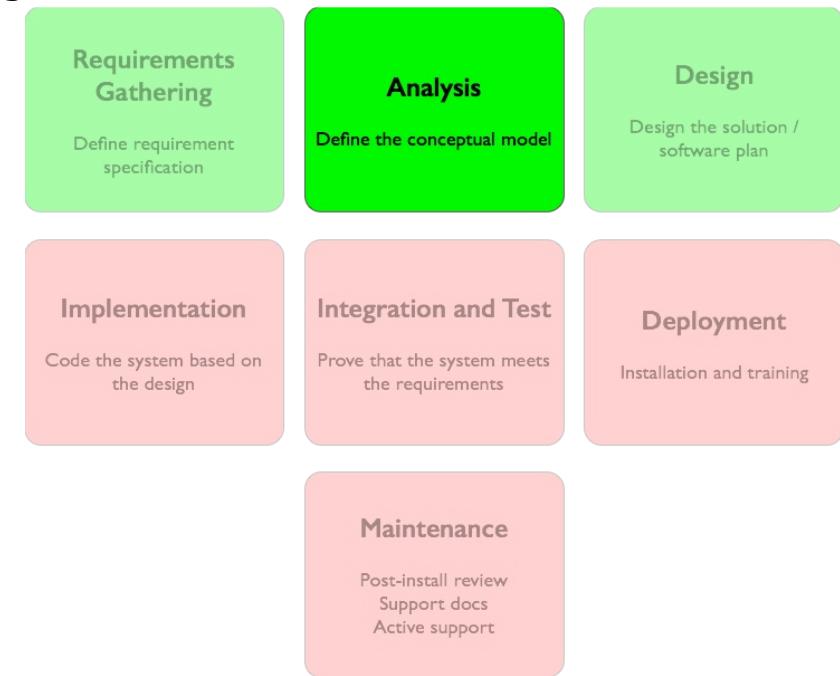


Design principles

- General Responsibility Assignment Software Principles/Patterns – GRASP

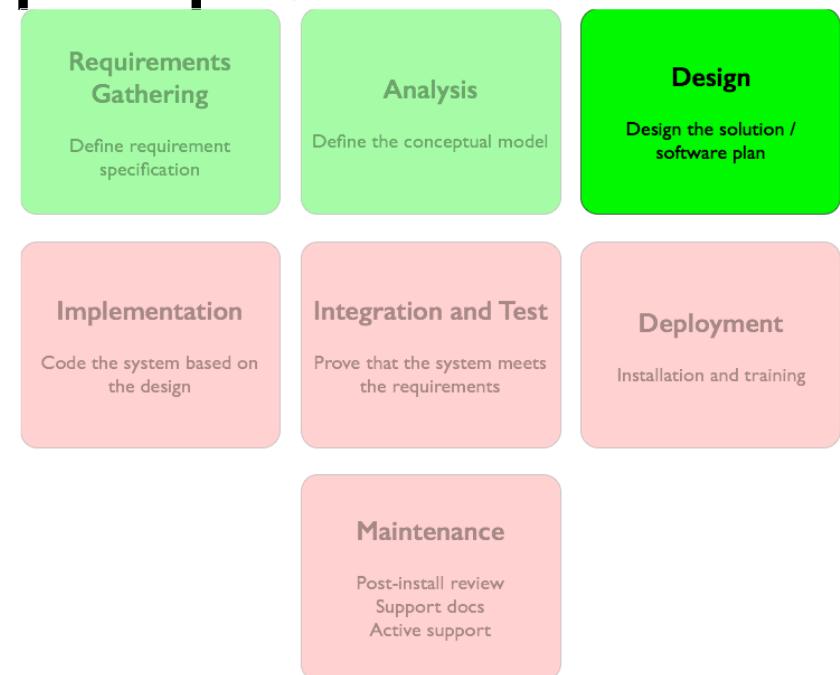
Analysis phase

- The analysis phase provides an understanding of the requirements, concepts and the behaviour of the system.
- Some documents may be obtained at the end of the analysis phase
 - A description of the functionalities
 - A description of the use-cases
 - A description of the conceptual models
 - The system sequence diagrams
 - Activity diagrams



Design phase

- The design phase is to construct diagrams to describe communications between objects and their responsibilities to meet the requirements
- The resulting main diagrams in the design phase are
 - Class diagrams
 - Interaction diagrams
- This phase requires some **design principles**
 - **GRASP design principles**
 - GoF Design patterns



*Understanding responsibilities is key to
object-oriented design.*

Martin Fowler

Responsibilities-Driven Design

- RDD is a metaphor for thinking about object-oriented design.
- Think of software objects similar to people with responsibilities who collaborate with other people to get work done.
- RDD leads to viewing an OO design as a community of collaborating responsible objects.

Responsibilities-Driven Design

```
class House {
    Room[] _rooms;
    double ComputeArea(Room r){
        double area;
        if (r.Shape is Rectangle) // tính diện tích hình chữ nhật
        else if (r.Shape is Triangle) // tính diện tích hình tam giác
        ... và có thể là các hình khác
    }
    void ShowRoomArea(Room r){
        var area = ComputeArea(r);
        Console.WriteLine("Room's area =", area);
    }
    void ShowArea(){
        double area;
        foreach (Room r in _rooms){
            area += ComputeArea(r);
        }
        Console.WriteLine("House's area =", area);
    }
}
```

Responsibilities-Driven Design

```
class Room {  
    double GetArea(){...} // tính diện tích  
}  
  
class House{  
    void ShowRoomArea(Room r)  
    {  
        Console.WriteLine("Room's area =", r.GetArea());  
    }  
    void ShowArea()  
    {  
        double area;  
        foreach (Room r in _rooms){  
            area += r.GetArea(); // gọi method riêng của Room  
        }  
        Console.WriteLine("House's area =", area);  
    }  
}
```

- General Responsibility Assignment Software Patterns or Principles (GRASP)
 - Pattern is a solution which can be applied to a problem in a new context
- A learning aid for OO Design with responsibilities.
- A collection of patterns/principles for achieving good design - patterns of assigning responsibility.

Responsibility

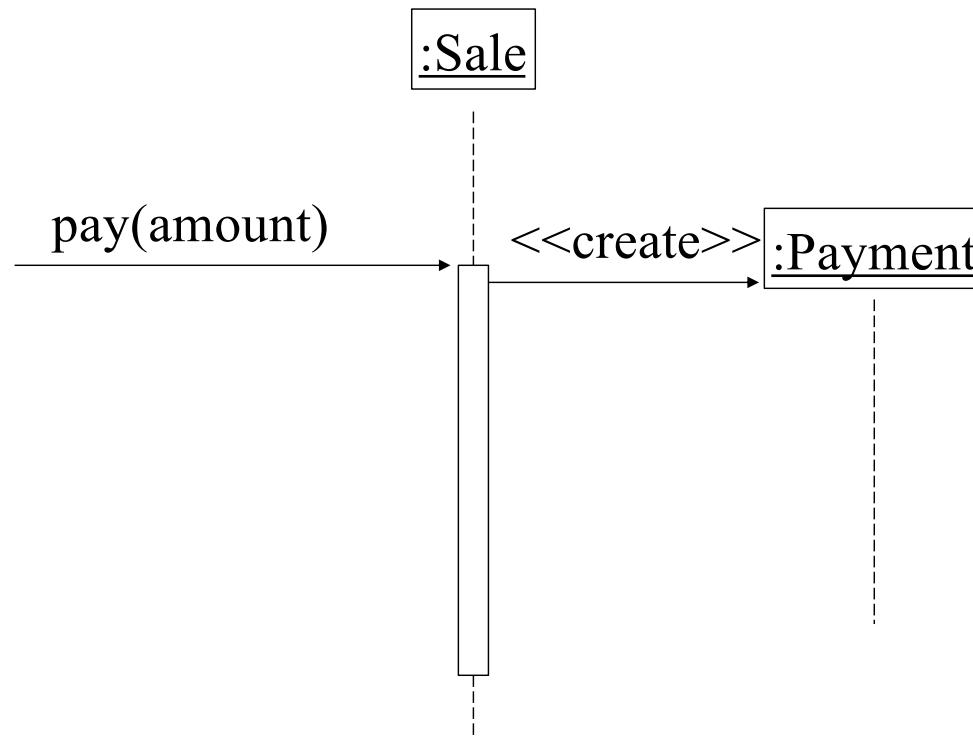
- A **responsibility** is a duty or a contract of a class
- The determination of the attributes and operations of a class is essentially based on its responsibilities
- The responsibilities of an object relate to the behaviour of an object
- Two main types of responsibility
 - **Do**
 - The object accomplishes something itself
 - The object initiates an action of another object
 - The object controls or coordinates activities of other objects
 - **Know**
 - The object knows private encapsulated data
 - The object knows the objects to which it is linked
 - The object has data that it can calculate or derive

Responsibility

- **The responsibilities are assigned to classes during the design phase**
 - Example
 - An object of *Sale* class is responsible for creating an object of *Payment* class (do)
 - An object of *Sale* class is responsible for knowing its total (know).
 - The translation of responsibilities into methods of classes depends on the granularity of the responsibilities
 - A responsibility can be translated by several methods of several classes
 - Responsibility “offer access to the database” can be translated to several methods of several classes
 - A responsibility can be translated by one method
 - Responsibility “create a *Sale*” can be translated by only one method.

Assignment and discovery of responsibilities

- The assignment of responsibilities to objects is very important in object-oriented design.
- **The discovery of responsibilities is achieved when building interaction diagrams**

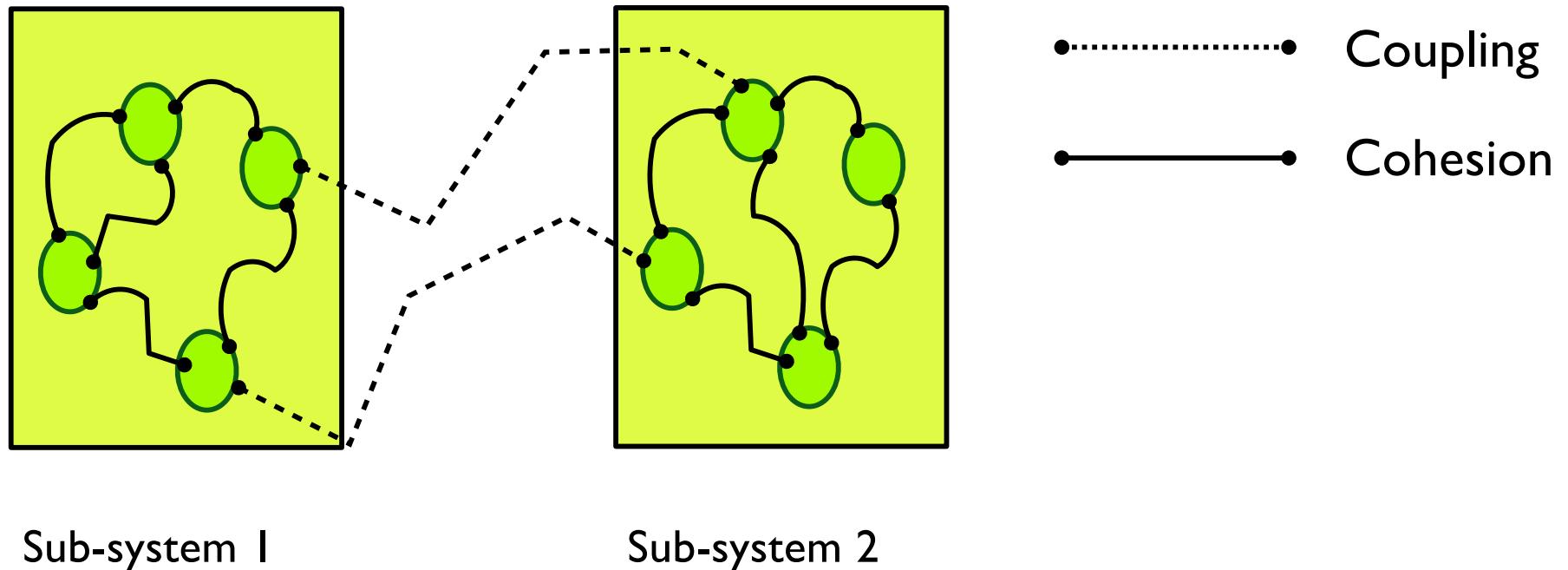


GRASP patterns

- We consider 5 among 9 GRASP patterns/principles
 - **Low Coupling**: assigning responsibilities in a low coupling way
 - **High Cohesion**: assigning the responsibilities to ensure that cohesion remains high
 - **Creator**: assigning the creation responsibility of an object to another object
 - **Information Expert**: the common principle when assigning responsibilities to classes
 - **Controller**: assigning the responsibility for management of the system event messages
 - **Polymorphism**
 - **Indirection**
 - **Pure fabrication**
 - **Protected variations**

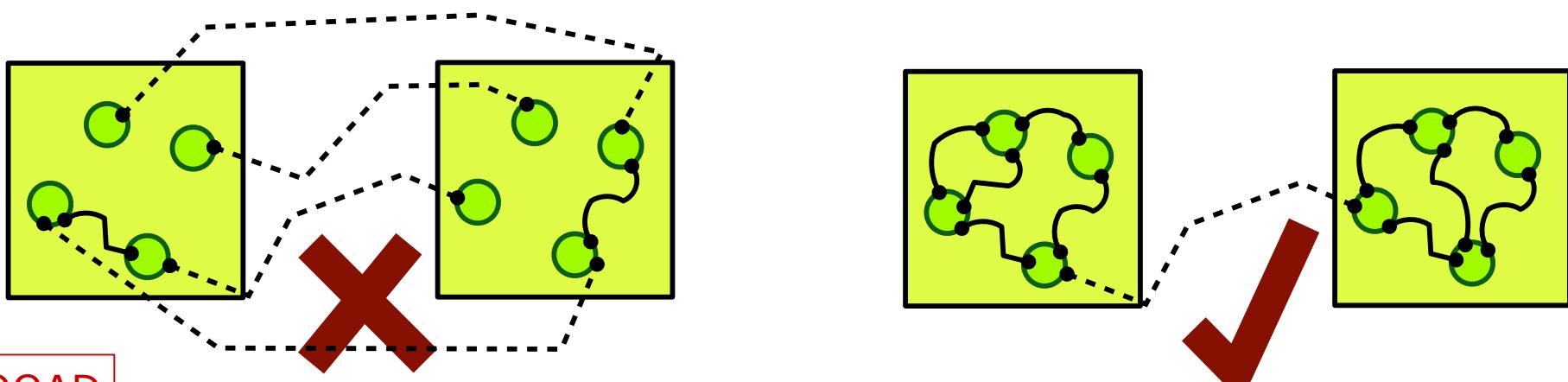
Coupling and Cohesion

- **Coupling:** Amount of relations between objects/sub-systems
- **Cohesion:** Amount of relations within sub-system



Properties of a good architecture

- Minimises coupling between modules
 - Goal: modules don't need to know much about one another to interact
 - Low coupling makes future change easier
- Maximises cohesion within modules
 - Goal: the content of each module are strongly inter-related
 - High cohesion makes a module easier to understand



Low coupling

- Problem: How to support low dependency, low change impact, and increase reuse?
- Coupling:
 - Measure how strongly one element is connected to, has knowledge of or relies on other elements
 - An element with low (or weak) coupling is not dependent on too many other elements

When are two classes coupled?

- Common forms of coupling from TypeX to TypeY
 - TypeX has an attribute that refers to a TypeY instance
 - A TypeX object calls on services of TypeY object
 - TypeX has a method that references an instance of TypeY (parameter, local variable, return type)
 - TypeX is a direct or indirect subclass of TypeY
 - TypeX is an interface and TypeY implements that interface

High coupling (Bad)

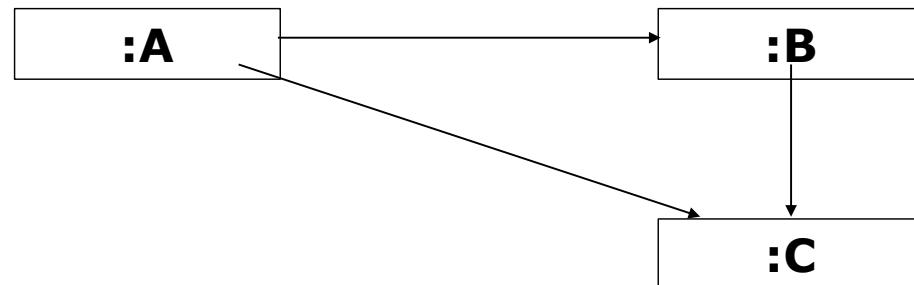
- A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable and suffer from the following problems:
 - Force local changes because of changes in related classes
 - Harder to understand in isolation
 - Harder to reuse because its use requires the additional presence of the classes on which it is dependent

Solution

- Assign responsibility so that coupling remain low
- Use this principle to evaluate alternatives

Low Coupling pattern

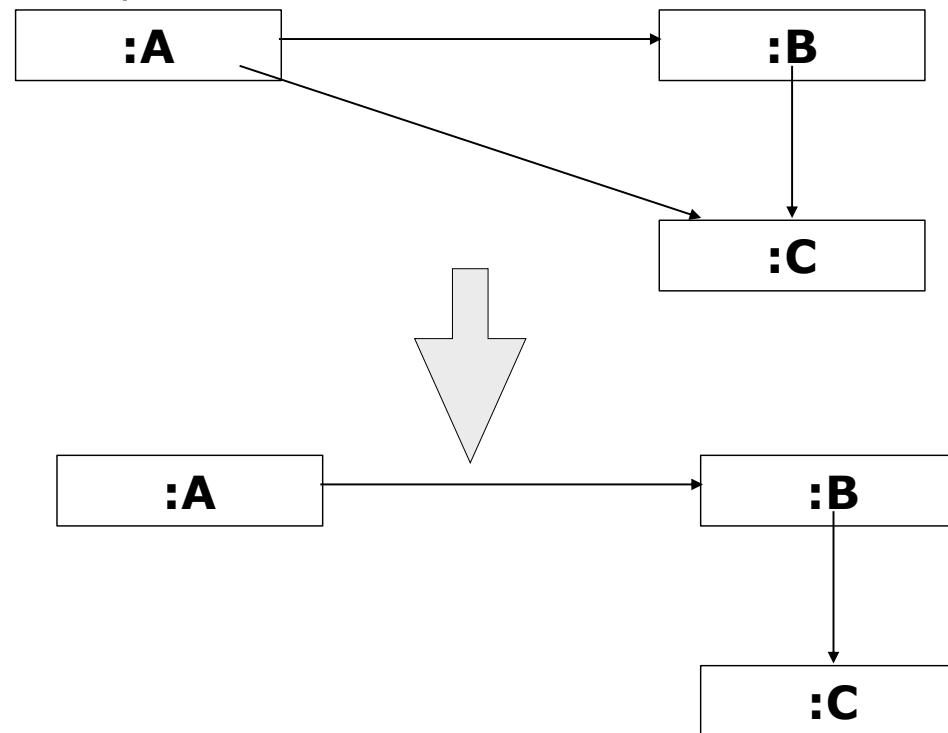
- Coupling
 - the dependency between objects



- When depended upon element changes, it affects the dependant also
 - Two elements are coupled, if
 - One element has aggregation/composition association with another element
 - One element implements/extends other element
 - A class has a low coupling if it is not dependent on too many other classes

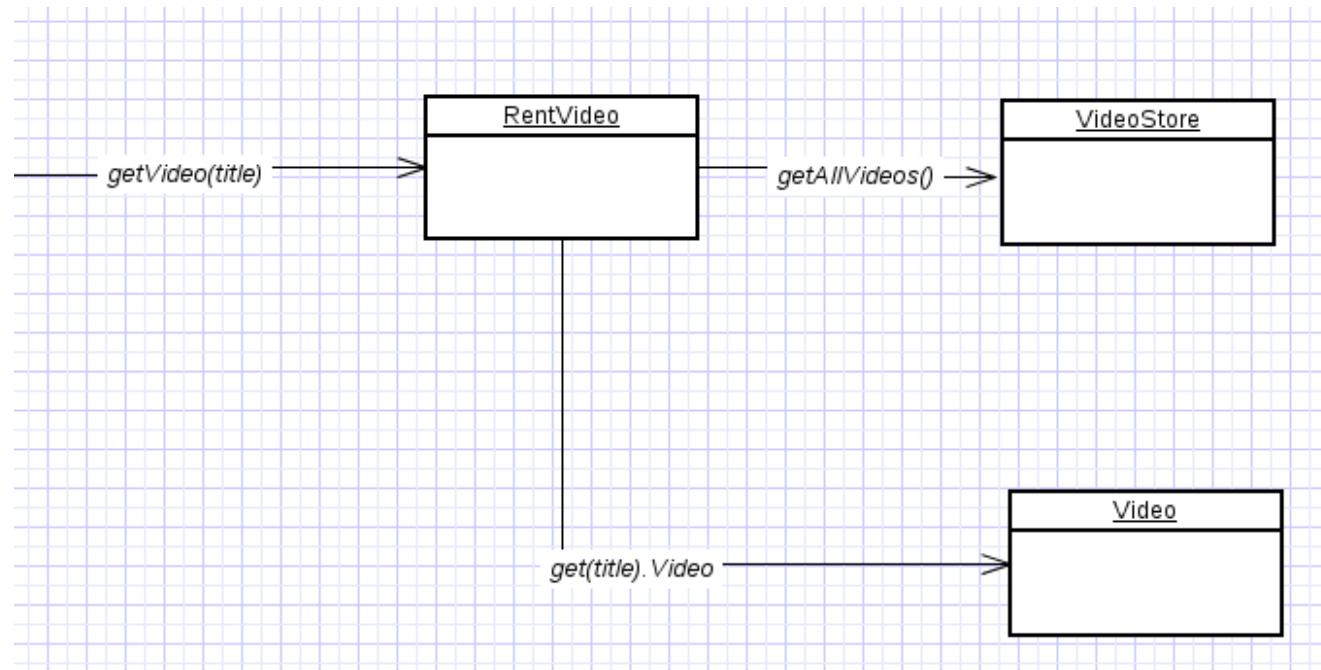
Low Coupling pattern

- Problem
 - How can we reduce the impact of change in depended upon elements on dependant elements?
- Solution
 - Assign responsibilities so that coupling remain low
 - Minimise the dependency hence making system maintainable, efficient and code reusable



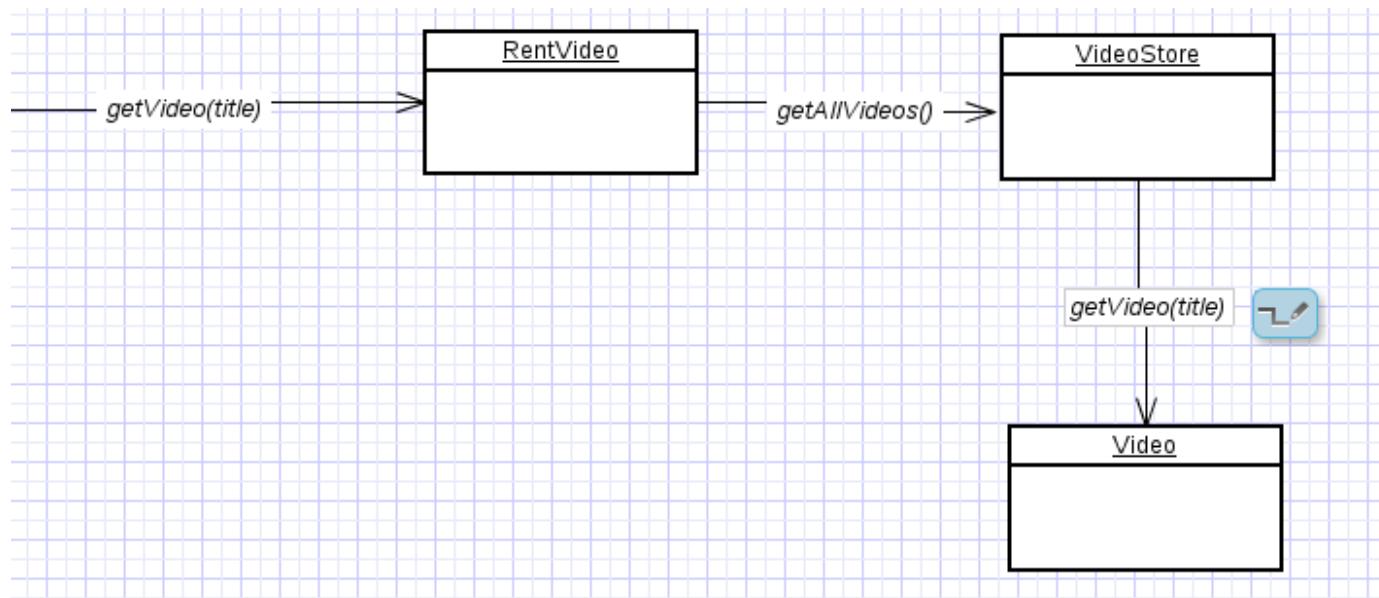
Low Coupling pattern

- Two elements are coupled, if
 - One element has aggregation/composition association with another element.
 - One element implements/extends other element.
- Example for poor coupling
 - class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes.



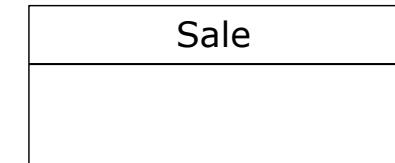
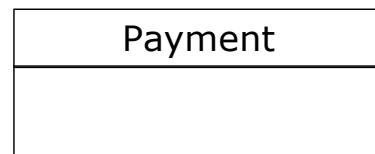
Low Coupling pattern

- Two elements are coupled, if
 - One element has aggregation/composition association with another element.
 - One element implements/extends other element.
- Example for poor coupling
 - VideoStore and Video class are coupled, and Rent is coupled with VideoStore. Thus providing low coupling.



Example

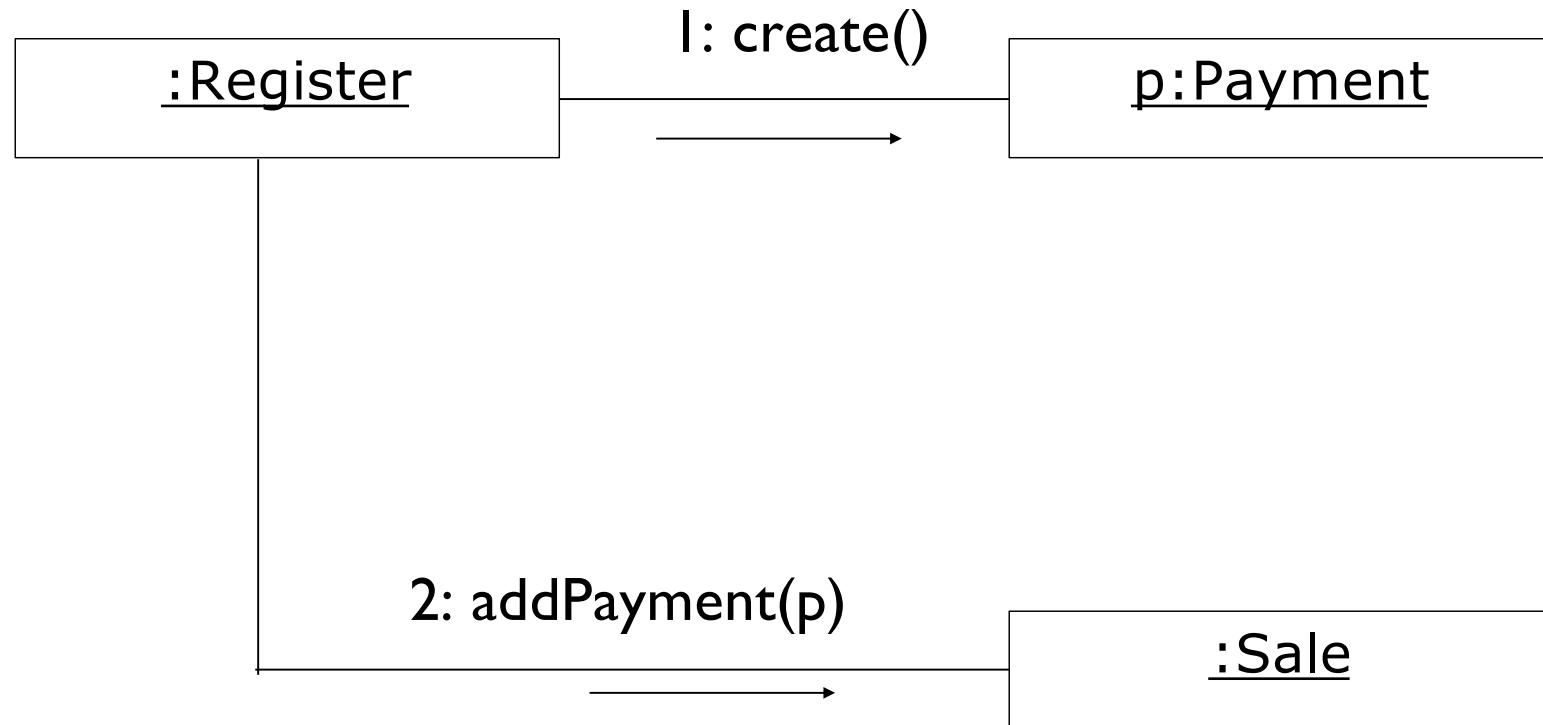
- We have three following class in the Cash Register system



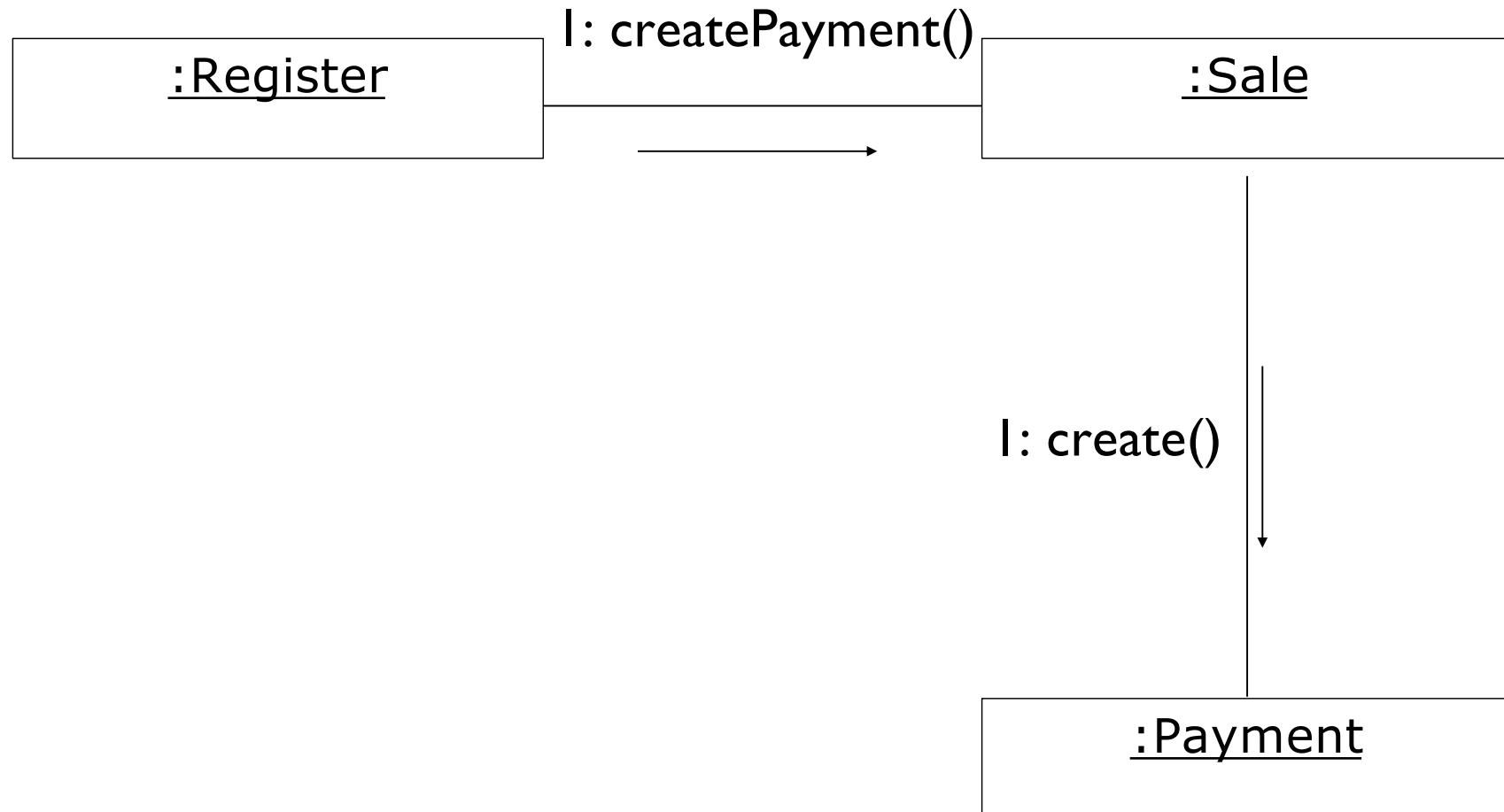
- Supposing that we would like to create an instance of Payment and associate it with Sale.
- How can we assign responsibilities to adhere to Low Coupling pattern?



Solution 1



Solution 2

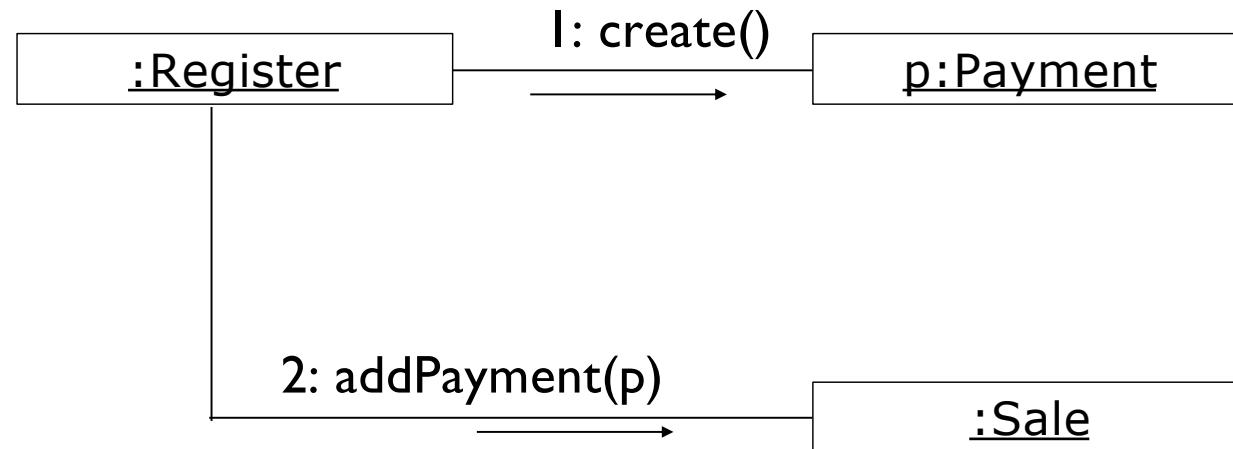


Which solution is better?

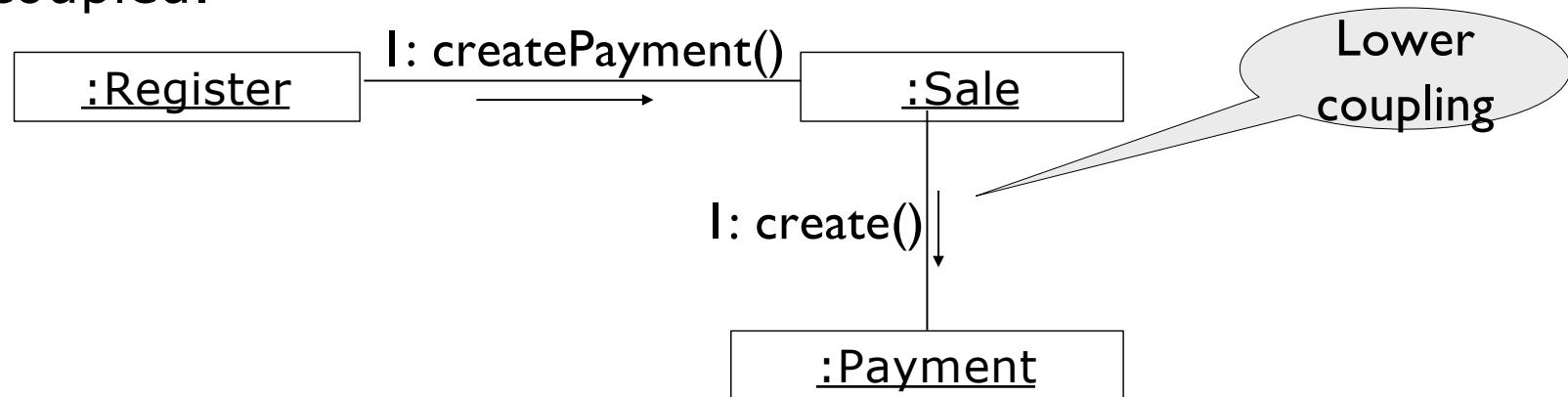
- Assume that each *Sale* will eventually be coupled with a *Payment*.
- Solution 1 has the coupling of Register and Payment, which is absent in Solution 2
- Solution 2 therefore has lower coupling
- Note that two patterns - Low Coupling and Creator - suggest different solutions
- Do not consider patterns in isolation

Solutions

- Solution 1: *Register* knows both *Payment* and *Sale*. *Register* depends on both *Payment* and *Sale*.

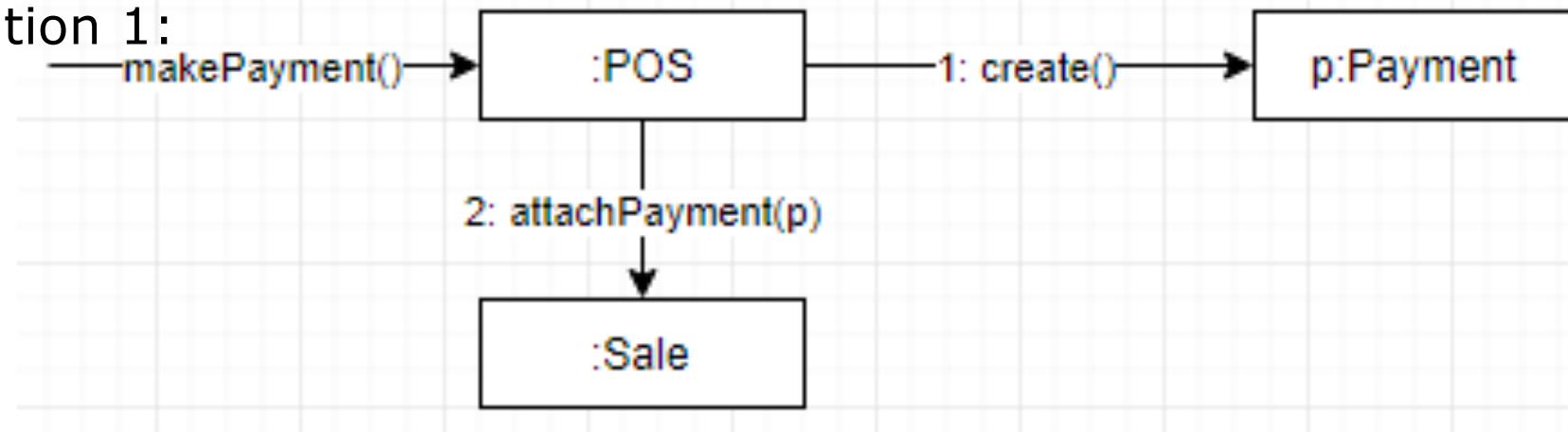


-
- Solution 2: *Register* and *Sale* are coupled, *Sale* and *Payment* are coupled.

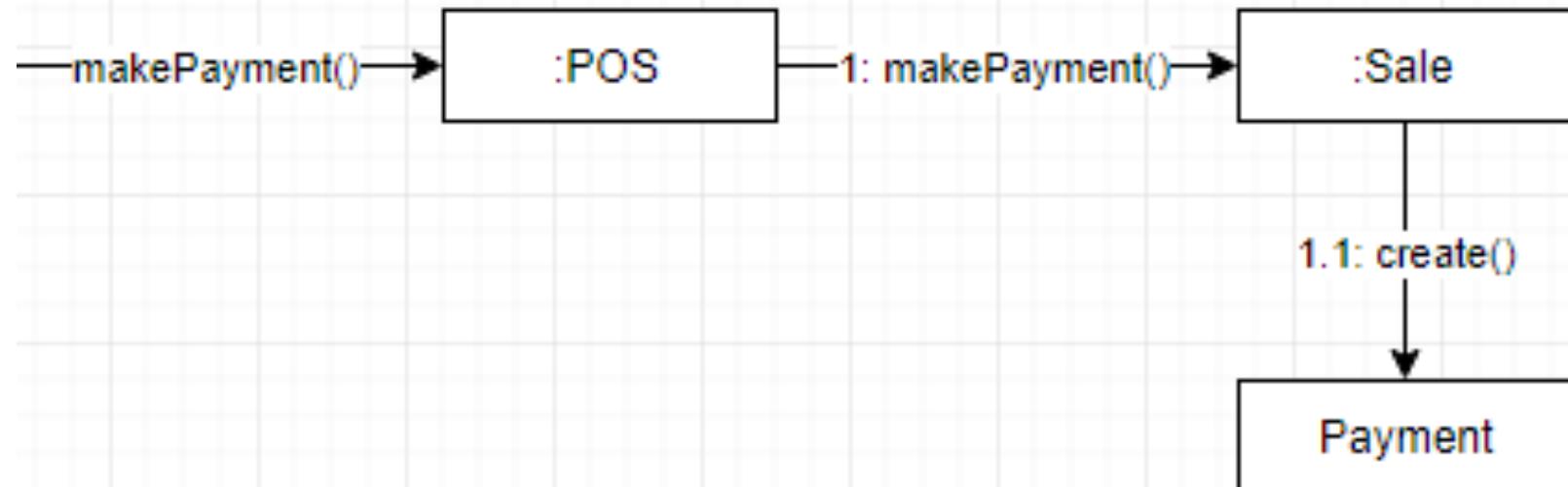


Example 2

- Solution 1:



- Solution 2:



High Cohesion pattern

- Problem
 - How to ensure that the operations of any element are functionally related?
- Solution
 - Clearly define the purpose of the element
 - Gather related responsibilities into an element
- Benefit
 - Easily to understand and maintain
 - Code reuse
 - Low coupling

High Cohesion pattern

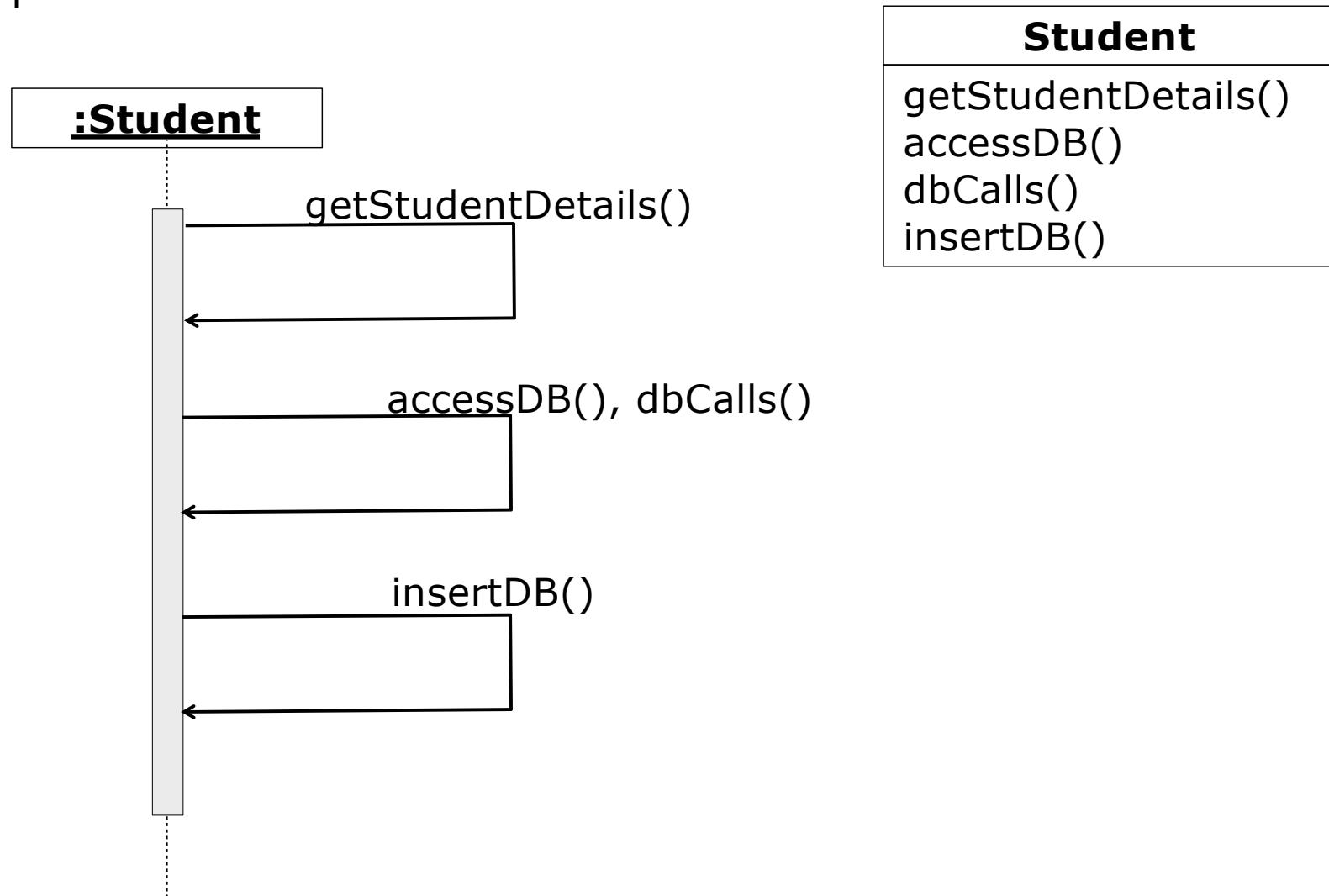
- Ví dụ:
 - responsibilities of Car:
 - (a) *start engine*
 - (b) *run engine*
 - (c) *drive*

Low cohesion

- A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:
 - hard to comprehend
 - hard to reuse
 - hard to maintain
 - constantly affected by change

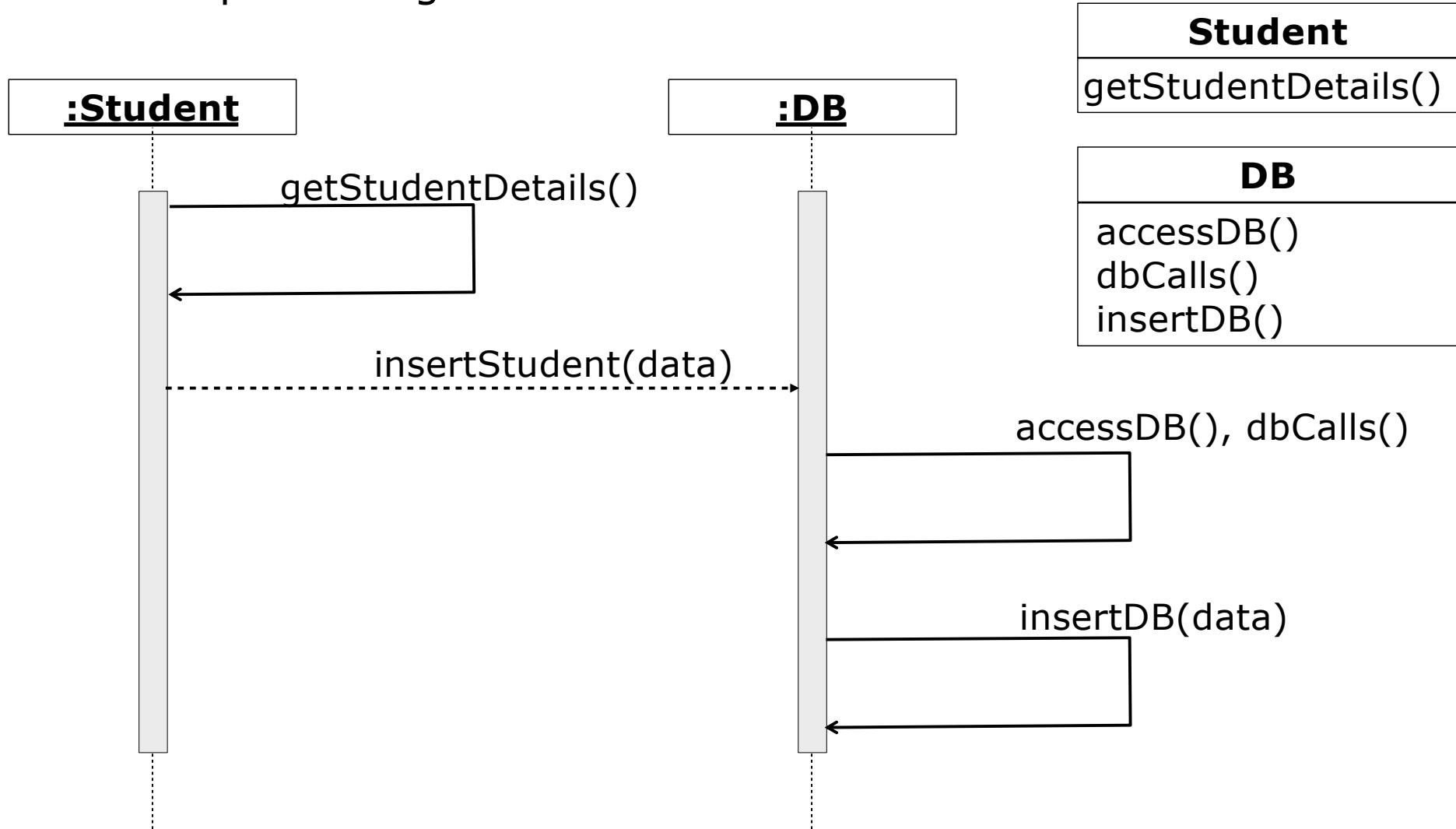
High Cohesion pattern

- Example for Low Cohesion



High Cohesion pattern

- Example for High Cohesion

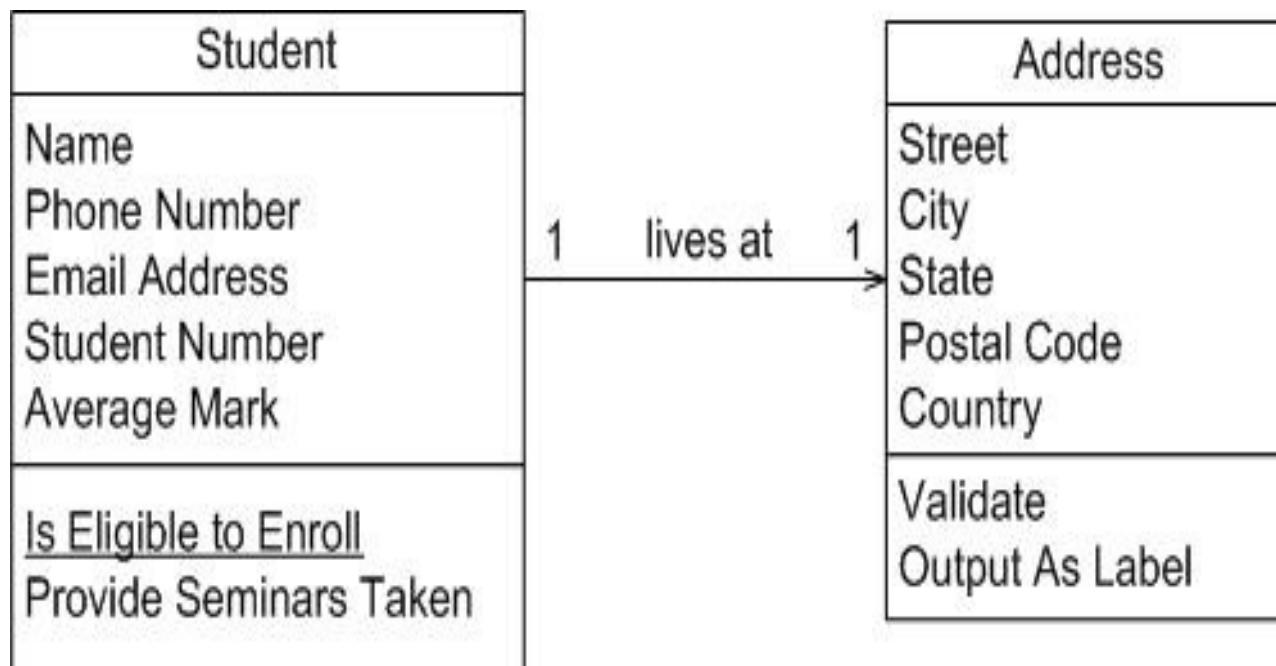


Rules of thumb

- For high cohesion, a class must
 - have few methods
 - have a small number of lines of code
 - not do too much work
 - have high relatedness of code

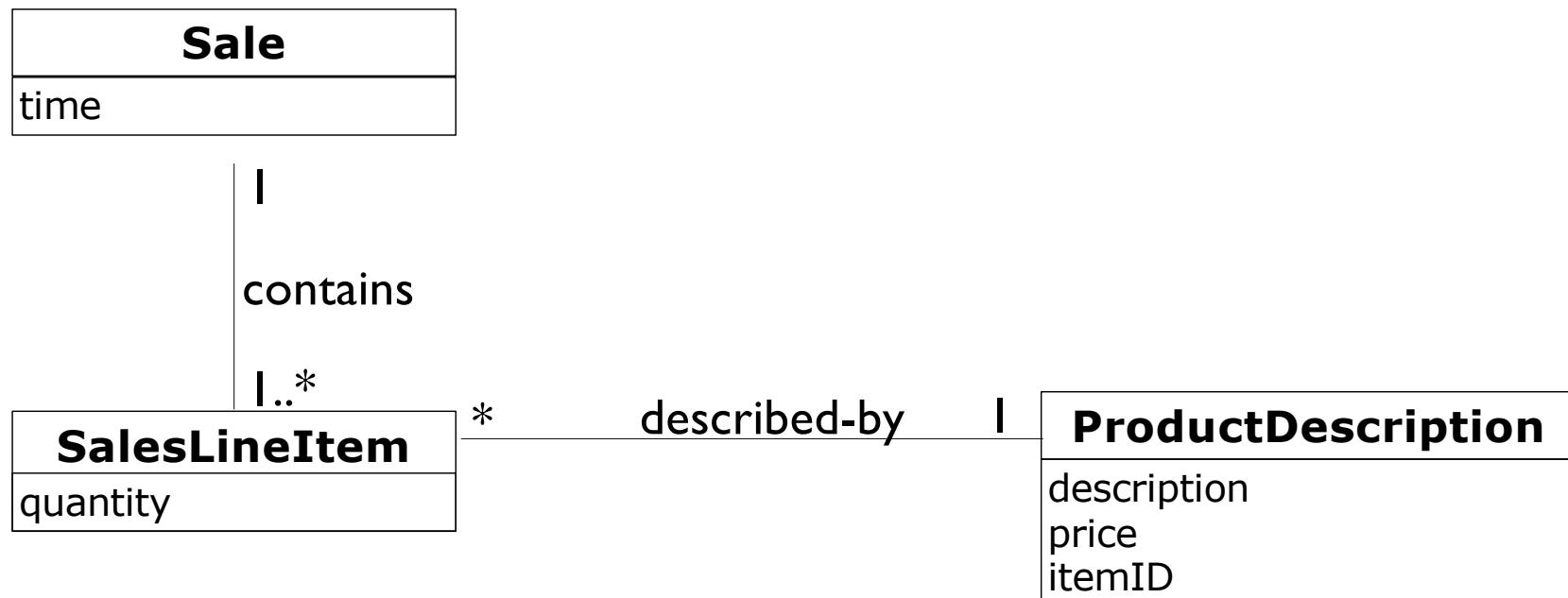
□ TODO

<http://www.agilemodeling.com/artifacts/classDiagram.htm>
example if low and high cohesion



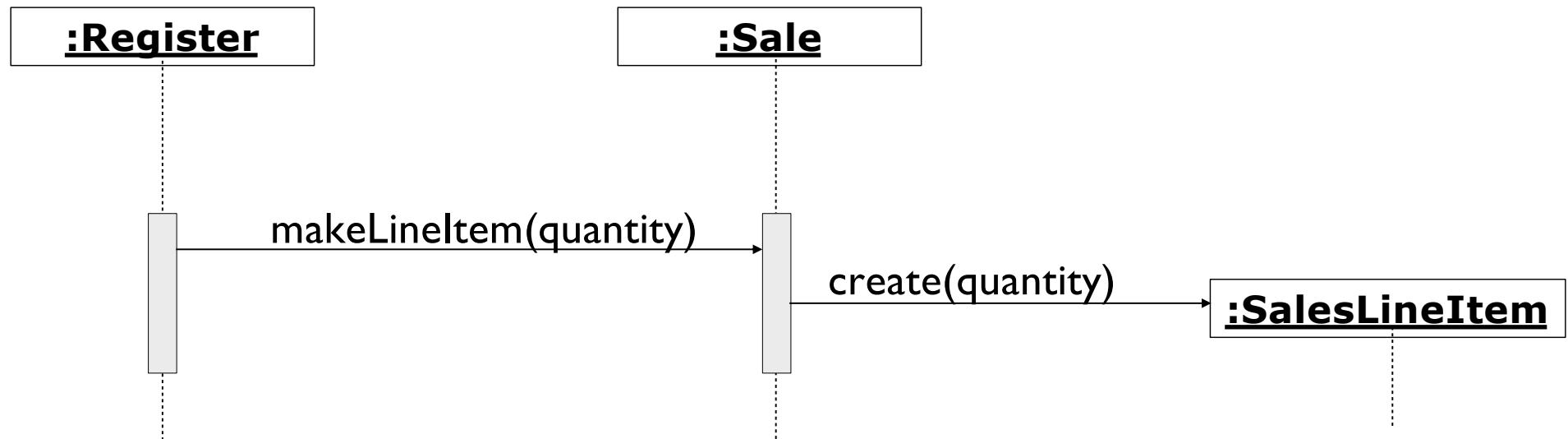
“creator” pattern

- Problem
 - Who is responsible for creating objects/instances of a class?
- Example
 - Who should be responsible for creating a SalesLineItem instance?



“creator” pattern

- Example (continue)
 - *Sale* contains *SalesLineItem*, so *Sale* should be responsible for creating objects of *SalesLineItem*



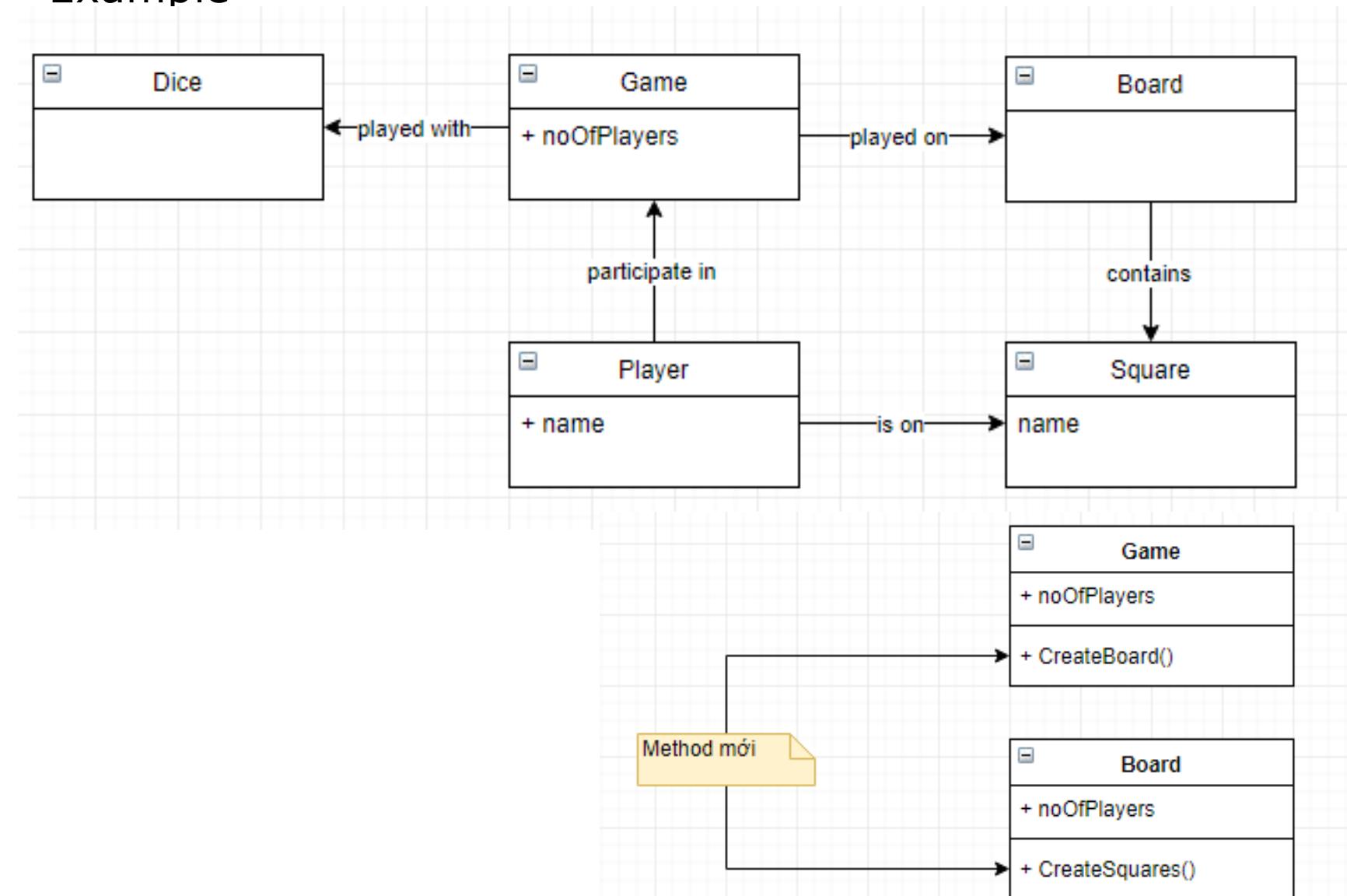
- “`makeLineItem(quantity)`” method will be introduced to *Sale* class

“creator” pattern

- Discussion
 - Basic idea is to find a creator that needs to be connected to the created object in any event
 - Also need initialisation data to be nearby - sometimes requires that it is passed into client. e.g., *ProductionDescription* needs to be passed in.
- Assign class B the responsibility to create an instance of class A if one of these is true
 - B contains A
 - B aggregates A
 - B has data for initialising A
 - B closely uses A

“creator” pattern

□ Example



“creator” pattern

- Application
 - Guide in the assigning responsibility for creating objects
 - Help to find the class who is responsible for creating objects

- Advantages
 - The “creator” pattern supports the low coupling between classes
 - Fewer dependencies and more reusability
 - The coupling is not increased because the created class is visible to the “creator” class

Information Expert pattern

- Problem
 - What is the general principle of assigning responsibilities to objects?
 - Consider that there may be 100s or 1000s of classes
 - To which ones do we assign a particular functionality?
 - Assigning well makes our design easier to understand, maintain, extend and reuse.
- Solution
 - Assign responsibility to the information expert - the class that has the information to fulfil the responsibility
- Application
 - One of the most used patterns in object-oriented design
 - Accomplishing of a responsibility can request information distributed among several objects or classes, this implies several “partial experts” working together to fulfil the responsibility

Information Expert pattern

- Example
 - In the *CashRegister* system, who is responsible for knowing the grand total of a *Sale*?



Information Expert pattern

- Example: Responsibilities



Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line items subtotal
ProductDescription	knows product price

Information Expert pattern

- Example (continue)
 - To calculate **grand total** of a *Sale*, it is necessary to know the instances of *SalesLineItem* and the sub-total of each instance.
 - According to the pattern, *Sale* knows the information



Information Expert pattern

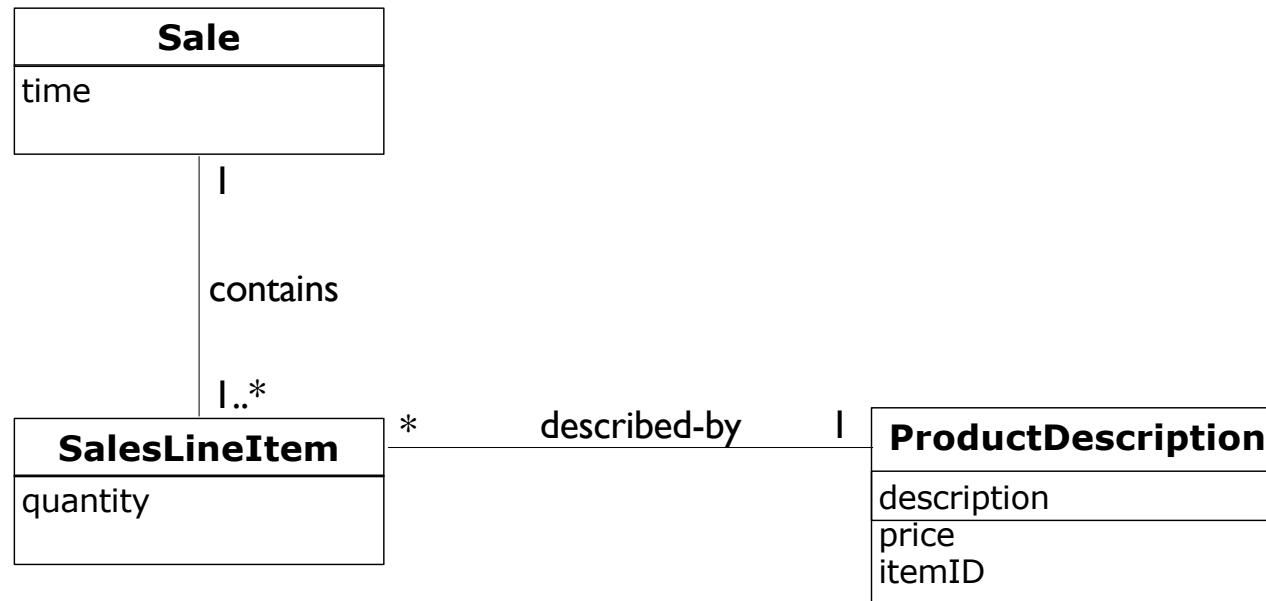
- Example (continue)
 - Introduce “*getTotal()*” method to *Sale* class



Information Expert pattern

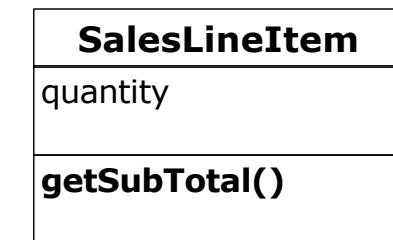
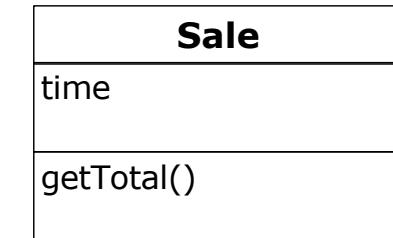
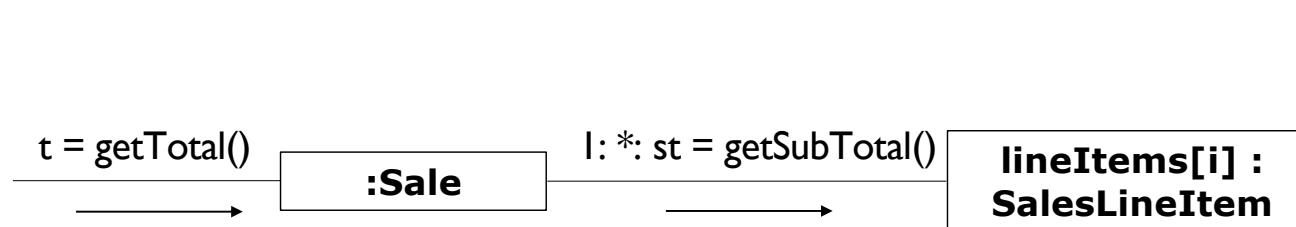
- Example

- Then, we need to determine the sub-total of each *SalesLineItems*. To do so, we need to know the number of *ProductDescription*
- According to the pattern, *SalesLineItem* is the expert.



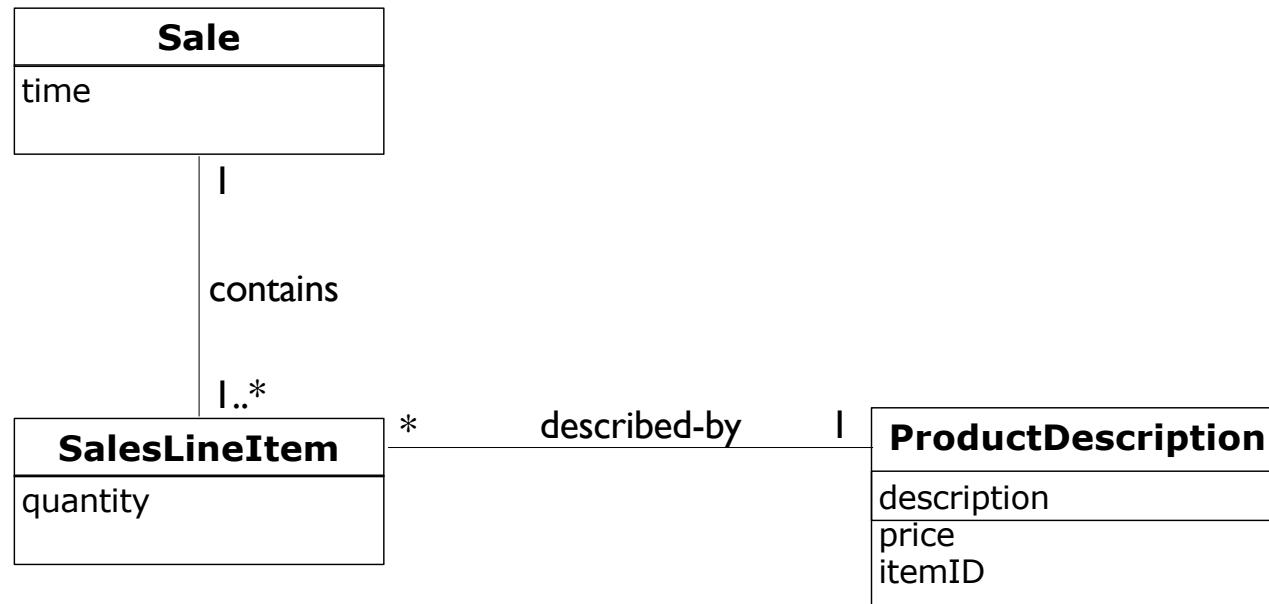
Information Expert pattern

- Example
 - Introduce the "getSubTotal()" method to *SalesLineItem* class



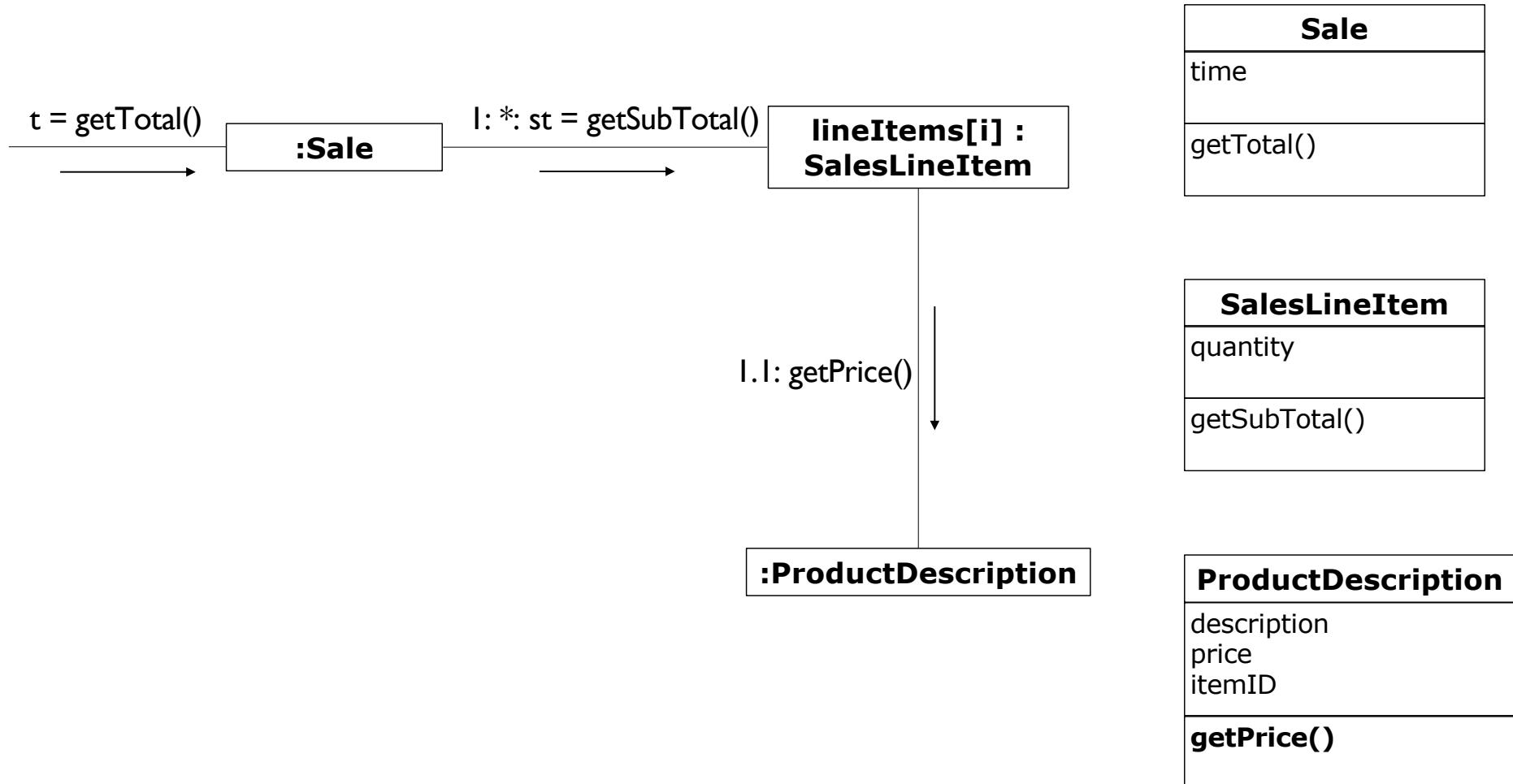
Information Expert pattern

- Example
 - To calculate the sub-total, *SalesLineItem* needs to know the price of each product.
 - *ProductionDescription* est expert.



Information Expert pattern

- Example
 - Introduce the “`getPrice()`” method to `ProductDescription` class

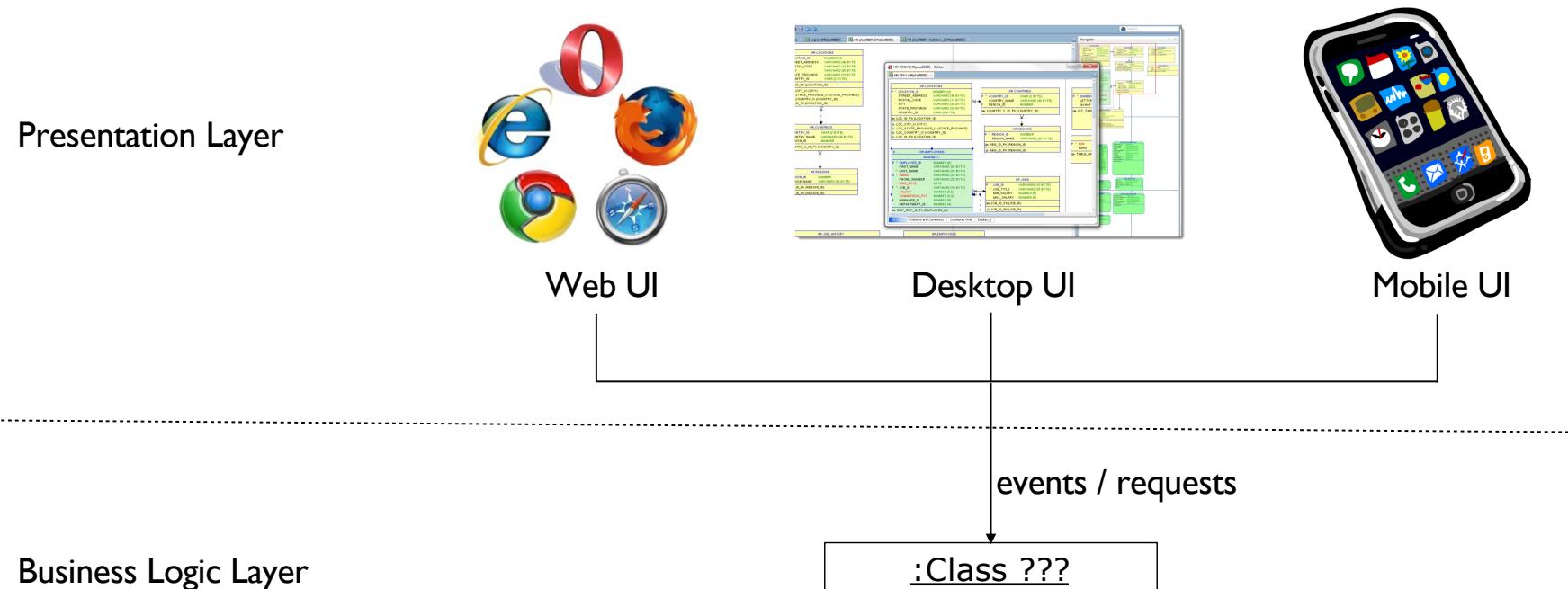


Information Expert pattern

- Advantages
 - The encapsulation is maintained since objects use their own information to satisfy responsibility
 - This pattern supports loose coupling, this allows the system to be more robust and easier to maintain
 - The behaviour is distributed among the classes that possess the necessary information, it encourages more coherent and smaller definitions are easier to understand and maintain

Controller pattern

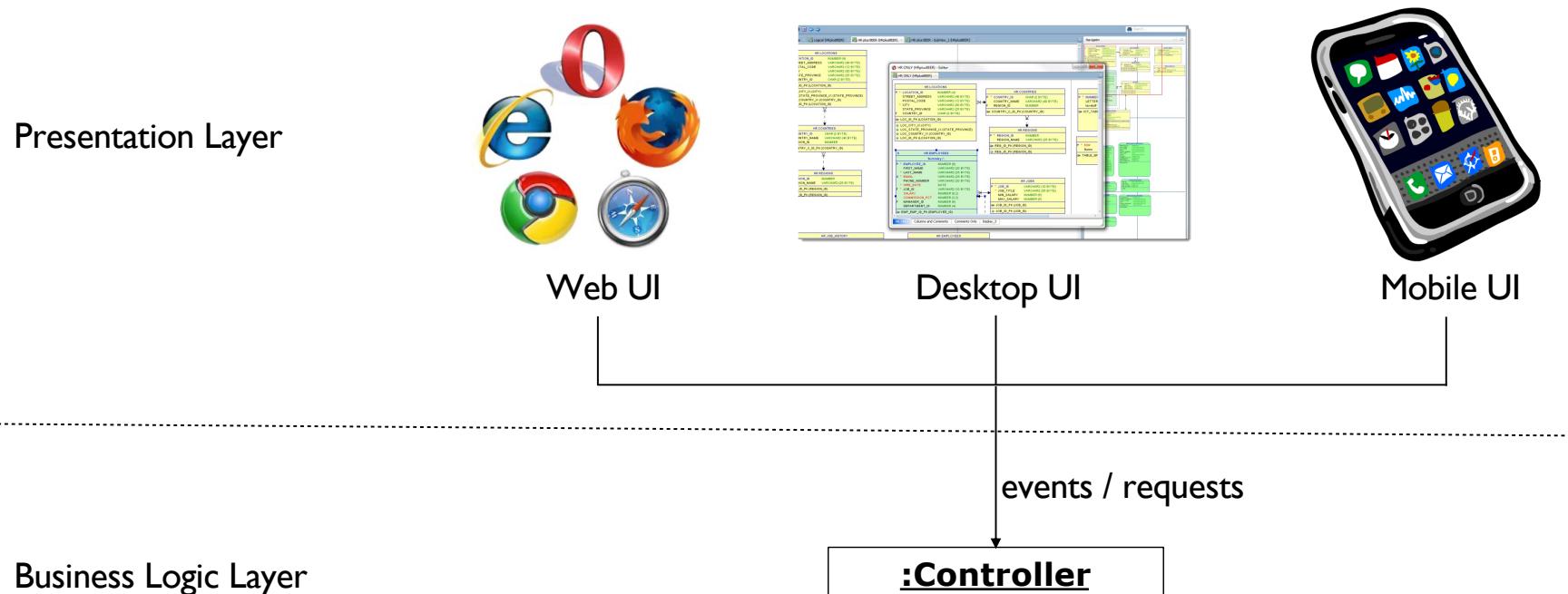
- Problem
 - Which first object beyond the User Interface (UI) layer receives and coordinates ("controls") a system operation?



Controller pattern

□ Solution

- A **Controller** is the first object beyond the UI layer that is responsible for receiving and handling a system operation.
- A controller should delegate the work to other objects. The controller only receives the requests but doesn't not actually solve them.

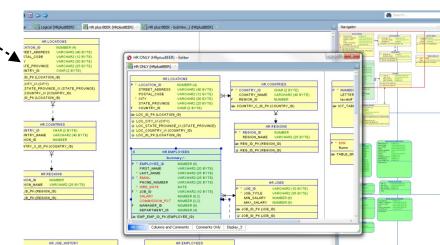


Controller pattern

- Application
 - The Controller pattern can be applied to all the systems that need to process external events
 - A controller class is selected to process the events
- Example
 - The Cash Register system has several events



Web UI

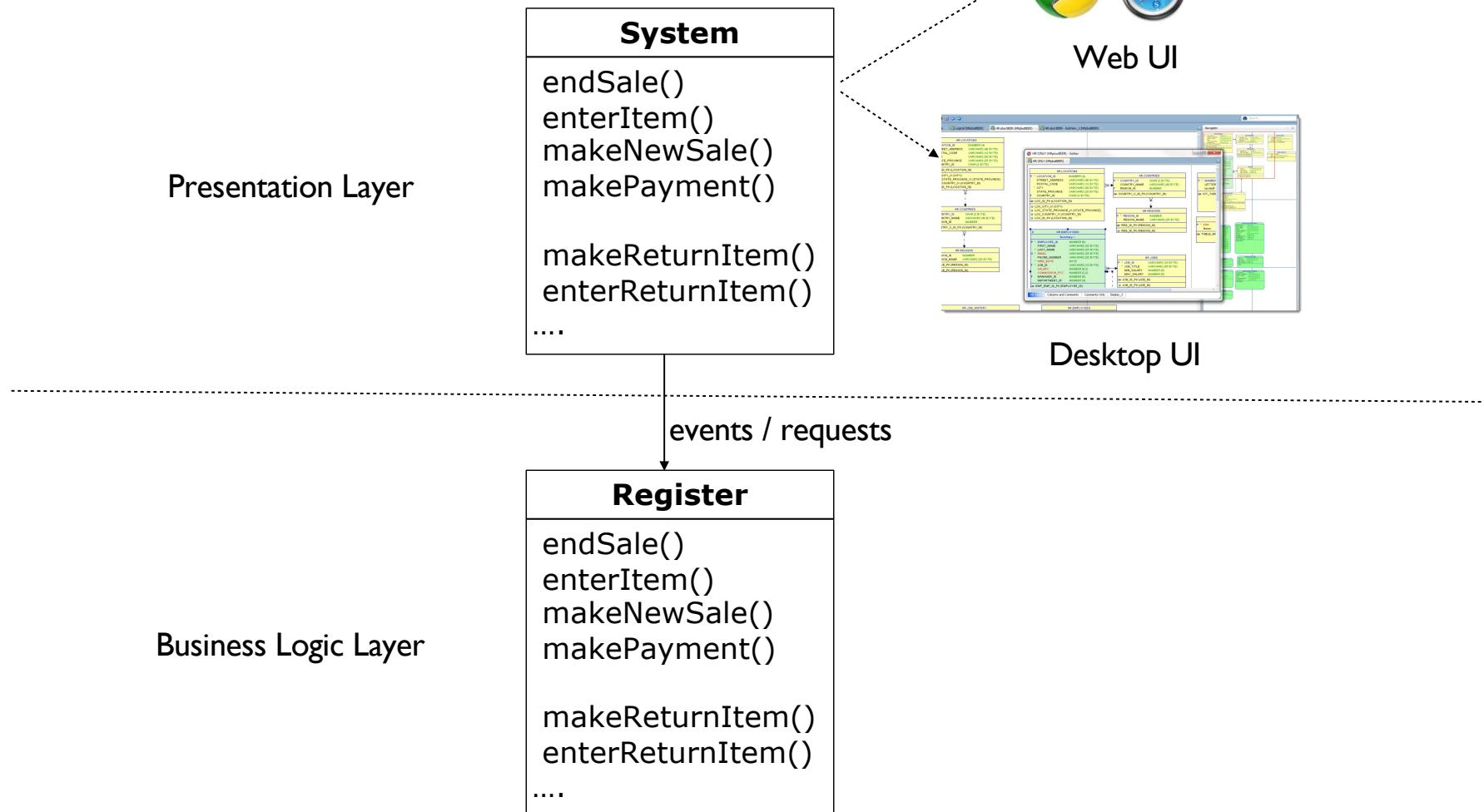


Desktop UI

- What class can be the controller (i.e., what class processes the events)?

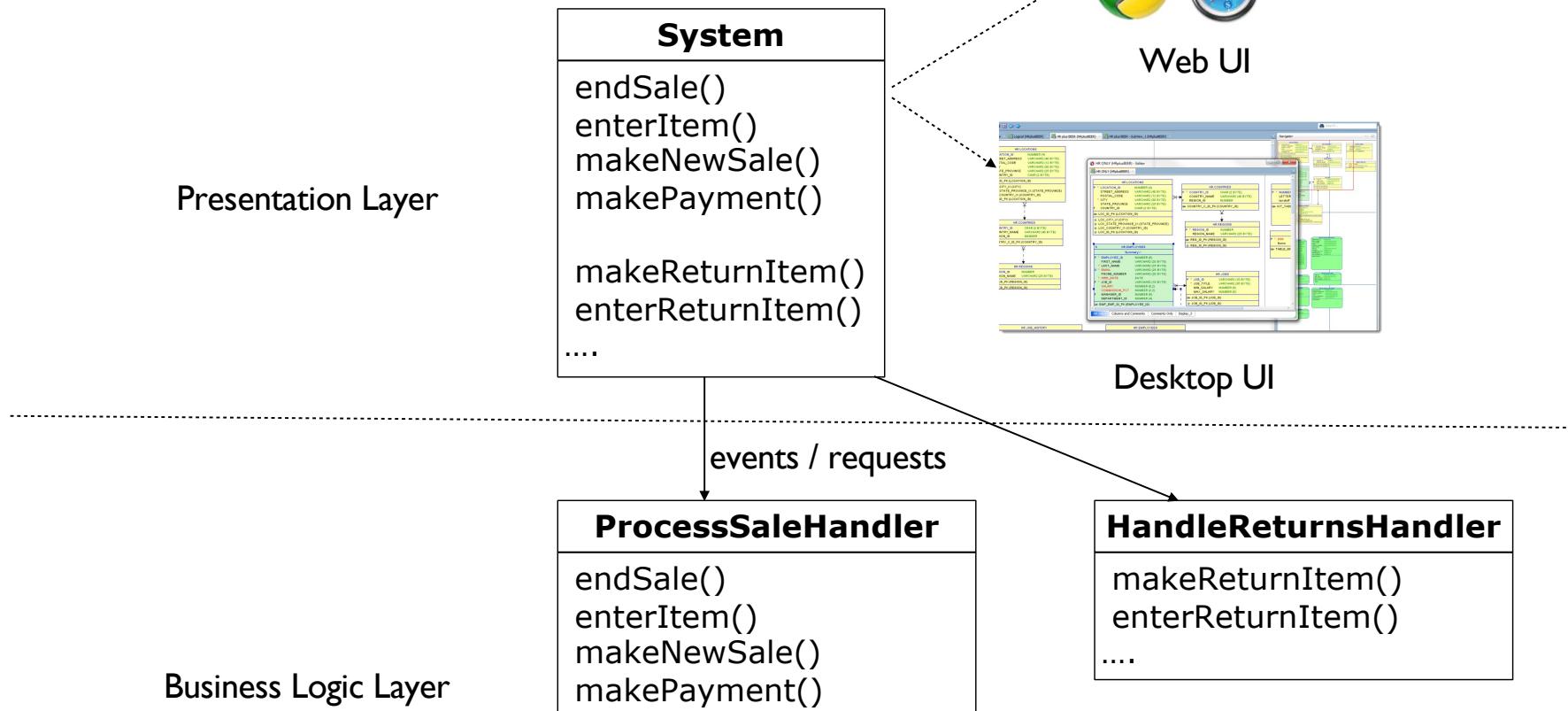
Controller pattern

- Example: Cash Register system
 - Solution 1: use one controller



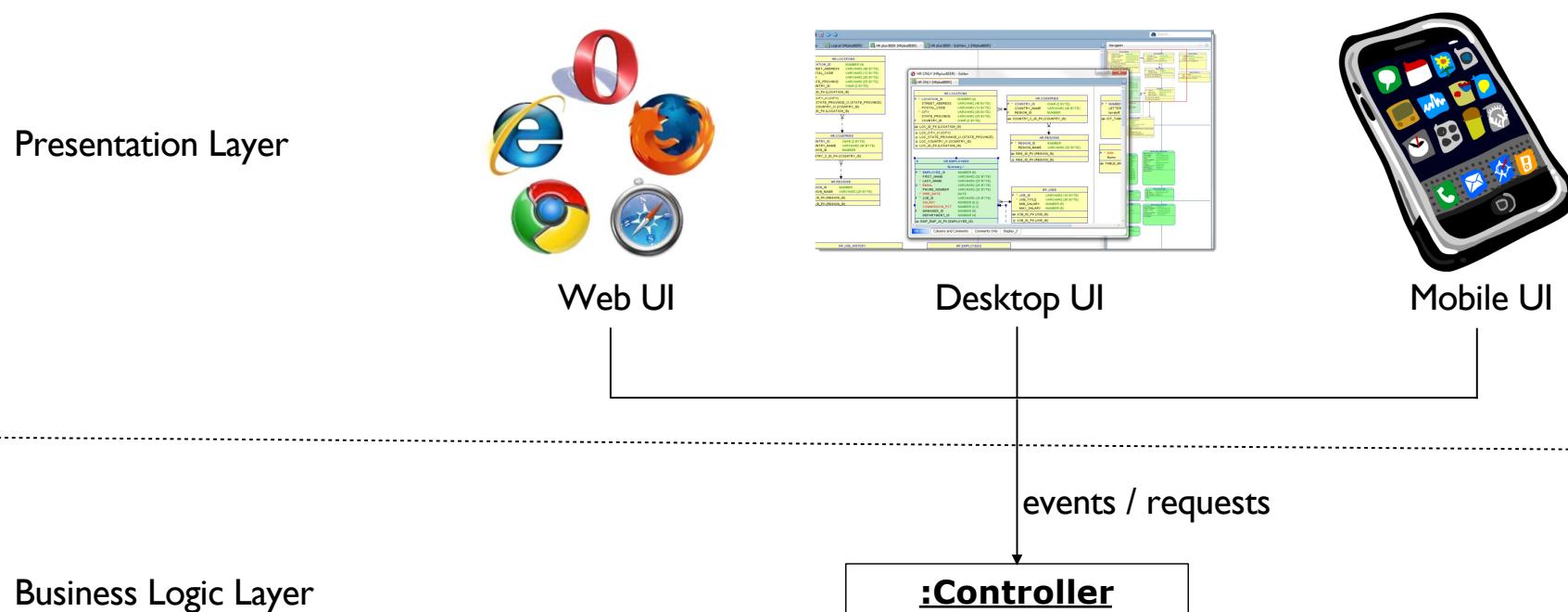
Controller pattern

- Example: Cash Register system
 - Solution 2: use several controllers



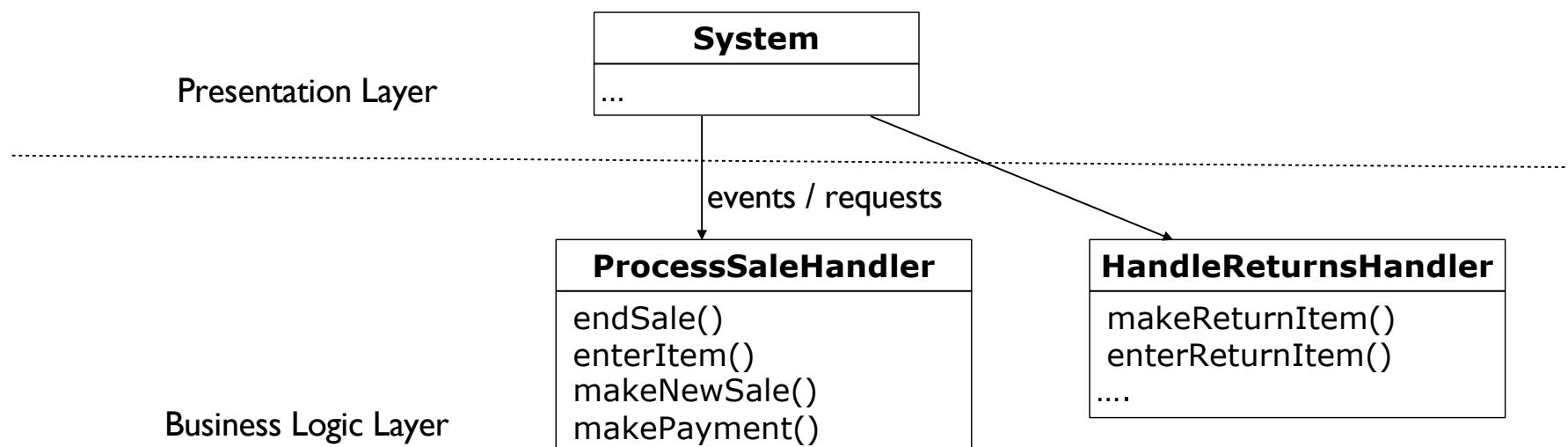
Controller pattern

- Discussion
 - Advantages
 - This is simply a delegation pattern - the UI should not contain application logic
 - Increase potential for reuse and pluggable interfaces
 - Creates opportunity to reason about state of a use-case, for example, to ensure that operations occur in a legal sequence.



Controller pattern

- Discussion
 - Difficulty: **Bloated controllers**
 - a single controller that receives all system events, does too much of the work handling events, has to many attributes (duplicating information found elsewhere), etc.
 - Remedies
 - Add more controllers
 - Design controller so that it primarily delegates the fulfilment of each system operation to other objects.

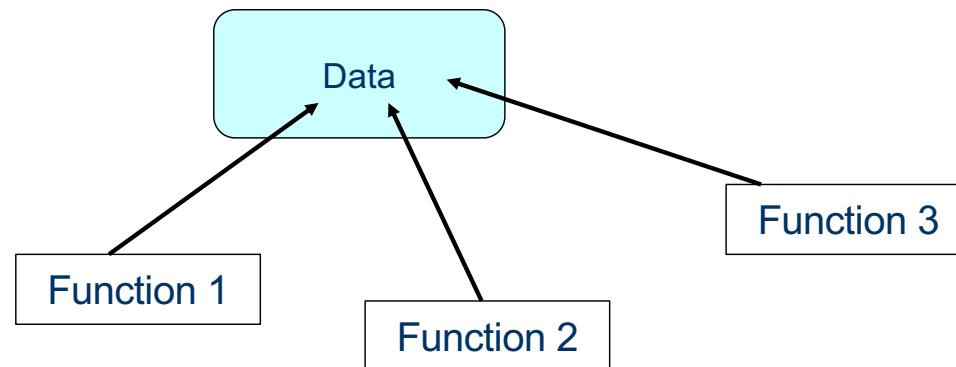


Implementation

- Reminders of object-oriented programming
- From design to implementation

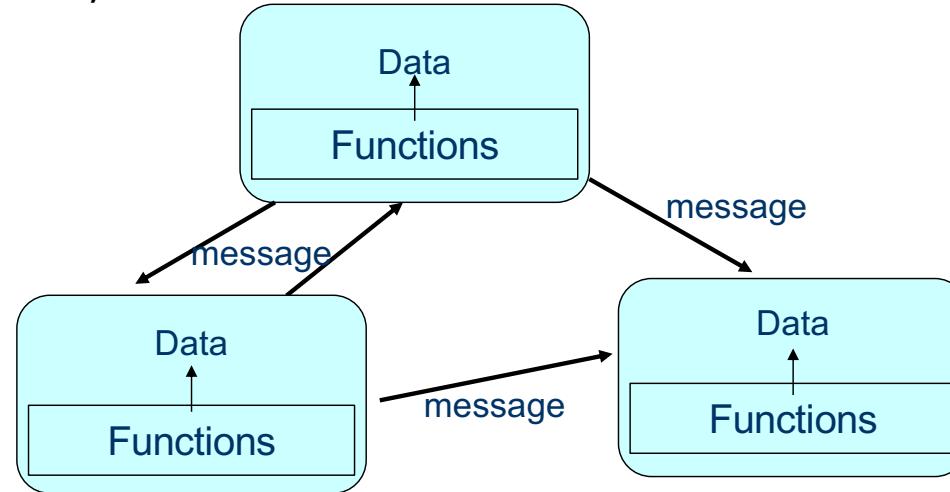
Object-oriented programming

- Functional/imperative programming
 - C/Pascal



Object-oriented programming

- Object-oriented programming
 - C++, Java, C#, ...



- Encapsulation : class
 - Attributes
 - Methods
 - Constructors et destructors
- Inheritance
- Abstract classes et interfaces
- Polymorphism

Object-oriented programming

□ Class : C++

```
class User {
public:
    User(string n, int a):name(n), age(a) {}
    string getName() {return name;}
    int getAge() const {return age;}
    void setName(string n) {name = n;}
    void setAge(int a) {age = a;}
    void print() const ;
    ...
[private:]
    string name;
    int age;
};

void User::print() const {
    cout << "name: " << name << " age: " << age
    << endl;
}

...
User u(" Nguyen Van A ", 35);
User* p = new User( " Nguyen Van A ", 35 );
...
delete p;
...
```

Object-oriented programming

□ Class : Java

```
class User {  
    public     User(String n, int a) {name = n; age = a;}  
    public     String getName() {return name;}  
    public     int   getAge() {return age;}  
    public     void  setName(String n) {name = n;}  
    public     void  setAge(int a) {age = a;}  
    public     void  print(){  
        System.out.println( "name: " + name + " age: " + age );  
    }  
    ...  
    private   String name;  
    private   int   age;  
}  
...  
User u = new User ("Nguyen Van A", 35 );  
...
```

Object-oriented programming

- Constructor and Destructor: C++
 - Constructor
 - initialise attributes and then allocate memory for the attributes
 - Destructor
 - De-allocate the dynamic memory
 - Mandatory: if there are pointer attributes and the memory allocations

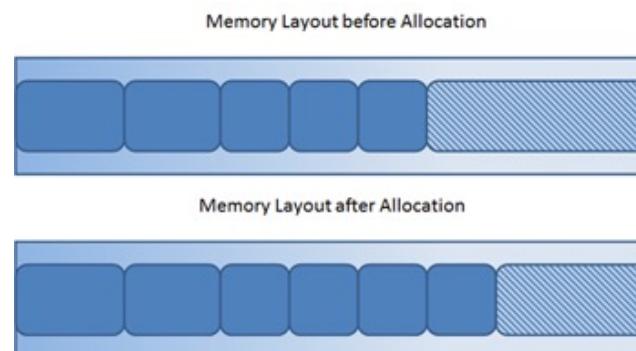
```
class TypeA {};
class ClassB {
    TypeA* p;
public:
    ClassB(TypeA* q) : p( new TypeA(*q) ) {}
    ~ClassB() { delete p; }
};
```

Constructor

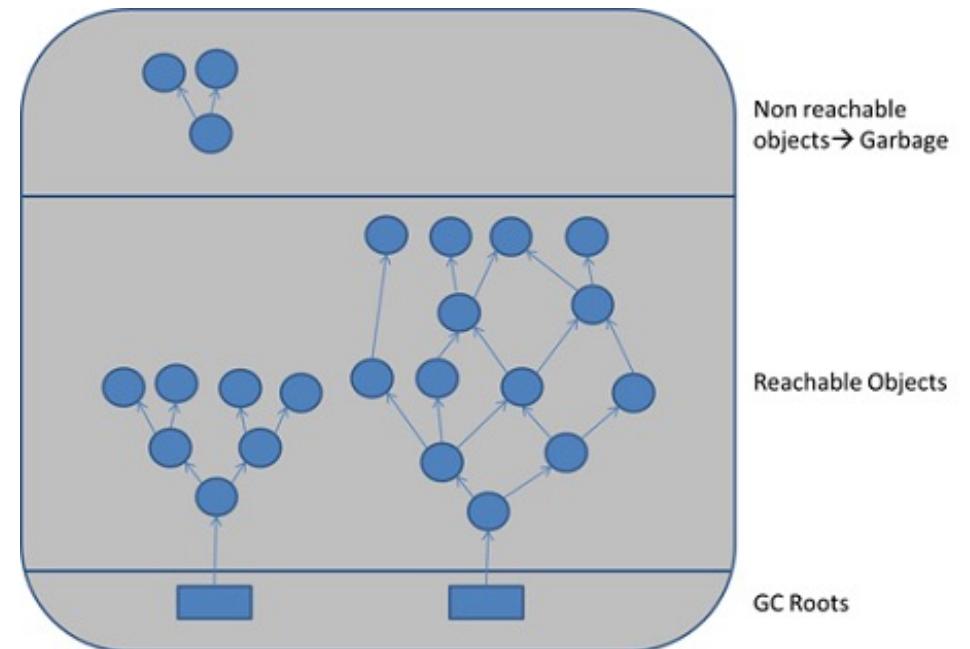
Destructor

Object-oriented programming

- Constructor and destructor: Java
 - There are constructors
 - There are no destructor: The **garbage collector** is responsible for managing the heap memory



New objects are allocated at the end of the used heap



GC roots are special objects referenced by the JVM.
Non reachable objects are garbage-collected

Object-oriented programming

- Inheritance: C++

```
class StudentUser : public User {  
public: StudentUser(string n, int a, string school) : User(n, a){  
    schoolEnrolled = school;  
}  
void print() {  
    User::print();  
    cout << "School Enrolled: " << schoolEnrolled << endl;  
}  
string schoolEnrolled;  
};
```

- Multiple inheritance: C++

```
class StudentUser : public User, public Student { ... };
```

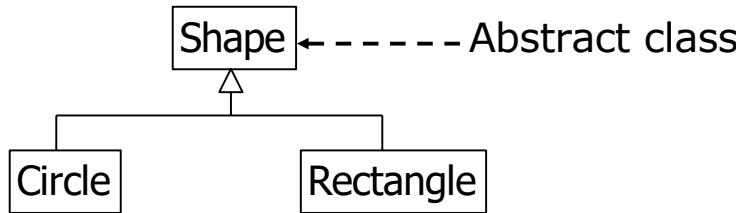
Object-oriented programming

- Inheritance: Java

```
class StudentUser extends User {  
    public StudentUser( String n, int a, String school ) {  
        super(n, a);  
        schoolEnrolled = school;  
    }  
  
    public void print() {  
        super.print();  
        System.out.print( " School: " + schoolEnrolled );  
    }  
  
    private String schoolEnrolled;  
}
```

Object-oriented programming

- Abstract classes and interfaces
 - Java and C++ offer the abstract class concept



- Additionally, Java offers the interface concept
 - An interface is similar a class abstract: no object can be created
 - An interface contains only the method declarations

Object-oriented programming

- Abstract class: C++

```
class Shape {  
public:  
    virtual double area( ) = 0;  
    virtual double circumference() = 0;  
    ....  
}
```

- Abstract class: Java

```
abstract class Shape {  
    abstract public double area( );  
    abstract public double circumference();  
    ....  
}
```

Object-oriented programming

- Java interface

```
interface MyInterface {  
    public double area( );  
    public double circumference();  
    ...  
}  
  
class MyClass implements MyInterface {  
    // Implement the declared methods of MyInterface  
}
```

- Multiple inheritance in Java

```
class MyClass extends SuperClass implements MyInterface1, MyInterface2 {  
    // Implement the declared methods of MyInterface1 and MyInterface2  
}
```

Object-oriented programming

□ Polymorphism in C++

```
class User {  
    string name;  
    int age;  
public:  
    User(string nm, int a) {name=nm; age=a;}  
    virtual void print() {  
        cout << "Name: " << name << " Age: " << age;  
    }  
};  
  
class StudentUser : public User {  
    string schoolEnrolled;  
public:  
    StudentUser(string name, int y, string school) : User(name, y) {  
        schoolEnrolled = school;  
    }  
    void print() {  
        User::print();  
        cout << " School Enrolled: " << schoolEnrolled;  
    }  
};
```

Object-oriented programming

□ Polymorphism in C++

```
int main()
{
    User* users[3];

    users[0] = new User( "Buster Dandy", 34 );
    users[1] = new StudentUser("Missy Showoff", 25, "Math");
    users[2] = new User( "Mister Meister", 28 );

    for (int i=0; i<3; i++)
    {
        users[i]->print();
        cout << endl;
    }

    delete [] users;
    return 0;
}
```

Object-oriented programming

□ Polymorphism in Java

```
class User {  
    public User( String str, int yy ) {  
        name = str;  
        age = yy;  
    }  
    public void print() {  
        System.out.print( "name: " + name + " age: " + age );  
    }  
  
    private String name;  
    private int age;  
}  
  
class StudentUser extends User {  
    public StudentUser( String nam, int y, String sch ) {  
        super(nam, y);  
        schoolEnrolled = sch;  
    }  
    public void print() {  
        super.print();  
        System.out.print( " School: " + schoolEnrolled );  
    }  
  
    private String schoolEnrolled;  
}
```

Object-oriented programming

- Polymorphism in Java

```
class Test
{
    public static void main( String[] args )
    {
        User[] users = new User [3];
        users[0] = new User( "Buster Dandy", 34 );
        users[1] = new StudentUser( "Missy Showoff",25, "Math");
        users[2] = new User( "Mister Meister", 28 );
        for (int i=0; i<3; i++)
        {
            users [i].print();
            System.out.println();
        }
    }
}
```

Main Activities of Software Development

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

Deployment

Installation and training

Maintenance

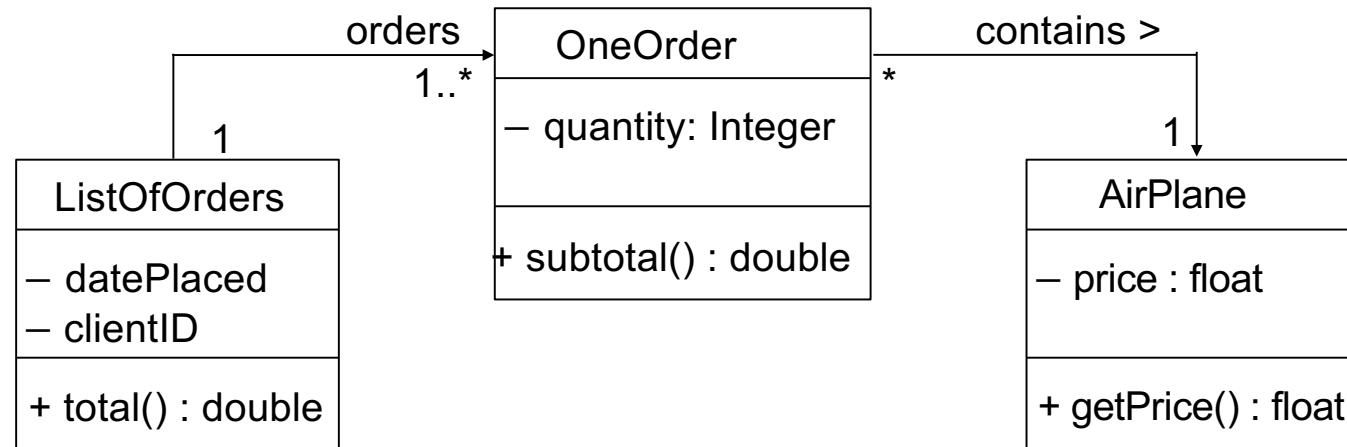
Post-install review
Support docs
Active support

From design to code

- Generation of source code from the design model
- Object-oriented code includes
 - Definitions of classes and interfaces
 - Definitions of methods
- The **class diagrams** are transformed to **classes** and **interfaces**
- The **interaction diagrams** are transformed to **code of methods**.
- Other diagrams allow to guide the programmer during coding

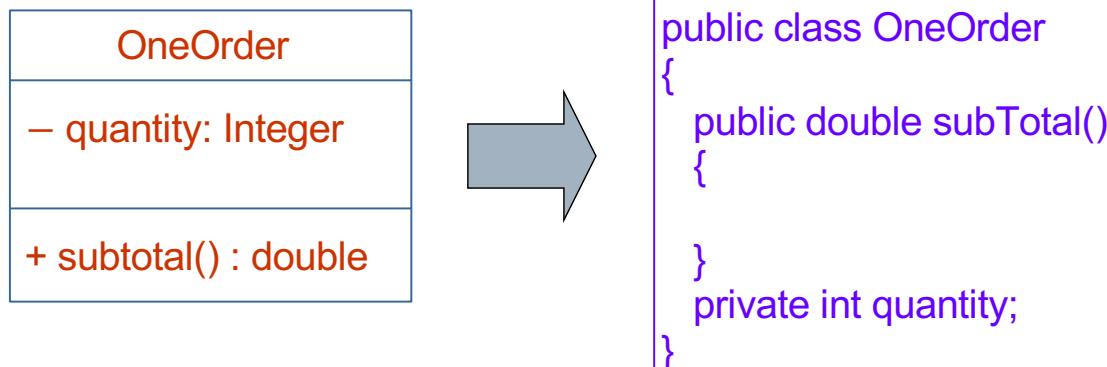
From design to code

- Class definition
 - Example of a part of class diagram



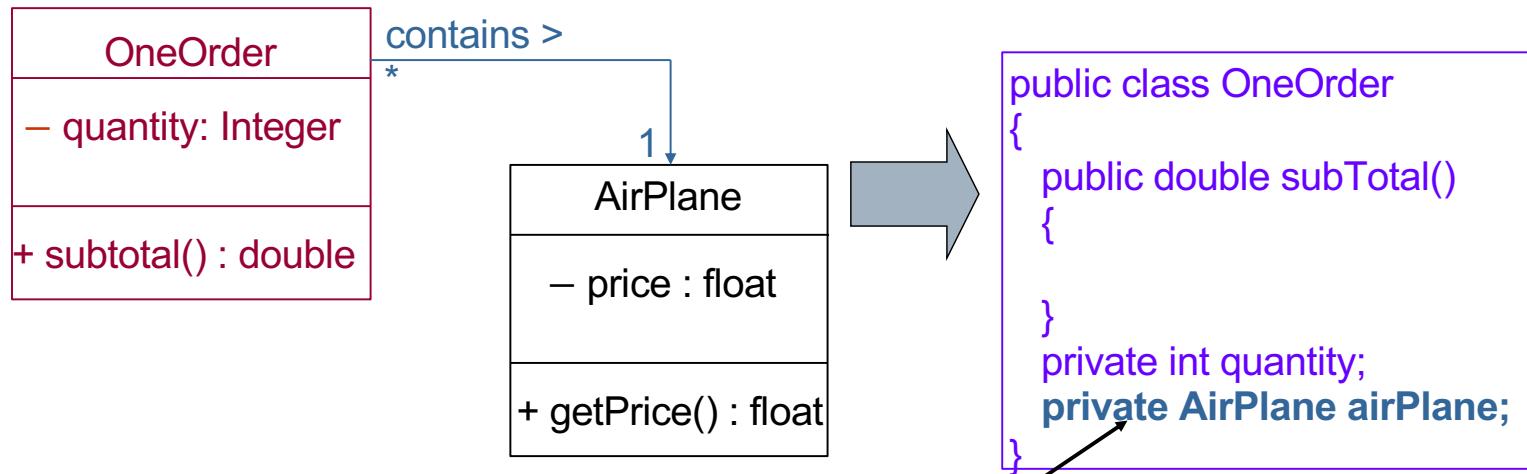
From design to code

- Class definition
 - Code of OneOrder class



From design to code

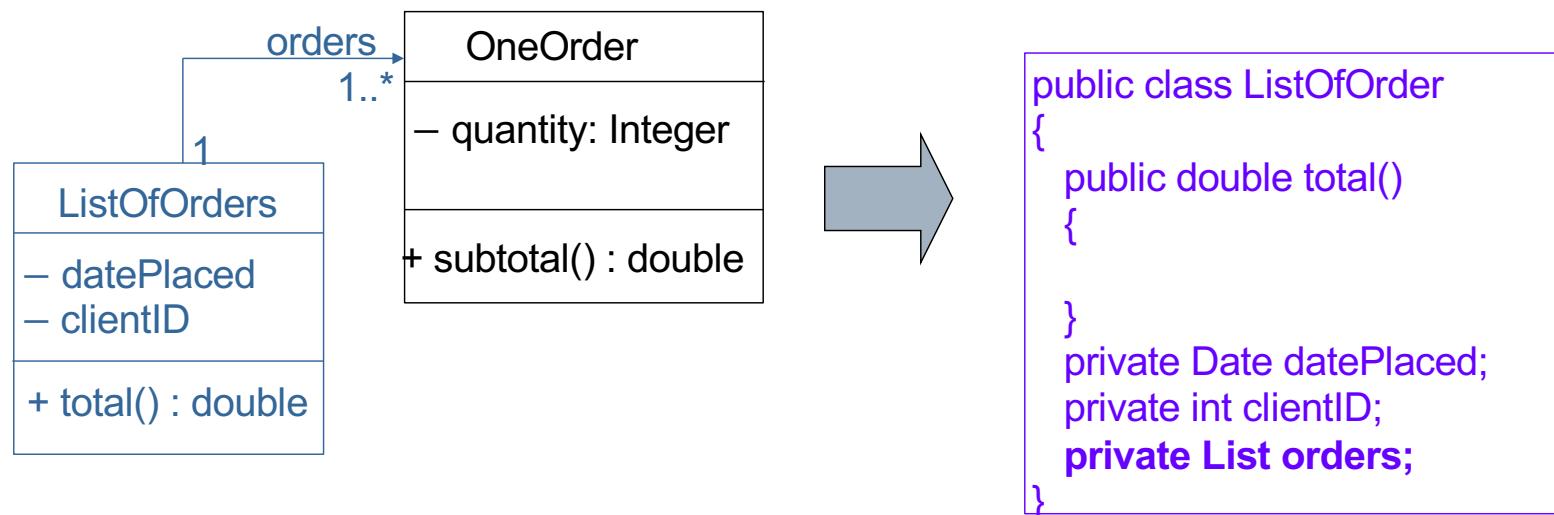
- Class definition
 - Code of OneOrder class



If the role of an association is not explicit, the created attribute takes the associated role.

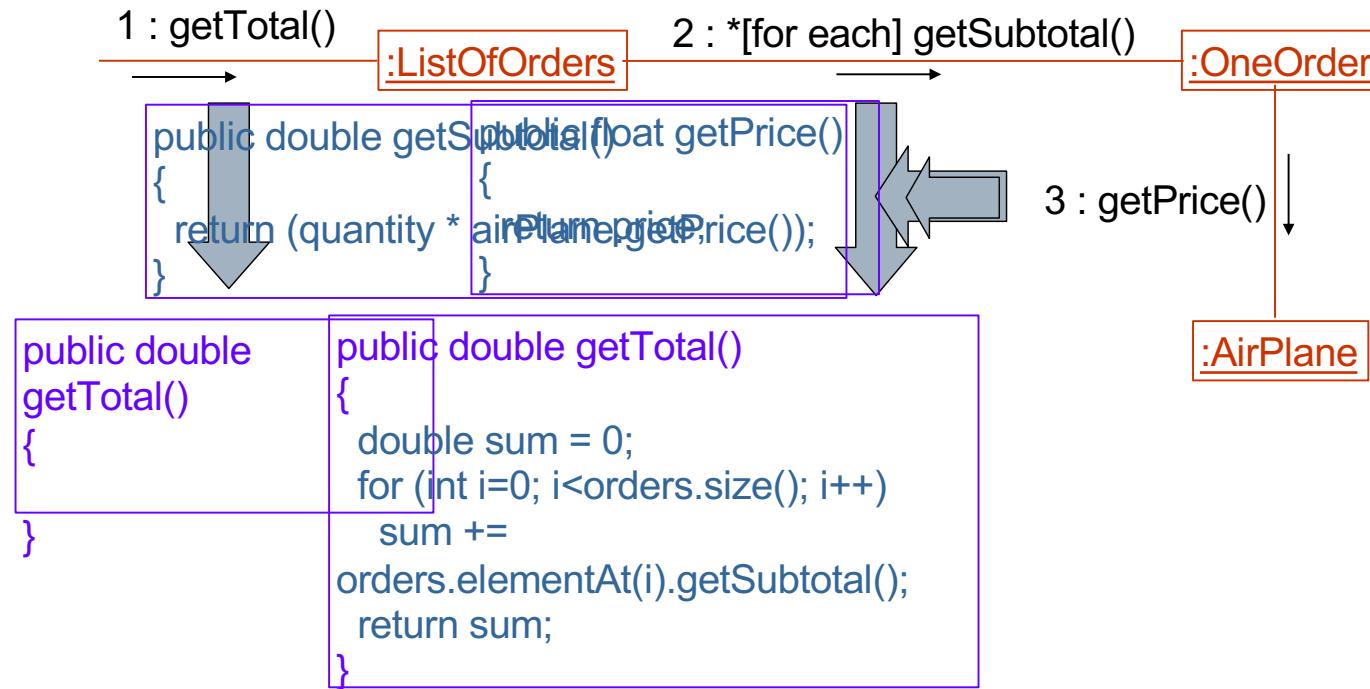
From design to code

- Definition of classes
 - Code of ListOfOrders class



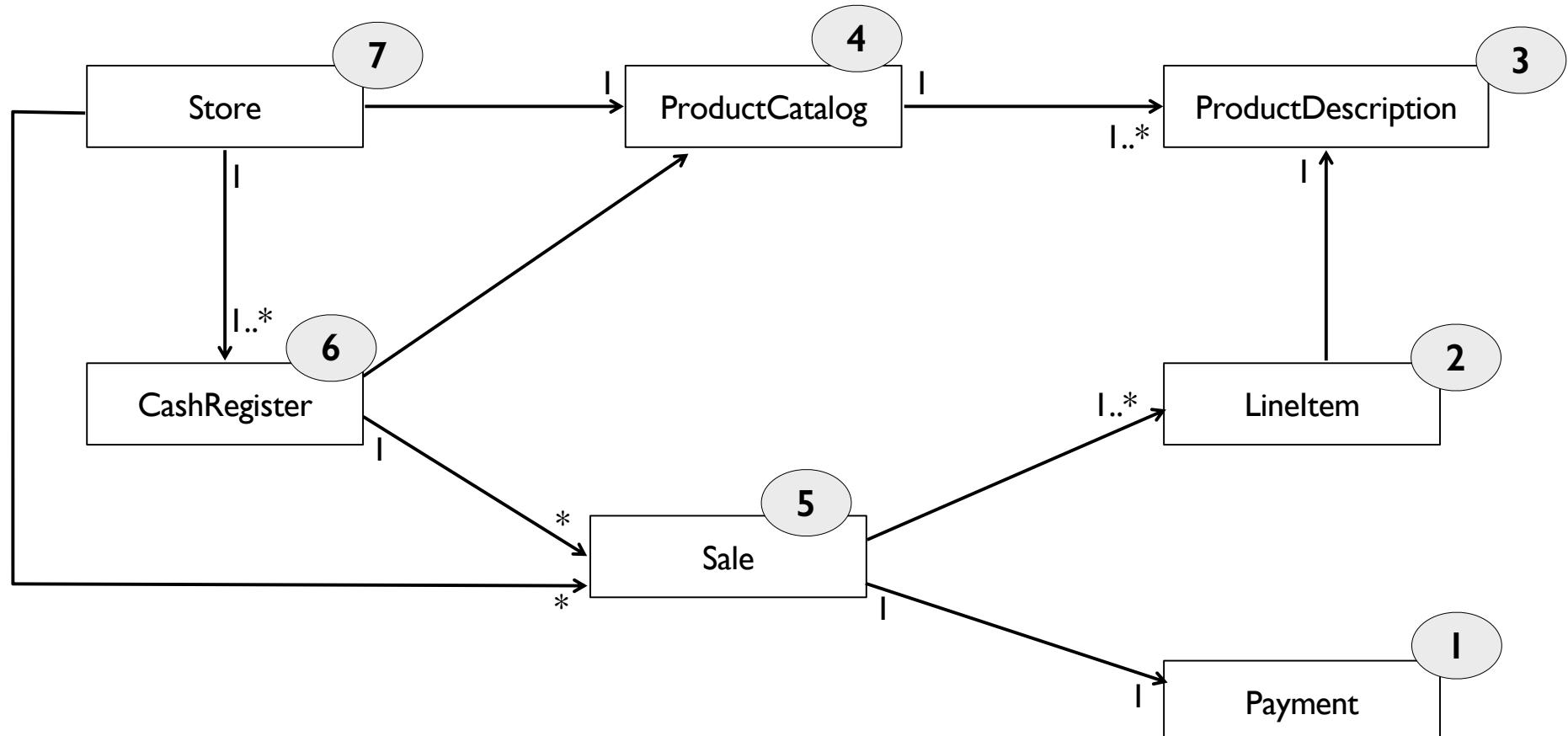
From design to code

- Method definition
 - Interaction diagram defines the **getTotal()** method



From design to code

- Implementation order
 - Class must be implemented from the least coupled/dependent to the most coupled/dependent



From design to code

- Several UML tools
 - Rational Rose, Dia ULM, Piosedon for UML, Umbrello, Power Design, Dia, **StarUML**
 - Draw UML diagrams
 - Automatically generate source code: Java, C++, C#, ...
- Automatically source code generation
 - Imperfect
 - Only the skeleton

Case study

Case study

- Problem
 - A very simple problem to show the use of UML in analysis and design
 - It is taken from the "Applying UML and Patterns" book of Claig Larman

- A dice game
 - The player rolls 10 times 2 dice. If the total of two dice is 7, he gains 10 points. At the end of the game, the score is saved to the scoreboard



Main Activities of Software Development

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

Deployment

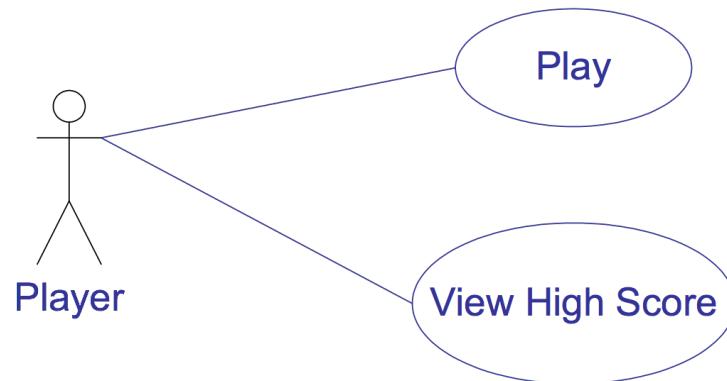
Installation and training

Maintenance

Post-install review
Support docs
Active support

Case study

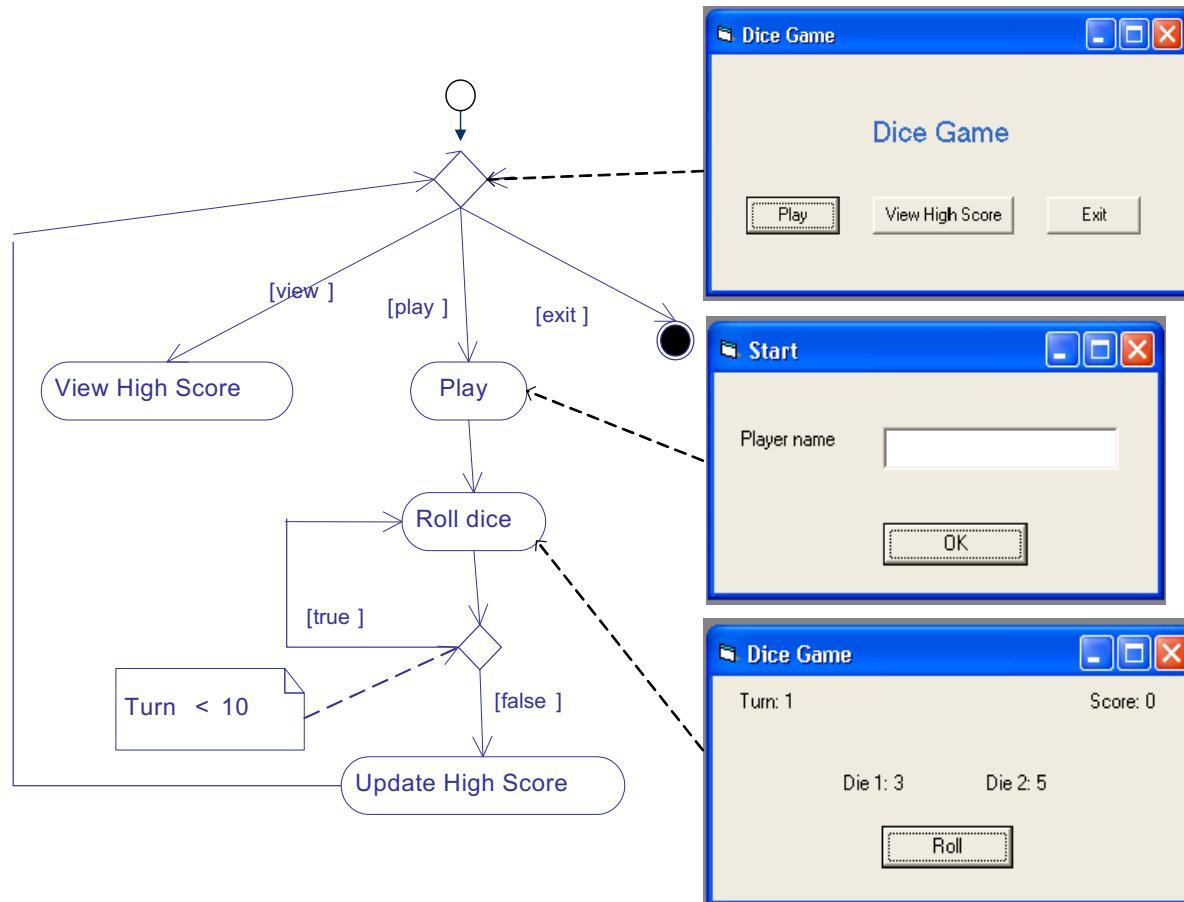
- Requirement analysis
 - Use-case diagram



- Use-case: Play
 - Description: The player rolls 2 dice 10 times. If each time the total is 7, he receives 10 points.
- Use-case: View High Score
 - Description: They player consults the scores

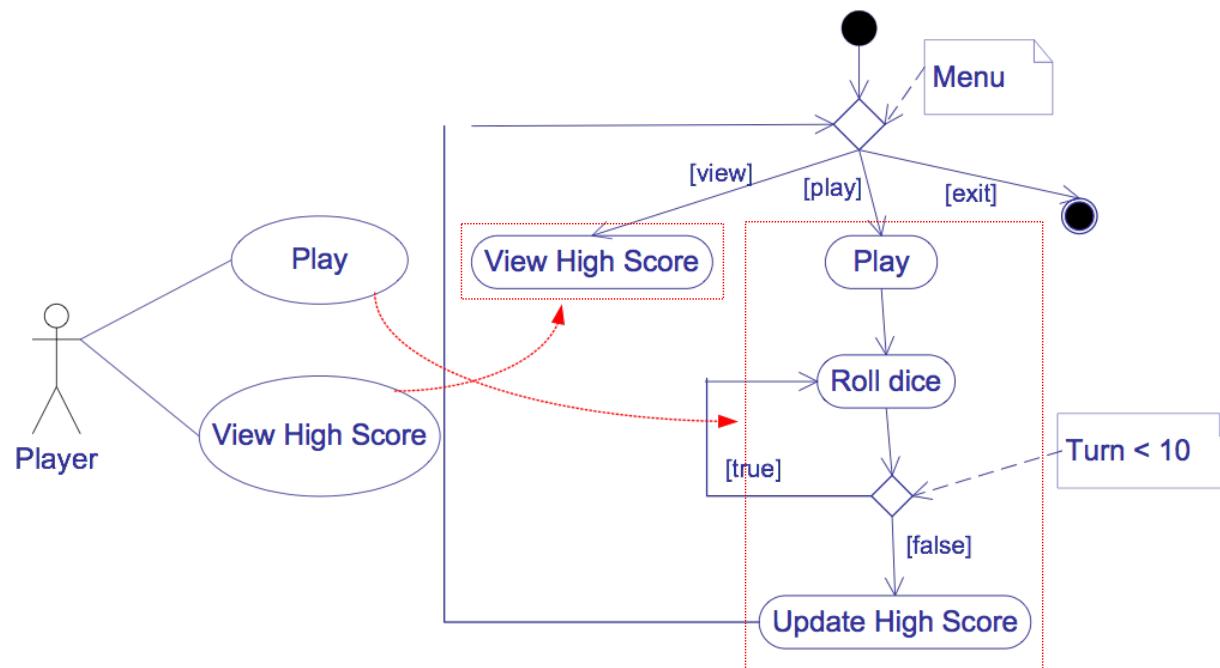
Case study

- Requirement analysis
 - Activity diagram
 - Some activities are linked to the graphical user interface



Use-case

- Requirement analysis
 - Activity diagram
 - The relationship between the use-case diagram and activity diagram



Main Activities of Software Development

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

Deployment

Installation and training

Maintenance

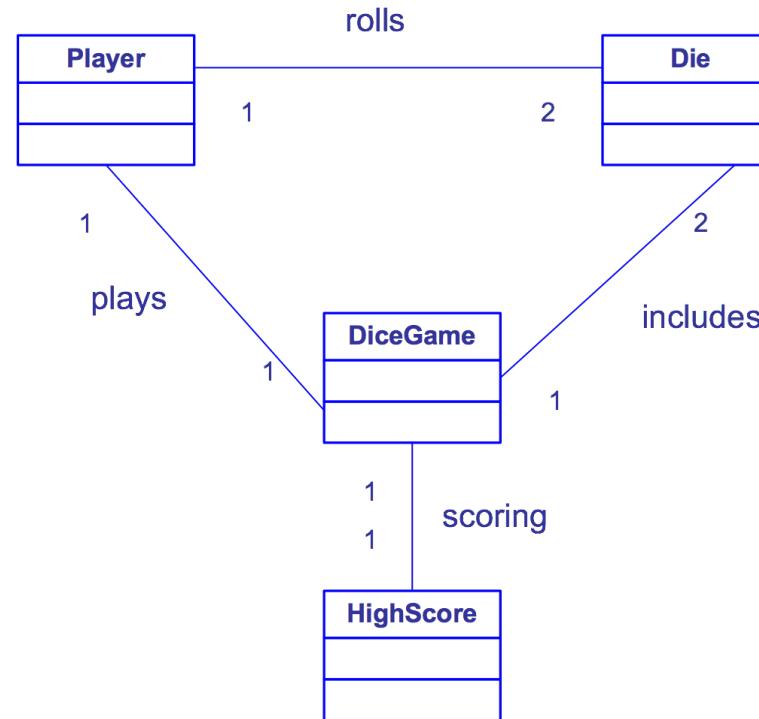
Post-install review
Support docs
Active support

Case study

- Analysis
 - Modelling the real world
 - Independent of the implementation
 - Modelling of the domain: conceptual class diagram
 - Modelling of the dynamic behaviour of the system: collaboration diagram

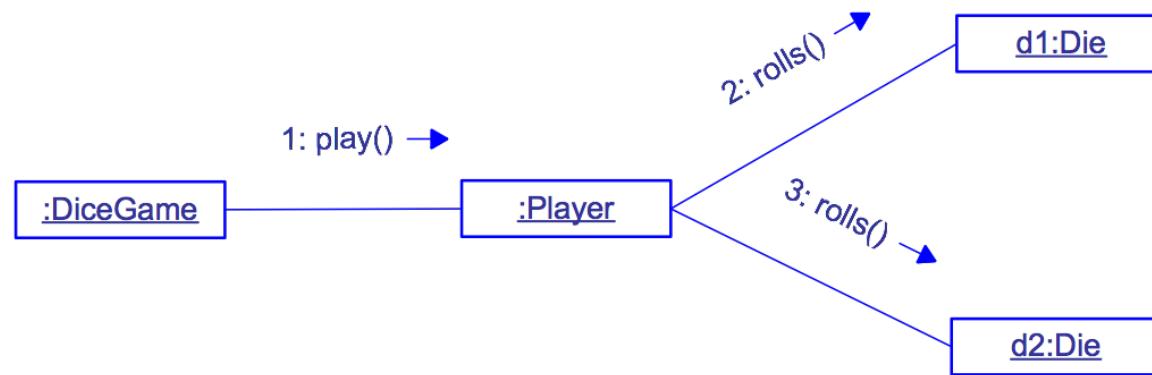
Case study

- Modeling of conceptual class diagram



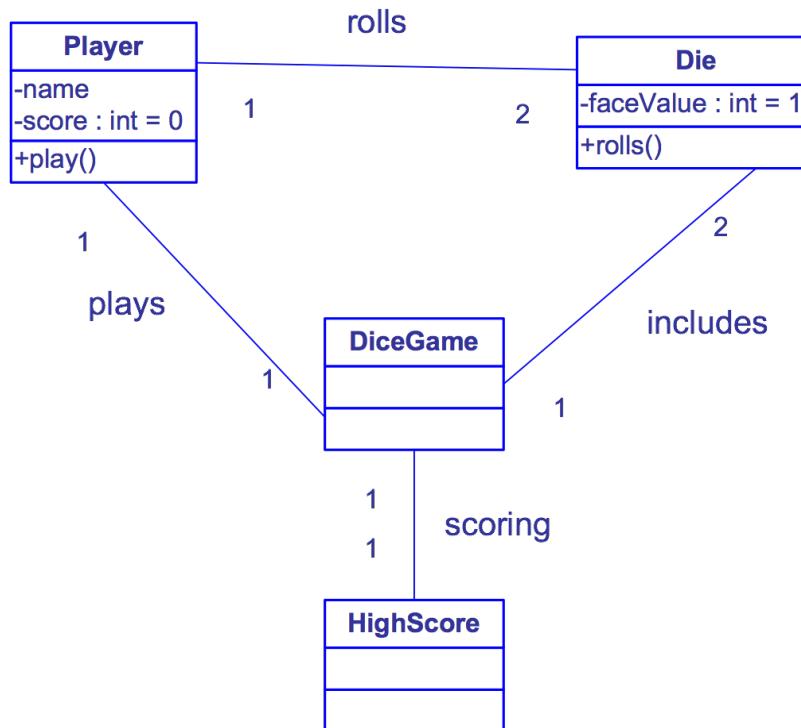
Case study

- A first collaboration diagram



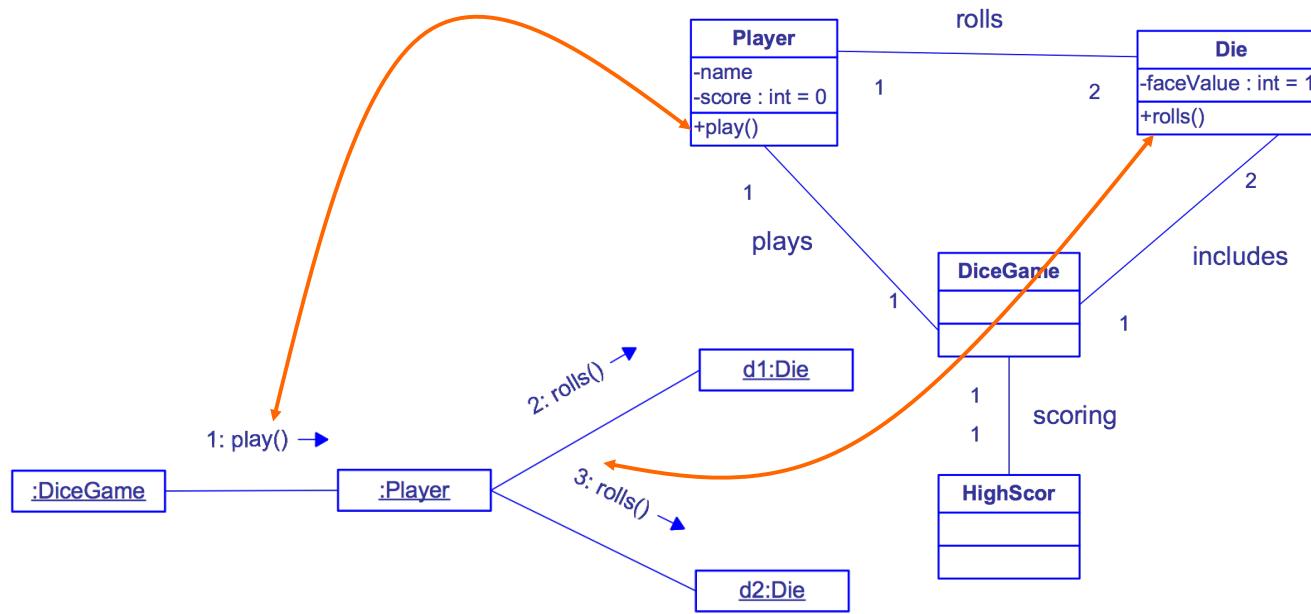
Case study

- A first class diagram



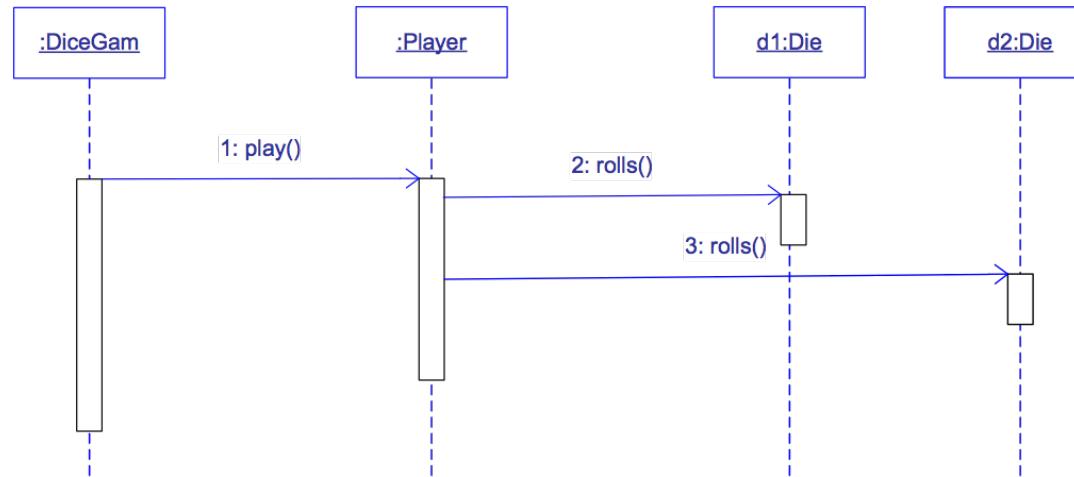
Case study

- Collaboration diagram and class diagram



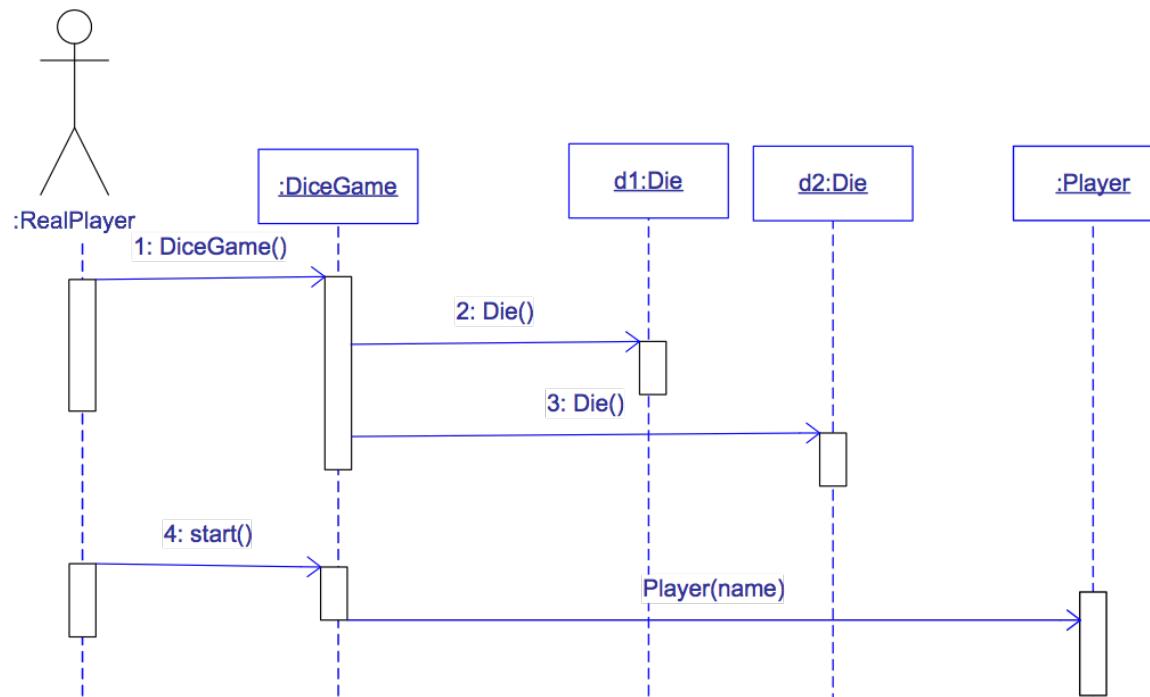
Case study

- Sequence diagram



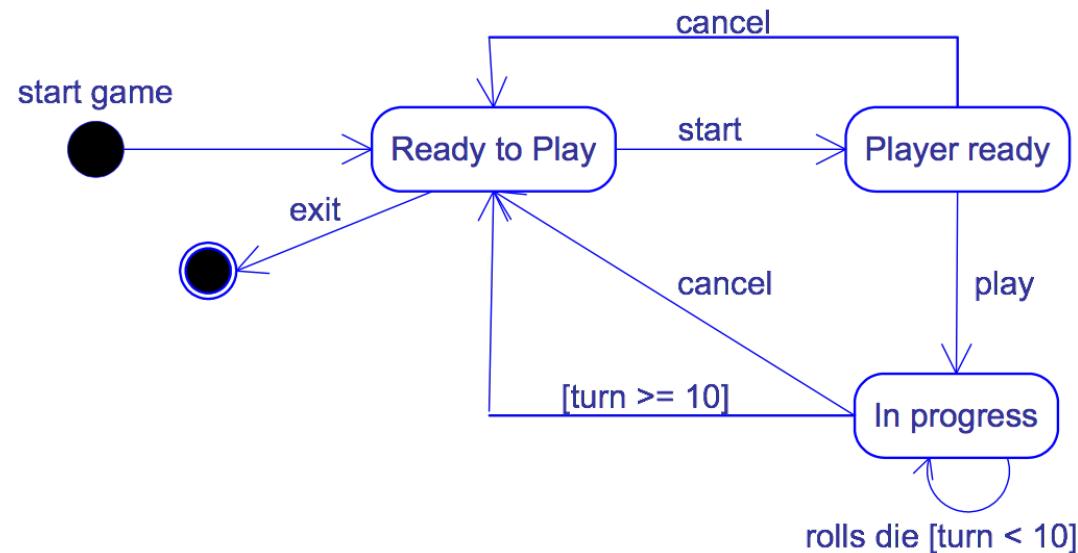
Case study

- The creation of objects at the beginning of the game (DiceGame) for a player



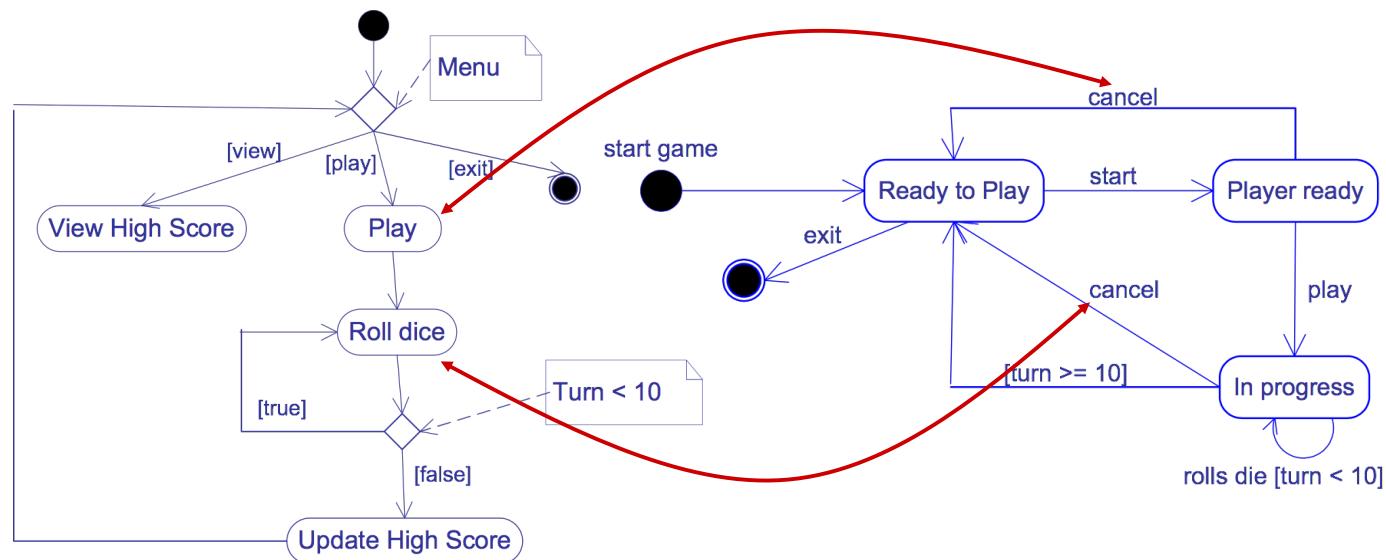
Case study

- State diagram: modelling the states of the DiceGame



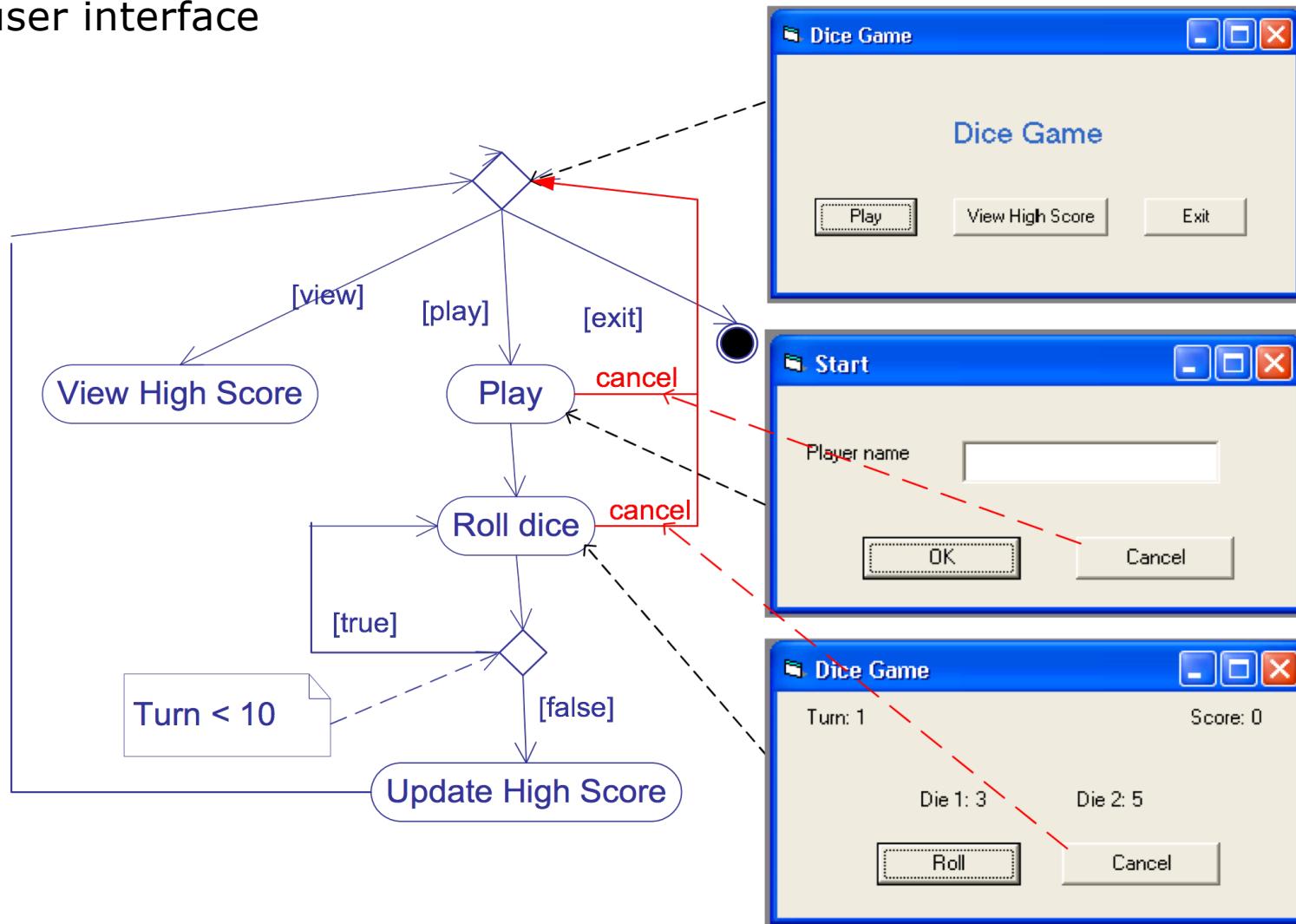
Case study

- Detection of inconsistency between the activity diagram and the state diagram



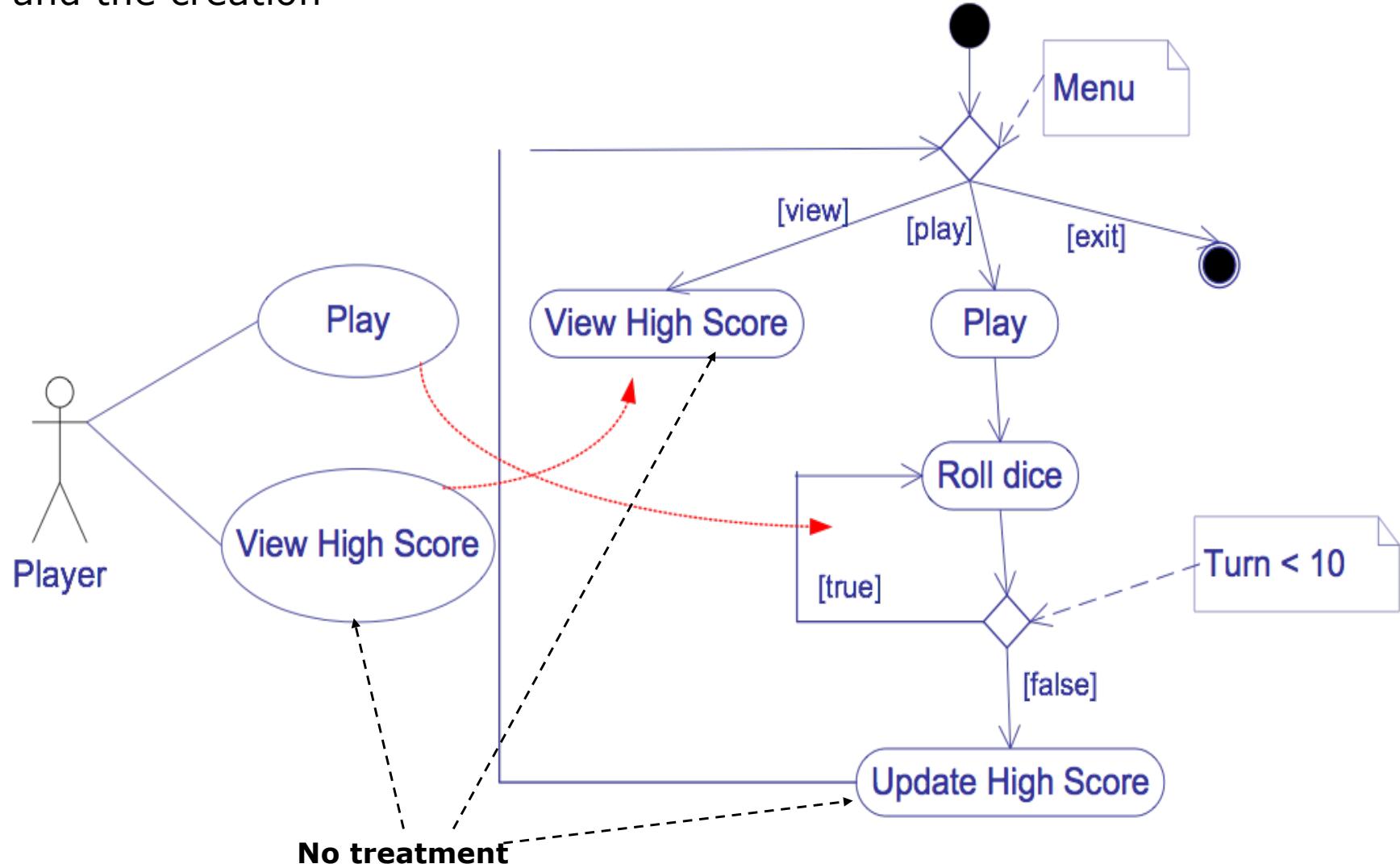
Case study

- Modification of the activity diagram as well as the envisaged graphical user interface



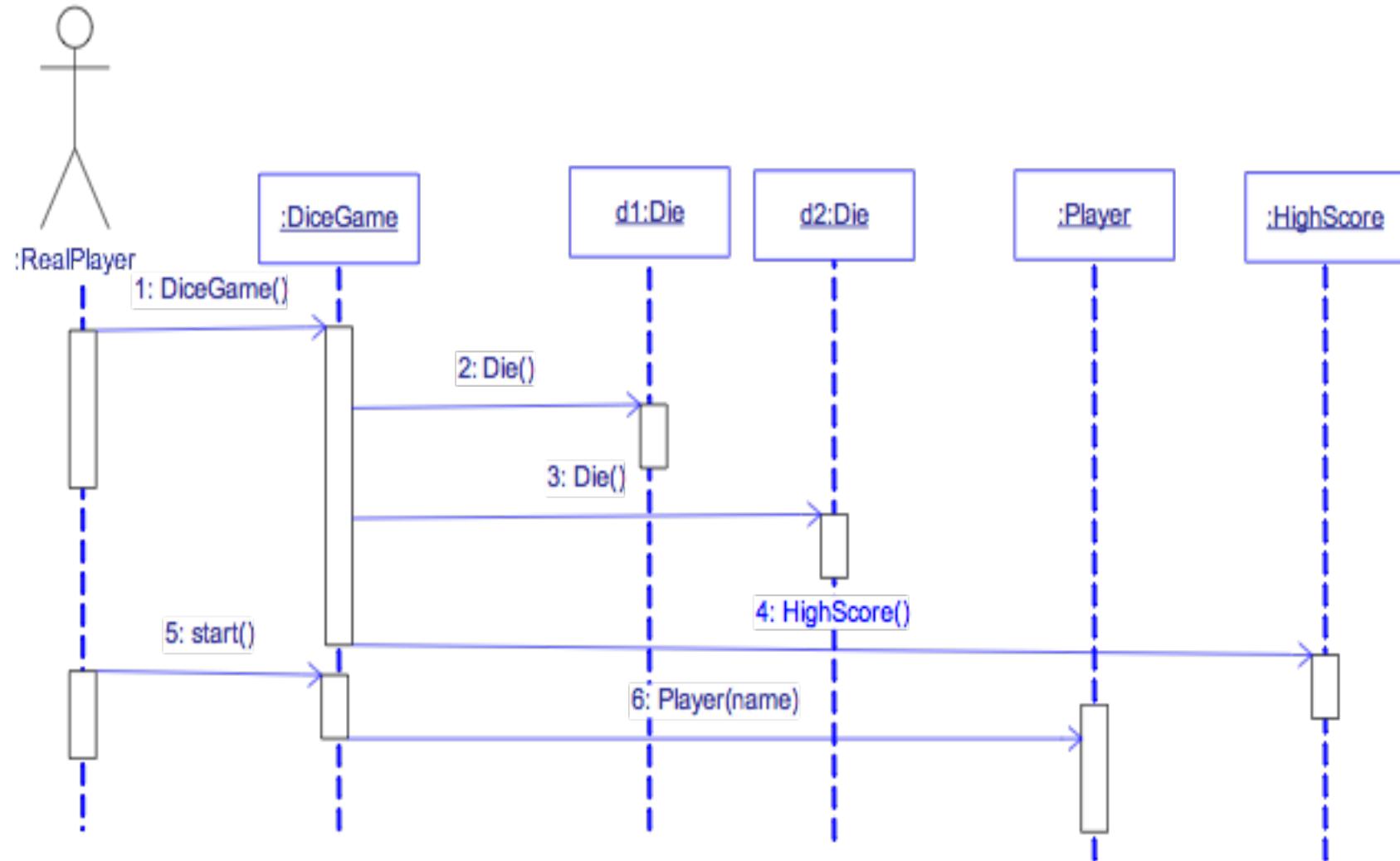
Case study

- The treatment of the scoreboard must be taken into account: the update and the creation



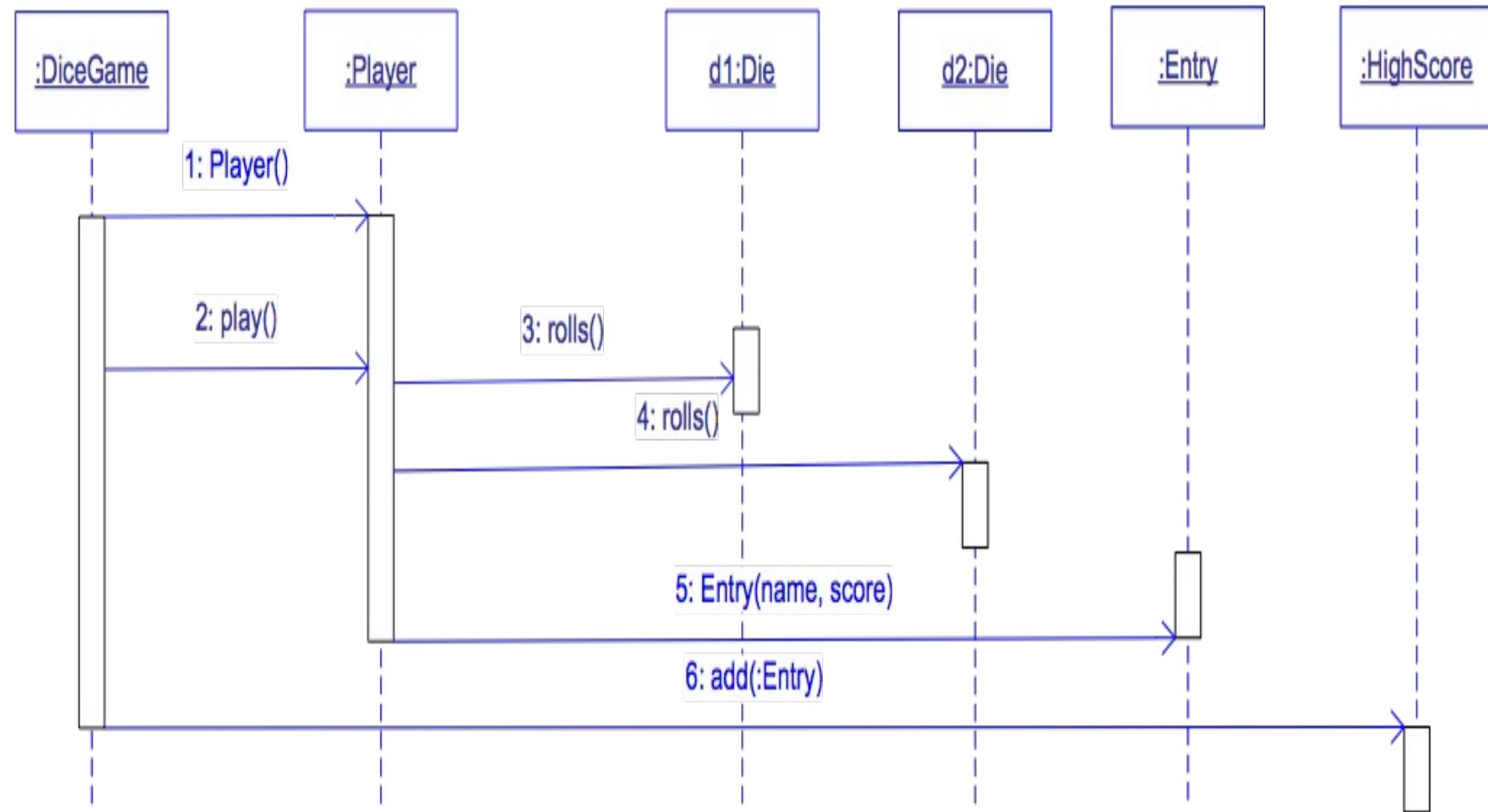
Case study

- Sequence diagram: manage high score, create new player



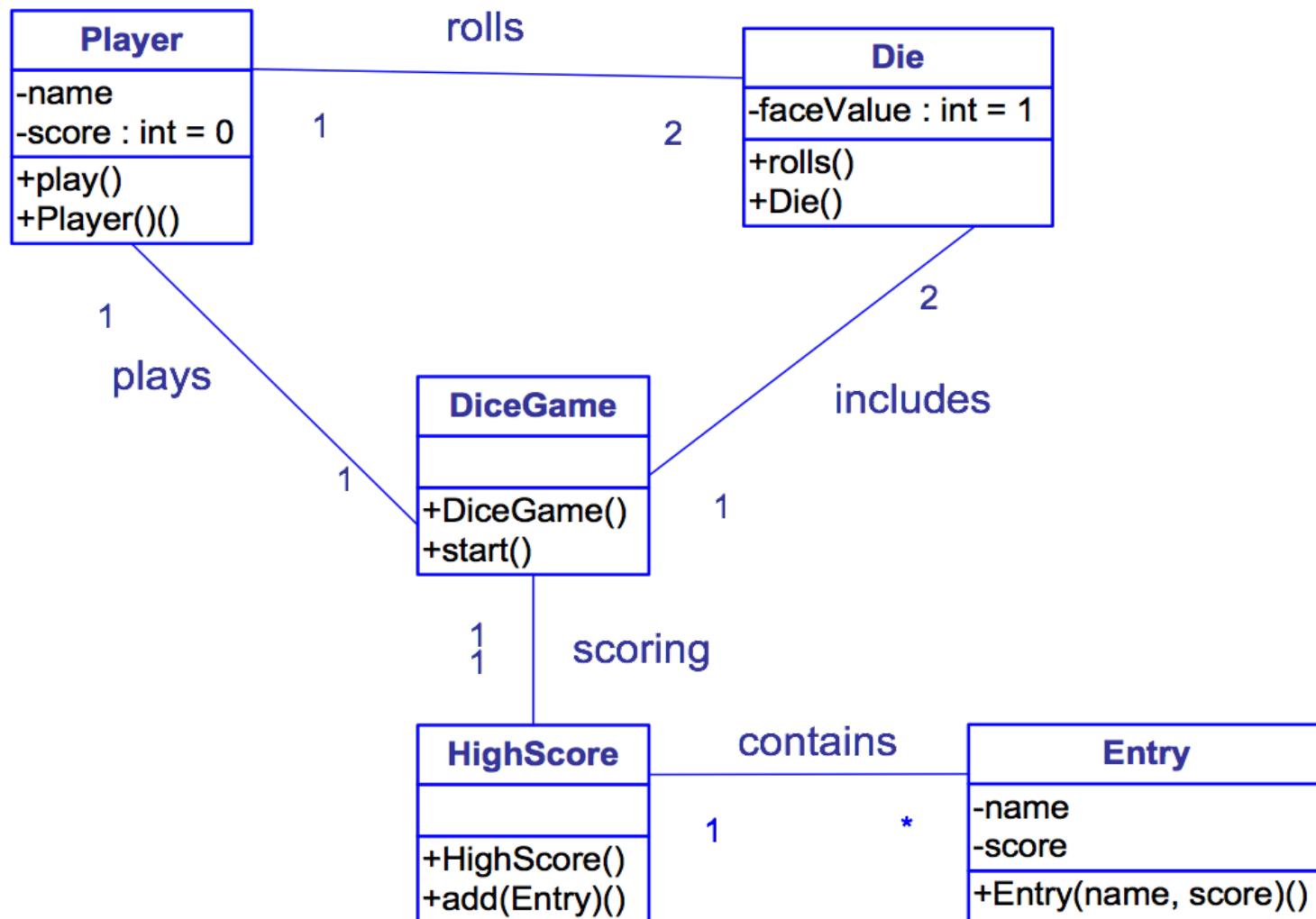
Case study

- Sequence diagram: add high score to score board



Case study

Class diagram



Main Activities of Software Development

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

Deployment

Installation and training

Maintenance

Post-install review
Support docs
Active support

Case study

- Design
 - Take into account the implementation
 - Manage the graphical user interface part
 - Manage the persistence of scoreboard
 - Define the logical architecture
 - Define the physical architecture
 - Introduce the technical class permitting to implement the architecture

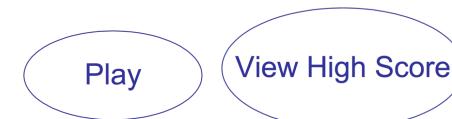
Case study

- General architecture
 - Classical three layer architecture

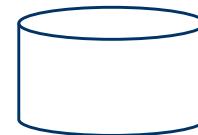
Presentation



Business Logic



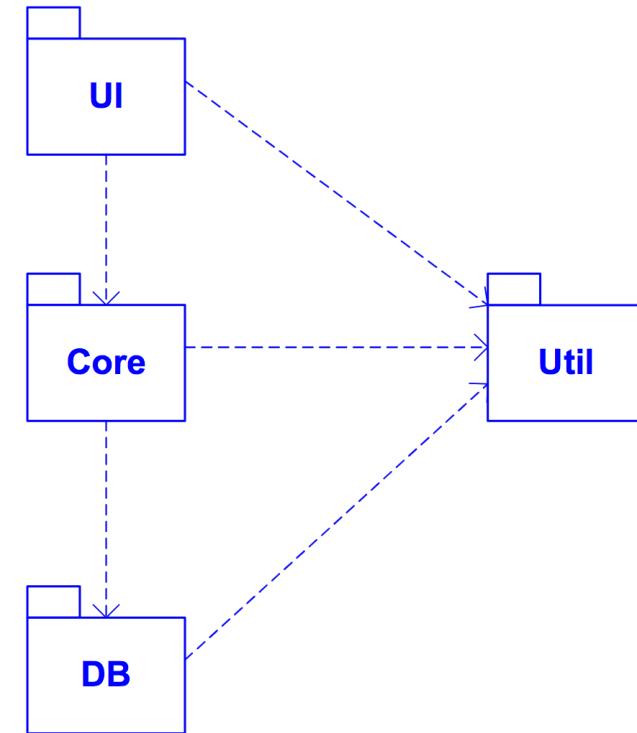
Persistence



Case study

- A package diagram corresponds to the architecture

UI : presentation layer
Core : Business logic layer
DB : Persistence layer
Util : utility services/classes/functionalities



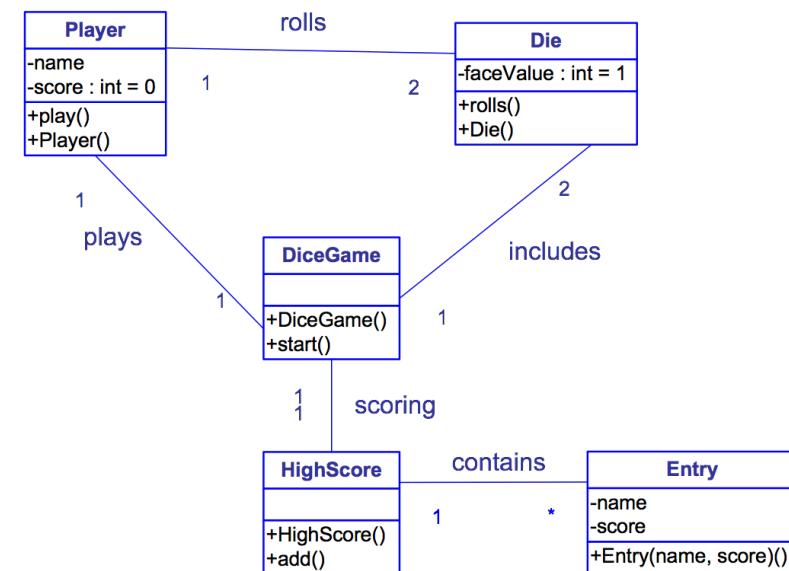
Case study

- Use design patterns to improve the classes of “Core” package

La classe DiceGame ne possède qu'un seul objet
La classe HighScore ne possède qu'un seul objet

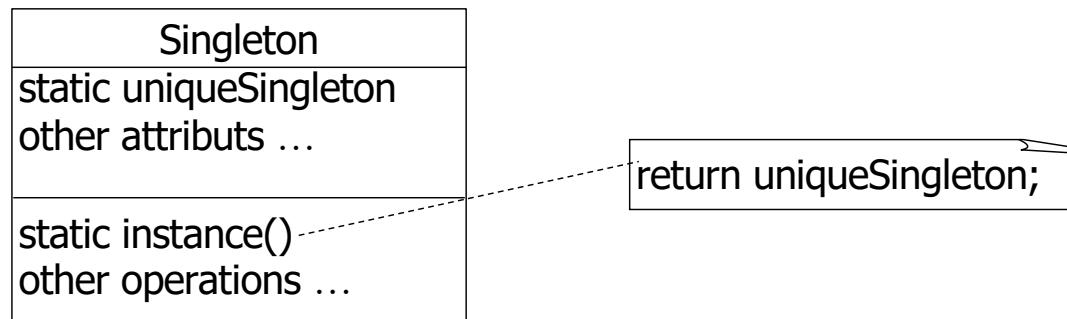


Le patron de conception : Singleton



Case study

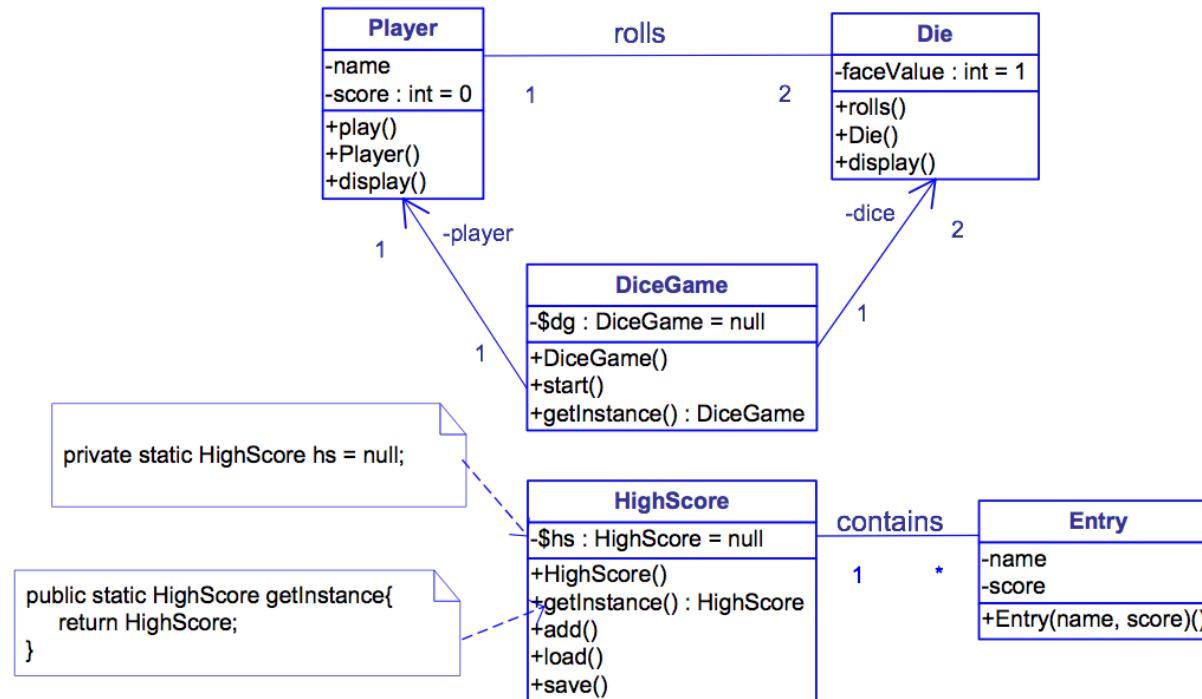
- **Singleton** design pattern



- Application to **DiceGame** and **HighScore**.

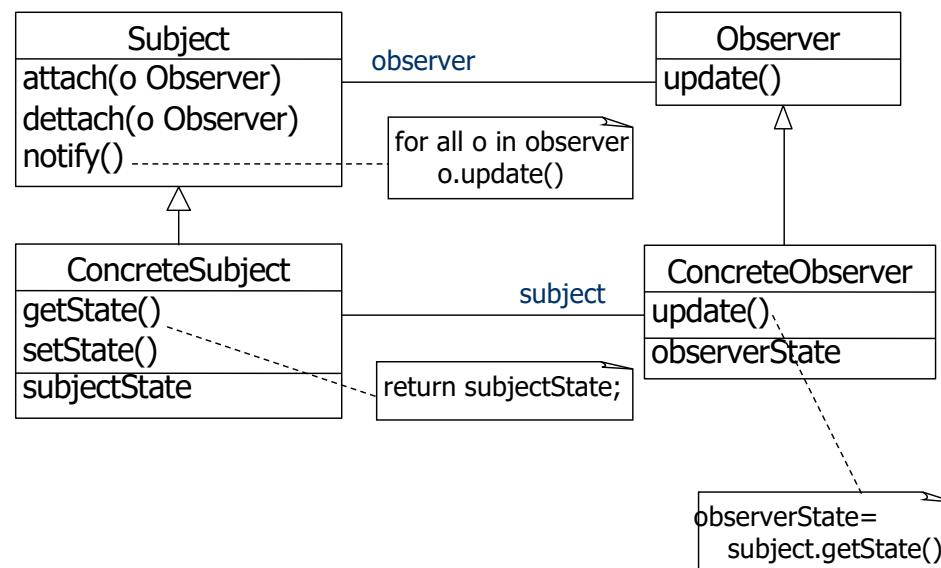
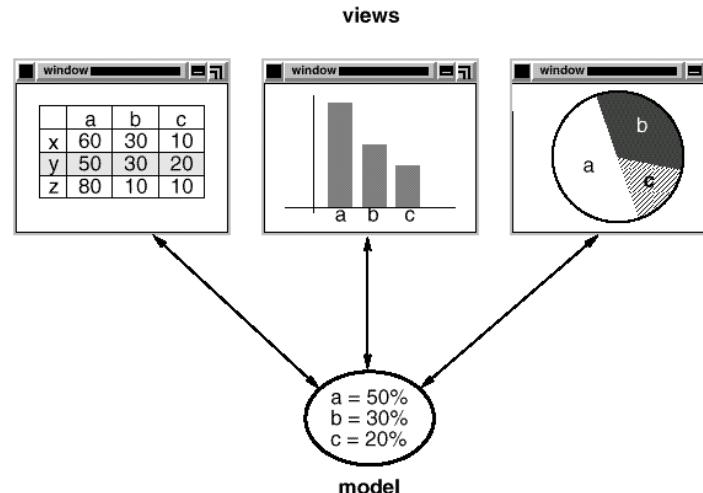
Case study

□ Modified class diagram



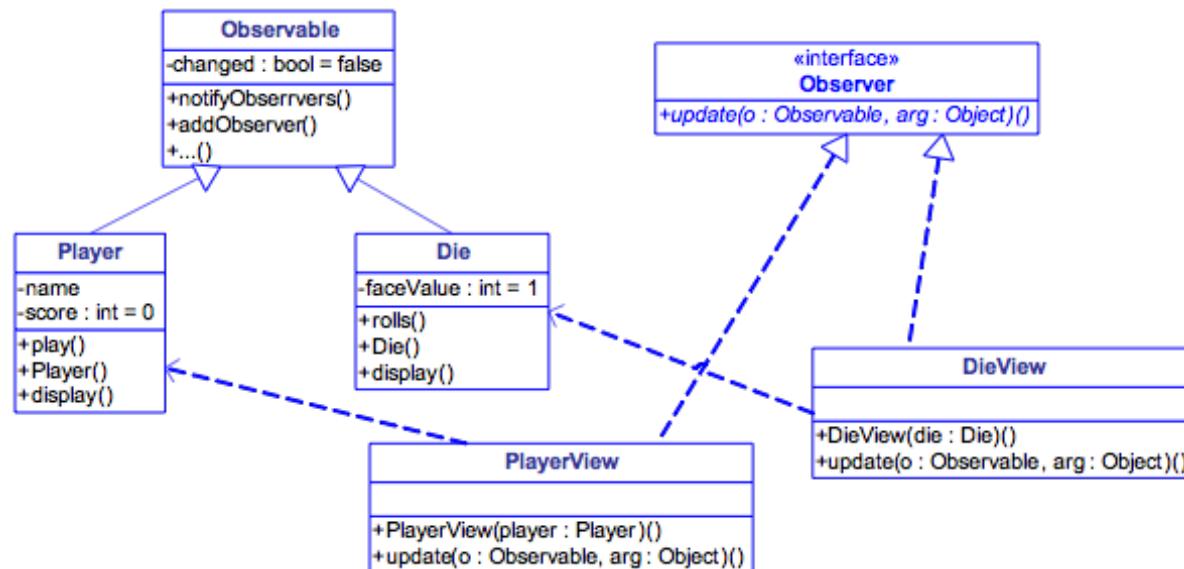
Case study

□ Observer design pattern



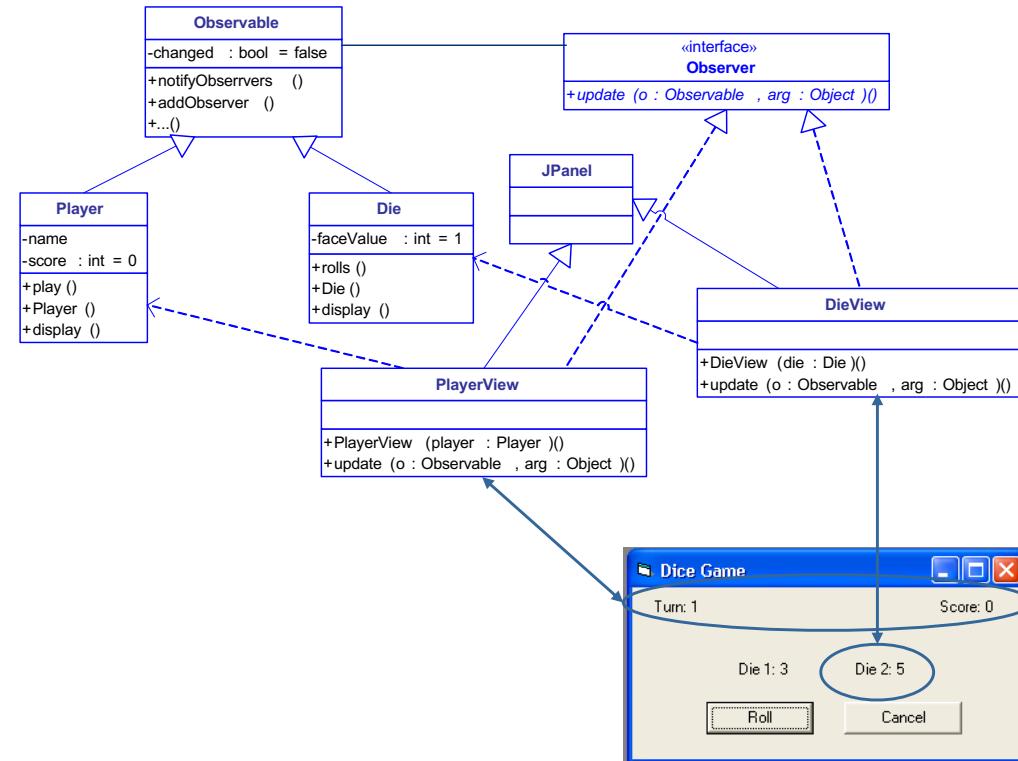
Case study

- Application of **Observer** design pattern to improve the class diagram
 - Decouple the graphical views and objects for the dice and players
 - Application of **Observer** pattern
 - **Die** and **Player** classes are **ConcreteSubject** class
 - Introduce **DieView** et **PlayerView** as **ConcreteObserver** classes



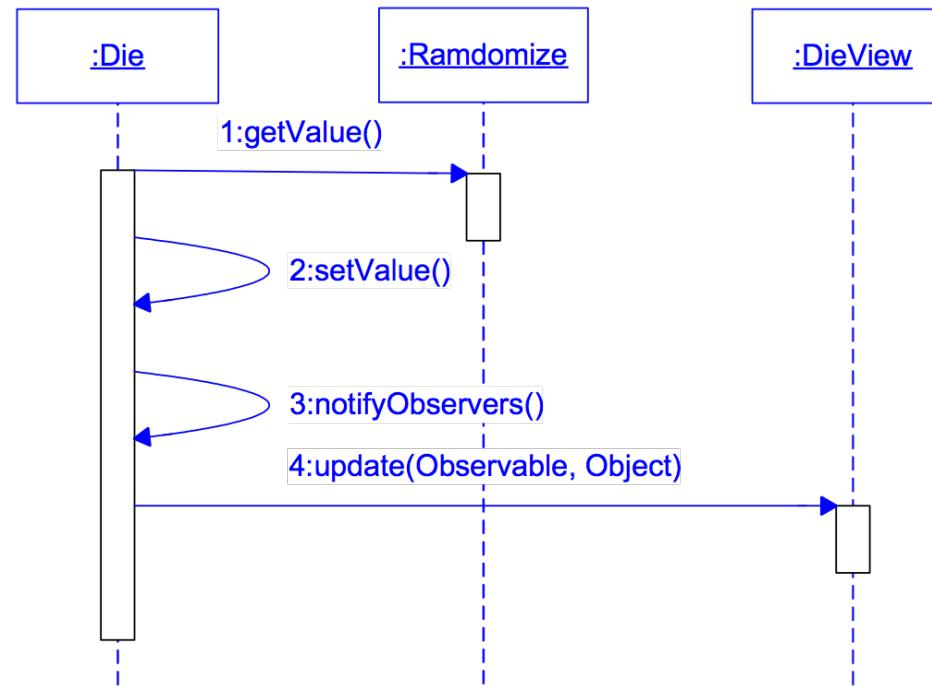
Case study

- User view are instances of `javax.swing.JPanel.java`



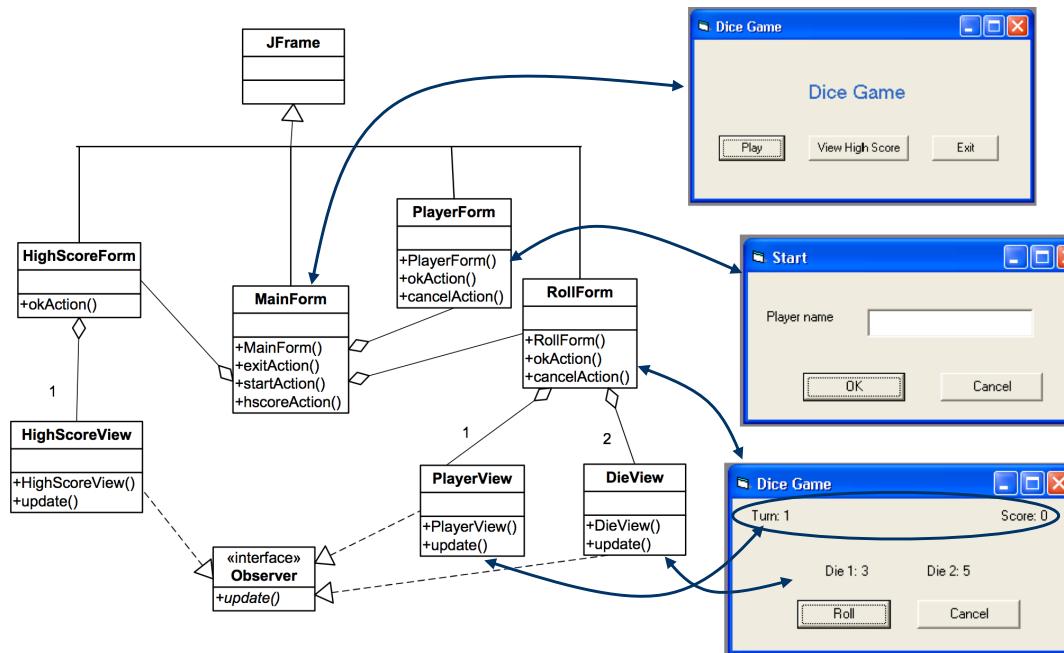
Case study

- Sequence diagram describes the interactions between **Die** object the its view



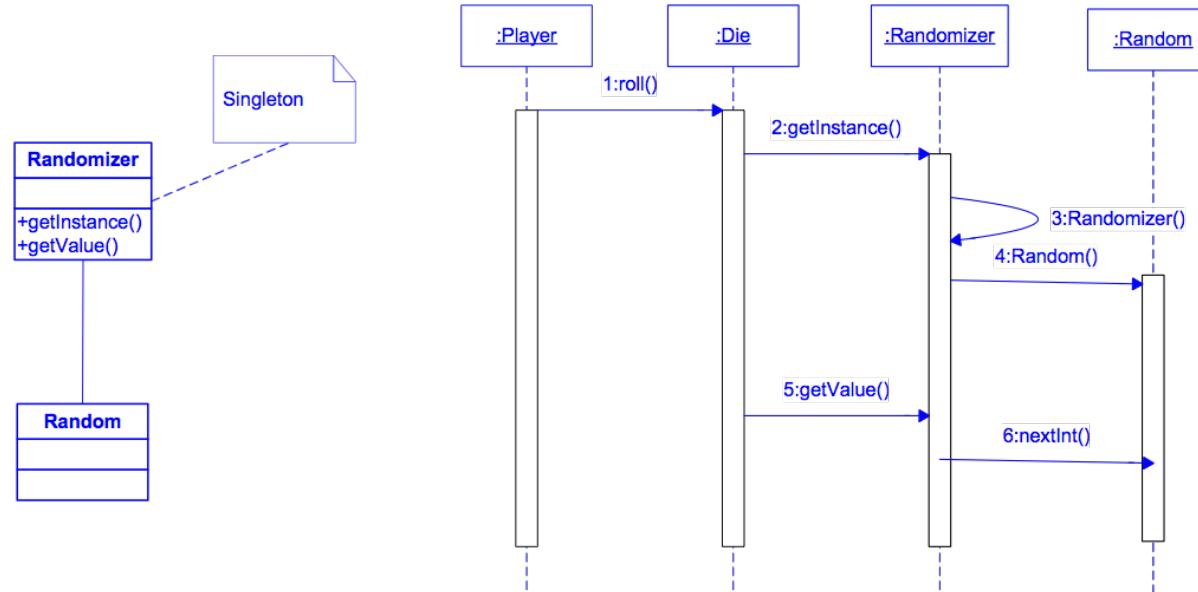
Case study

□ The design of “UI” package



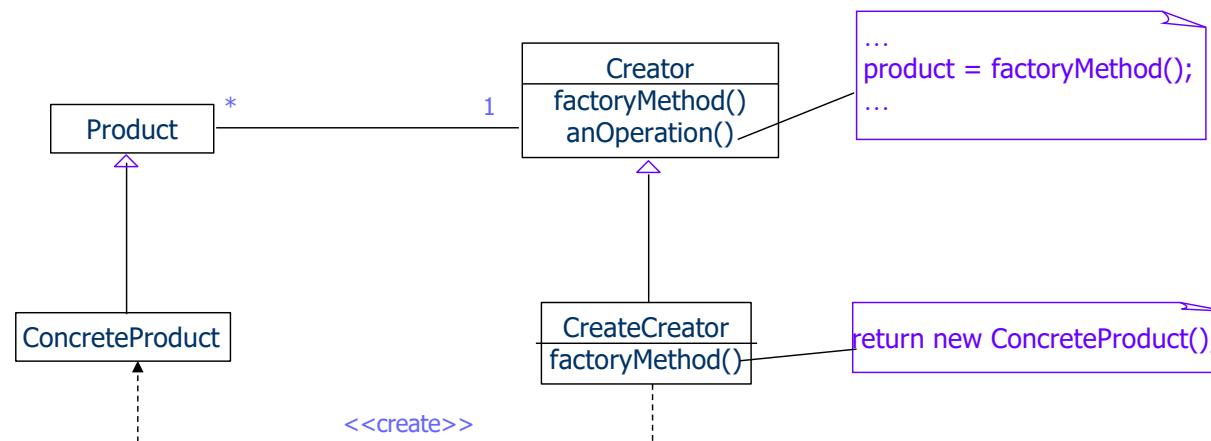
Case study

- The design of “Util” package



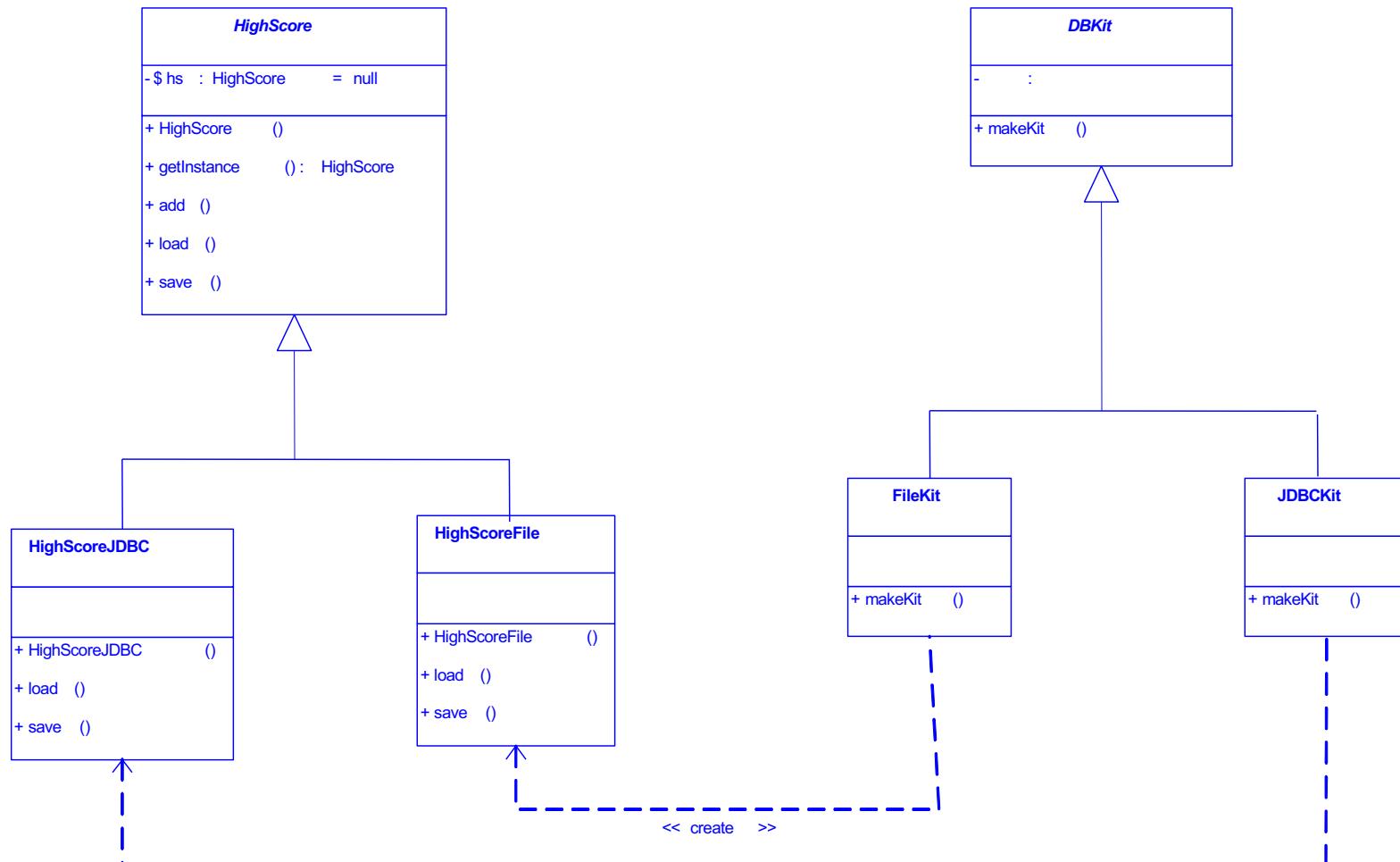
Case study

- The design of “DB” package
 - How to ensure the independence between “Core” and “DB” package
 - In order to be able to use several persistence types
 - File (serialisation)
 - Relation Database Management System (via JDBC)
 - Use **FactoryMethod** design pattern



Case study

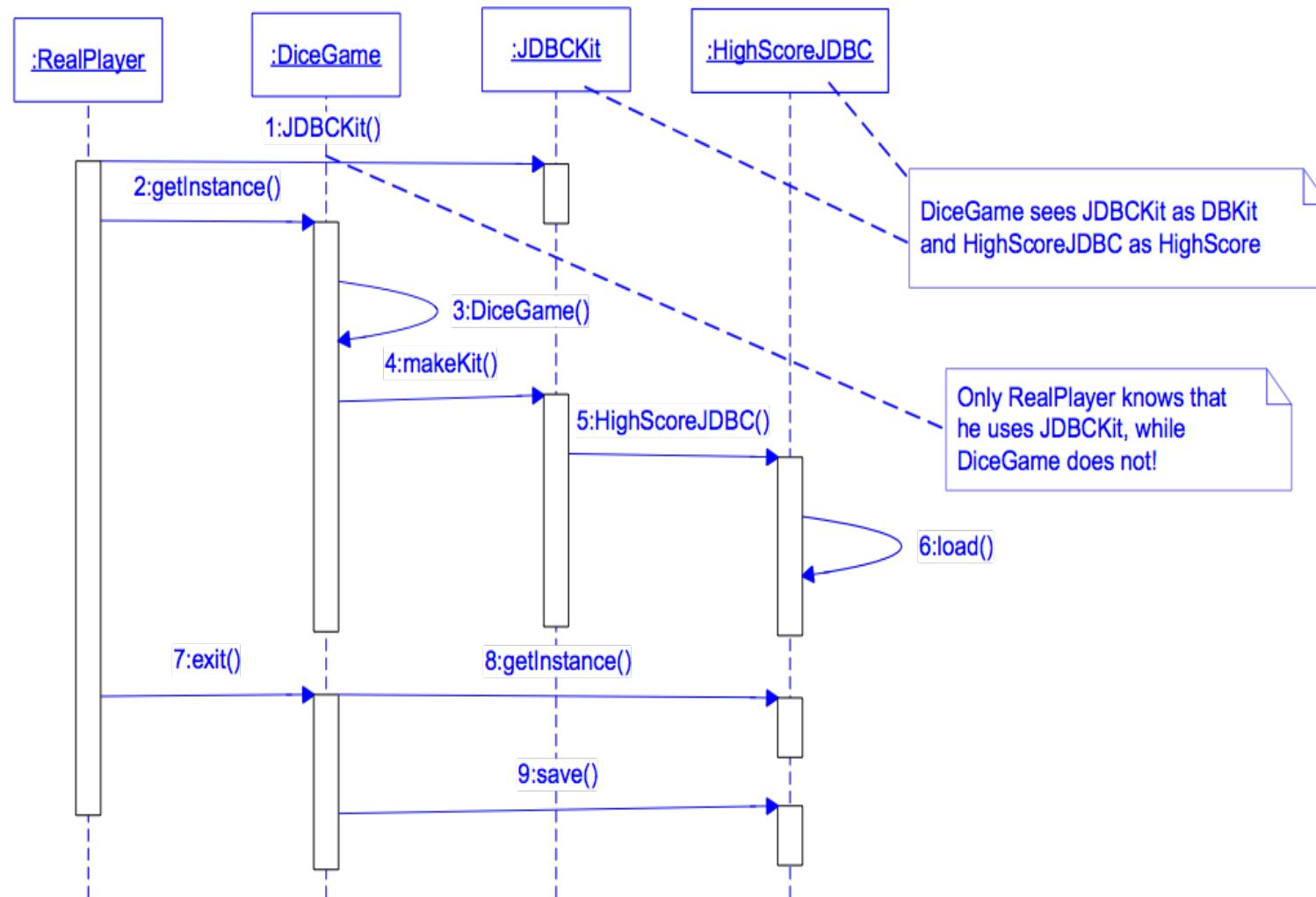
- The design of “DB” package
 - Class diagram



Note : HighScore class is a Singleton

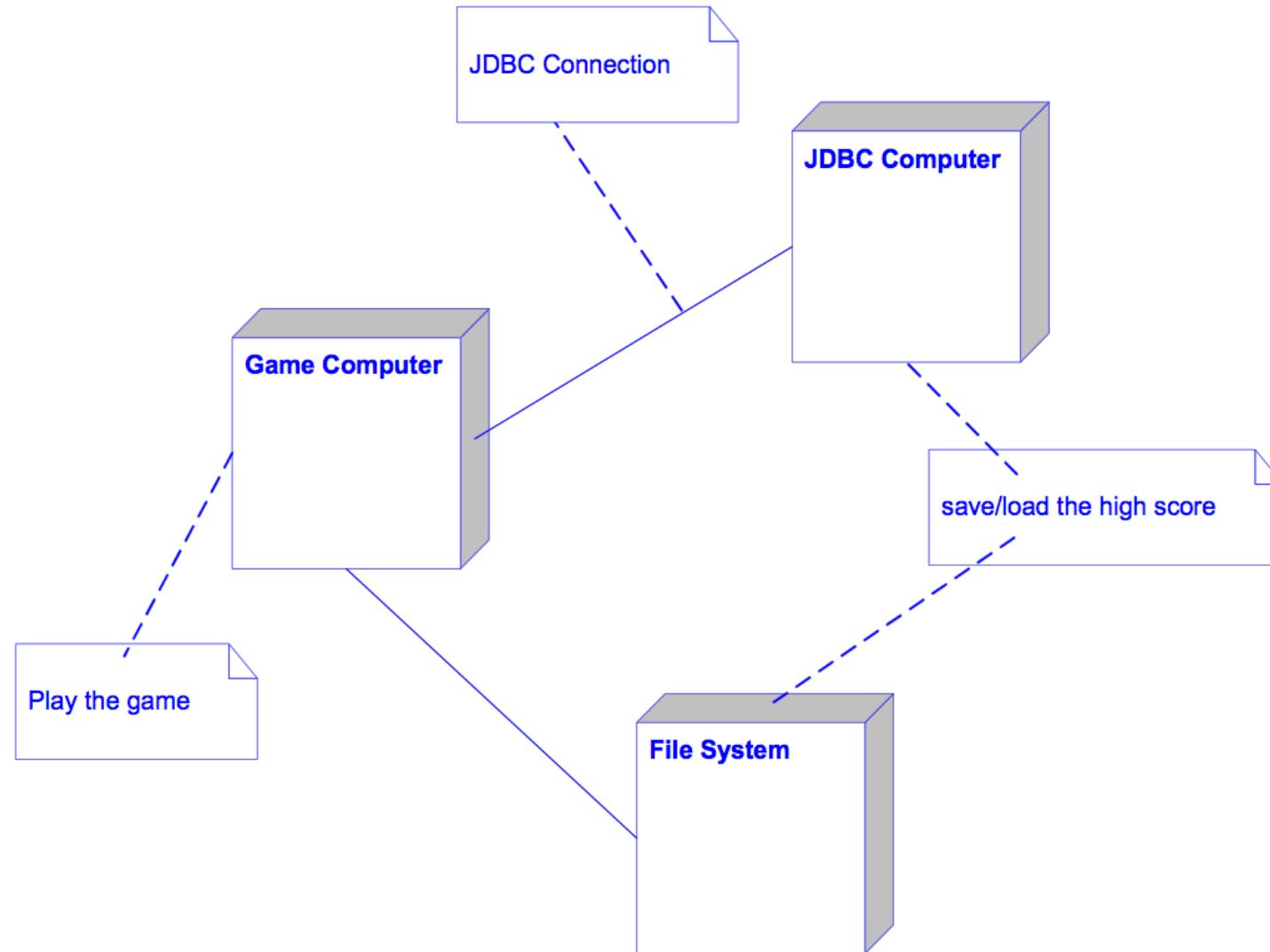
Case study

- The design of the “DB” package
 - Sequence diagram



Case study

- Deployment diagram



Main Activities of Software Development

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

Deployment

Installation and training

Maintenance

Post-install review
Support docs
Active support

Case study

- Complete the interaction diagrams
- Generate the code

Conclusions

Conclusions

- Distinction between functional approach and object-oriented approach
 - Master the basic object-oriented concepts
-
- UML: a modelling language
 - Need a development process
 - Different views
 - Different models
 - Use of the models in different development activities
-
- Master the main diagrams
 - Use-case diagram
 - Class diagram
 - Interaction diagram

Conclusions

- The UML concepts can be extended
 - The extensions
- Transformation of models to code
 - Models independent of programming language
- The automatic code generation is only a supplement
 - The models guide the coding process
- Master design principles
 - GRAPS principles/patterns
 - Some design patterns