Modelling, Object Orientation, UML

Based on IBM Rational materials

Objectives

- To identify the different Unified Modeling Language (UML) diagrams and their key purposes and characteristics
- To explain the benefits of UML diagrams during the different phases of development

Outline

- Modeling principles
- UML Introduction – What is UML and why is it useful for SE?
- Basic Object Oriented Concepts – What is OO and how is it linked to UML?
- UML and the Development Process
- An Example Process – The Traditional Object Oriented Method

Modelling

Why Model?

- Modeling achieves four aims:
  - Helps you to visualize a system as you want it to be.
  - Permits you to specify the structure or behavior of a system.
  - Gives you a template that guides you in constructing a system.
  - Documents the decisions you have made.
- You build models of complex systems because you cannot comprehend such a system in its entirety.
- You build models to better understand the system you are developing.

Four Principles of Modeling

- The model you create influences how the problem is attacked.
- Every model may be expressed at different levels of precision.
- The best models are connected to reality.
- No single model is sufficient.

UML Introduction

What is the Unified Modeling Language? - 1

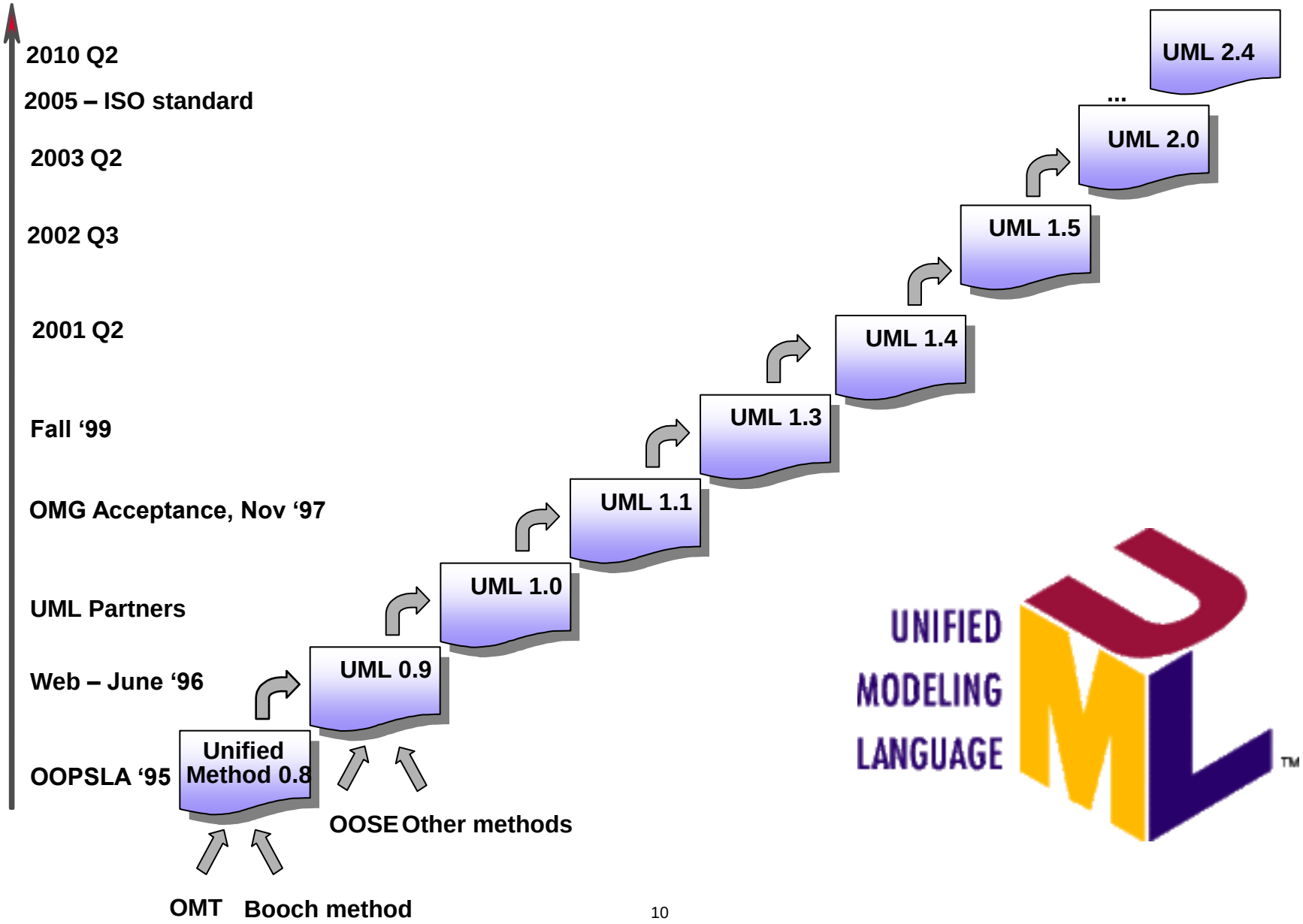
UML is an object-oriented language for:

- **Visualizing:** graphical models with precise semantics
- **Specifying**: models are precise, unambiguous and complete to capture all important Analysis, Design, and Implementation decisions
- **Constructing**: models can be directly connected to programming languages, allowing forward and reverse engineering
- **Documenting**: diagrams capture all pieces of information collected by development team, allowing one to share and communicate the embedded knowledge

What is the Unified Modeling Language? - 2

- UML is a standard from the Object Management Group (OMG)
- UML is Object Oriented
- Forms the foundation for Model-Driven Architecture (MDA)
- Model / view paradigm
- Target language independent.
- Based on the Meta Object Facility (MOF) Specification by the OMG

A brief history of UML



A unified view of the system

- UML provides a common language to simplify collection of system artifacts and communicate them between the different stakeholders.

Goals of UML

- Provide a common language that can be used by all stakeholders
- Its main purpose is to allow stakeholders to communicate
- It is mainly a graphical language because people understand more from pictures than from words
- Integrate best engineering practices that have proven successful in the modeling of large and complex systems
- Be independent of particular programming languages and development processes
- Help project teams easily experiment to explore multiple solutions

UML and modeling

- UML is for visualizing, specifying, constructing and documenting the components of software and non software systems

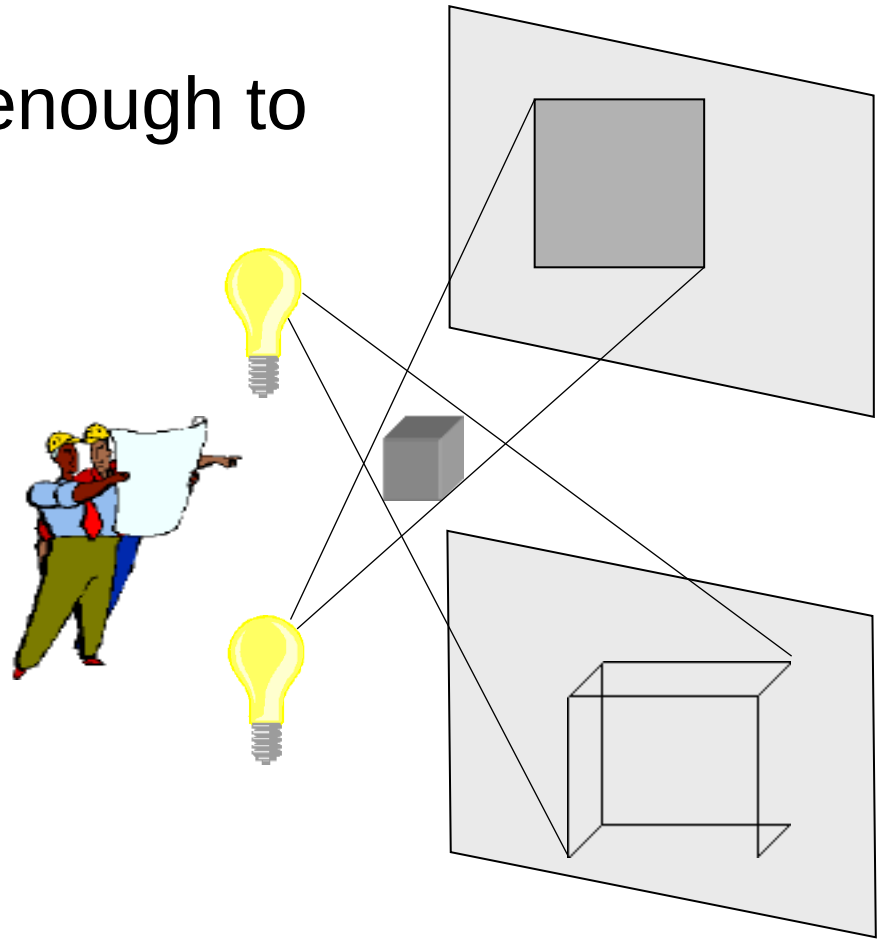
- Efficient and appropriate use of notations is very important for making a complete and meaningful **model**.

- The model is useless unless its **purpose** is depicted properly.

- We prepare UML diagrams to understand a system in better and simpler way.

One system - Several views

- Complex systems cannot be modeled clearly and completely in a single view.
- A single diagram is not enough to cover all aspects of the system.
- UML provides a set of diagram types, each type offering a different view of the system.



Models used to describe the system-1

- UML defines the semantics and notation for the following models:
  - **The Use Case Model** - describes the boundary and interaction between the system and users (Use Case Modeling)
  - **The Logical Model** - describes the classes and objects that will make up the system (Logical Modeling)
  - **The Interaction Model** - describes how objects in the system will interact with each other to get work done (Interaction Modeling)

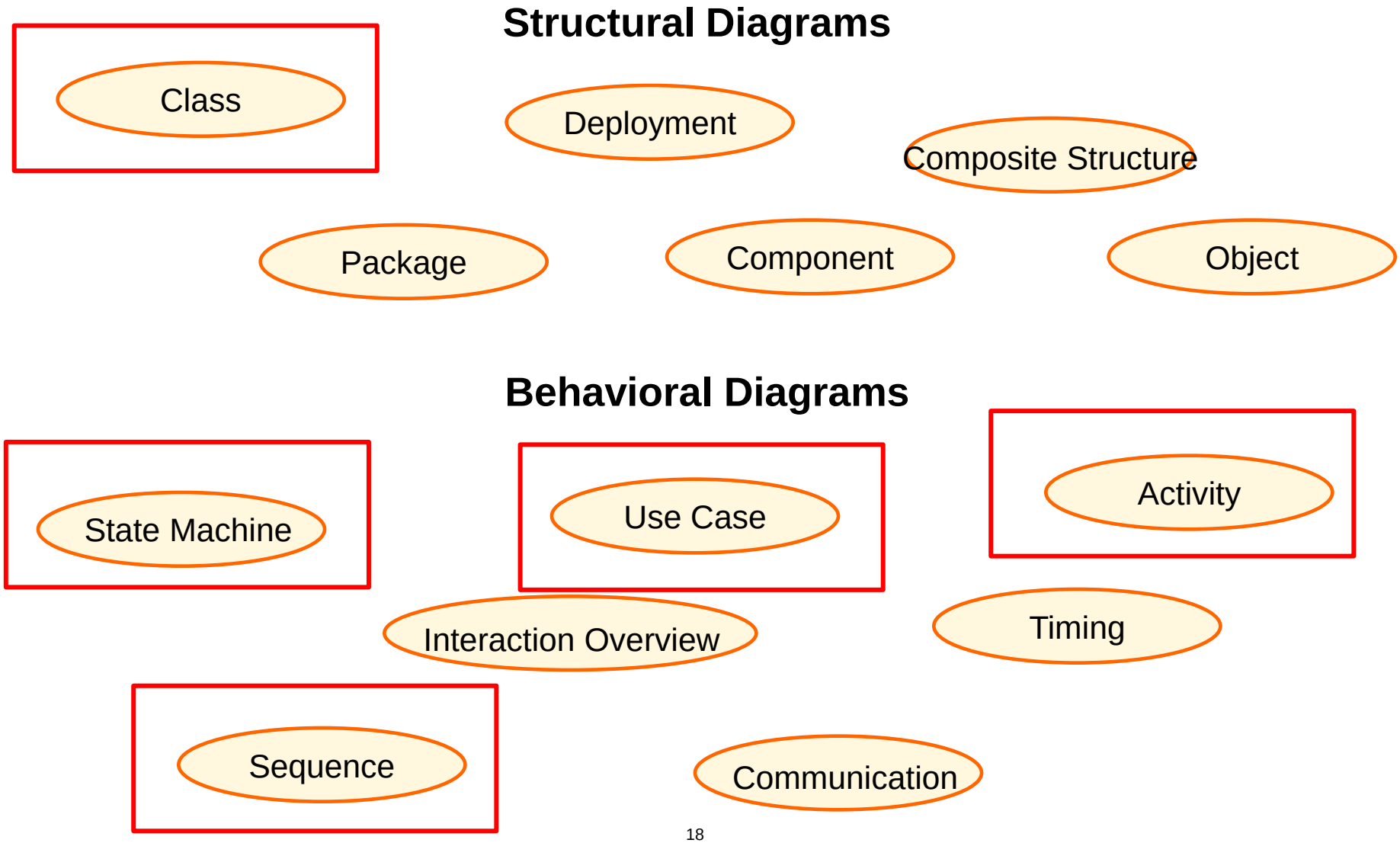
Models used to describe the system-2

- **The Physical Component Model** - describes the hardware and software that make up the system (Physical Component Modeling)
- **The Dynamic Model** - State Machines describe the states or conditions that classes assume over time. Activity graphs describe the workflows the system will implement (Dynamic Modeling)
- **The Physical Deployment Model** - describes the physical architecture and the deployment of components on that hardware architecture

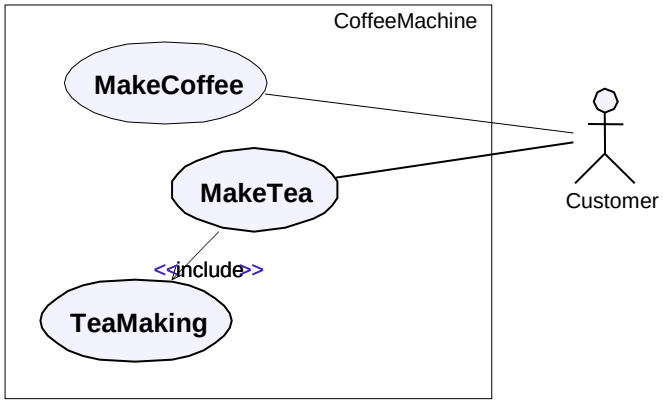
UML diagrams

- All the diagrams have symbols
  - Which may be also be shown as bitmaps
- Symbols contain text
- The symbols are often linked with lines
  - Relationships
- The relationships have different line styles and shapes at the ends
  - For different meanings
- Relationships sometimes have text and numbers on them

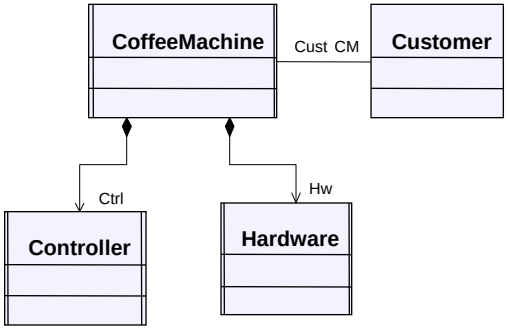
The thirteen (13) UML 2.0 diagrams



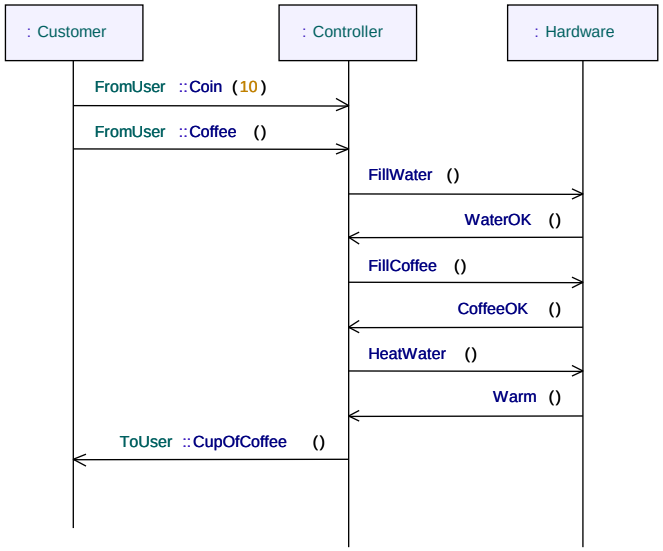
UML 2.0 diagrams



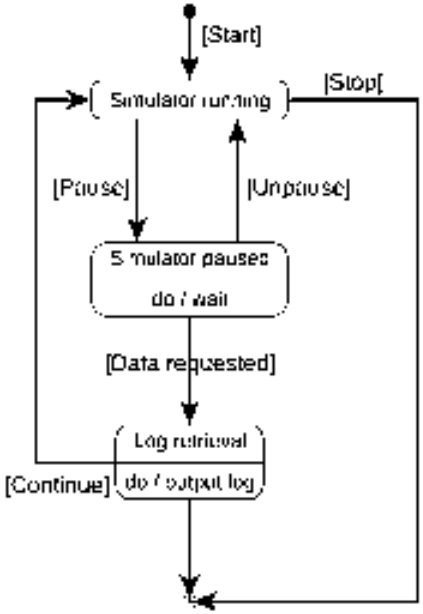
Use Case Diagram



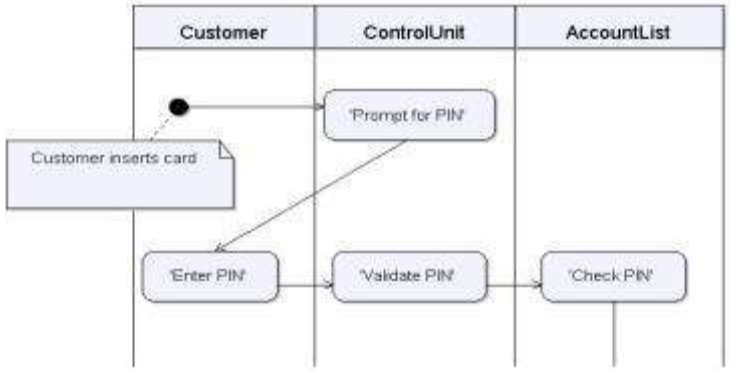
Class Diagram



Sequence Diagram



State Diagram



Activity Diagram

Object orientation and UML

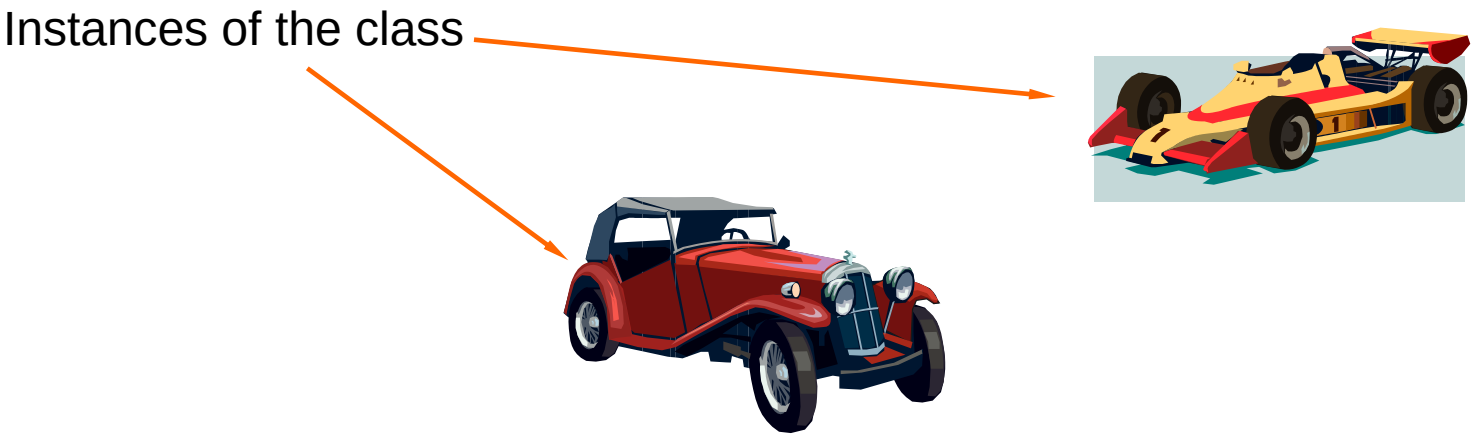
- UML assumes that an OOM of the system is required, i.e. the system can be created using objects
- What is an Object?
  - An object is something that is identifiably separate from other objects e.g. a car
  - Objects can be described – they have properties e.g. a car has color, number of gears
  - Object can do things – e.g. a car can move
- Virtually anything can be effectively described by identifying its characteristics, or properties and its behavior, or operations:
  1. Properties - describe the **state** of the object
  2. Operations - describe what the object can do. They affect the state of the object itself or other objects

Principal OO concepts

- Classification and Abstraction
- Encapsulation and information-hiding
- Modularity
- Hierarchy

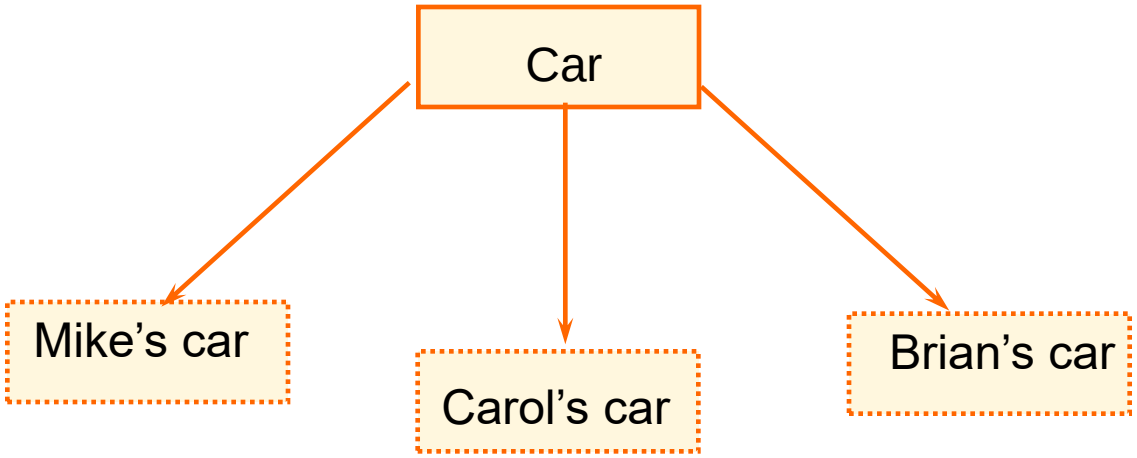
Classification - 1

- Classifying involves identifying identical properties and operations and grouping them into a single entity.
- A class is a template/blueprint, composed of a set of properties and operations.
- A car is a class.
  - Properties: wheels, passenger compartment, engine
  - Operations: accelerate, turn, illuminate street



Classification - 2

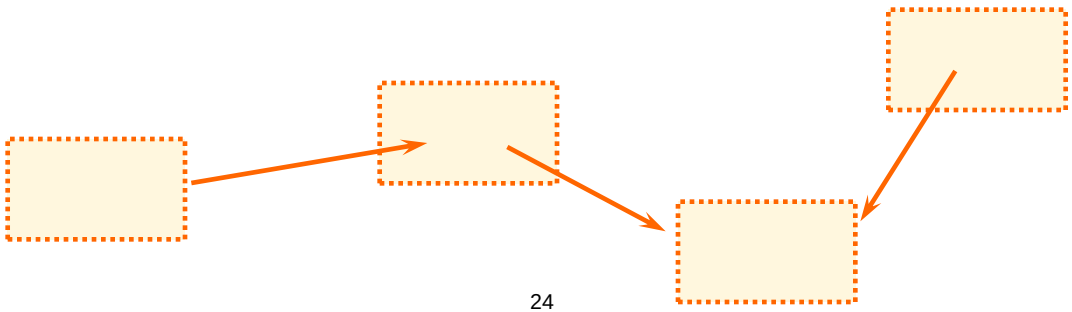
- An object is generated by instantiating a class
- Every object is an instance of a class
- All instances of a given class are structurally and behaviorally identical



Objects interact

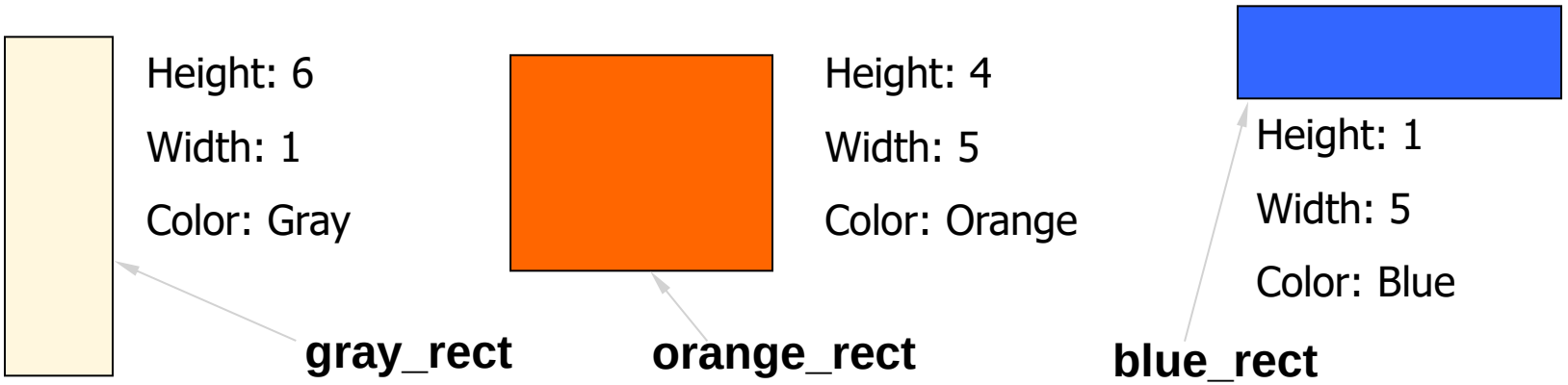
An object-oriented system is regarded as a network of cooperating objects which:

- Are related to (or are aware of) each other in some way.
- Interact by sending each other messages. Objects need to interact in order to perform the overall functions of the system. If objects do not interact, nothing happens
- Maintain their own state
- Have an individual identity



Class and instance example

- Class Rectangle
  - Properties: Width, Height, Color
  - Operations: Change-Height, Change-Color
- Objects or Instances of the class Rectangle
- In UML, an object is represented in text as:
  - gray_rect:Rectangle
  - orange_rect:Rectangle
  - blue_rect:Rectangle



Abstraction

- Abstraction is the technique of focusing on essential aspects of a problem and ignoring unnecessary details
- Abstraction must always be for some purpose, because the purpose determines what is and is not important
- Many different abstractions of the same thing are possible, depending on the purpose for which they are made

What kind of class is this?

- Properties
  - Number of books
  - Number of CDs
  - Location
- Operations
  - Borrow book
  - Return book
  - Reserve a book

Classes are abstract descriptions

- The group of descriptions and operations on the previous slide is an abstract description of a library. It only includes the important characteristics of a library and not every detail

- It is not possible, or necessary to give every operation and property to identify the class as a library

- One of the most difficult questions to answer is what properties and operations should be used to describe a class

How many properties and operations are needed?

- Enough to deal with the problem at hand – no more – no less.
- Determining this group is not always easy. Some will be more obvious than others. It depends on the problem to be modelled
- At the beginning it is important to get the obvious properties and operations correct but it’s not a disaster to get it a bit wrong. Iteration will be used to repeat the process of class definition to ensure that all detail required is captured

Encapsulation and information hiding

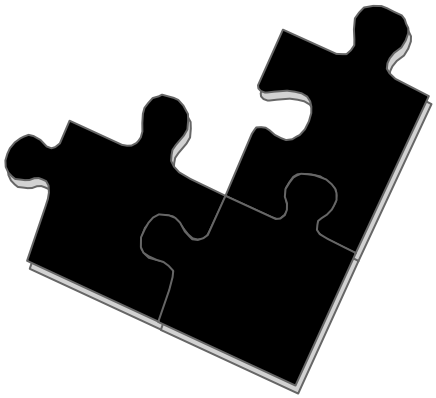
- Encapsulation is the technique of keeping implementation details from being externally visible
- Attributes and behaviors are encapsulated to create objects
- Encapsulation prevents a program from becoming so interdependent that a small change has massive ripple effect
- The implementation of an object can be changed without affecting the applications that use it

Encapsulation – Example

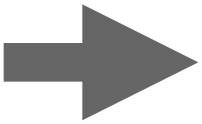
- When you change the channel on your TV you do not need to know how the TV actually does this. The details of how the TV does this are hidden from you.
- This is a form of **encapsulation**. All you need to know is how to use the interface to the TV.
- Encapsulation – it is about hiding information. By information we mean: details of how operations are performed.

Modularity

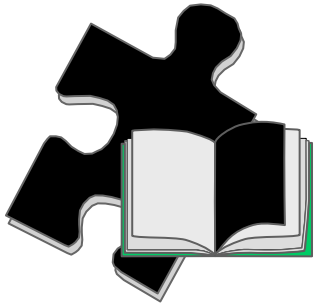
- Breaking complex system into smaller modules



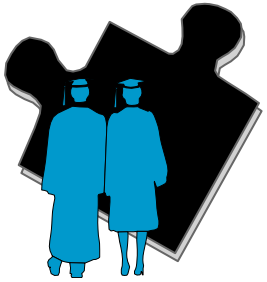
Course Registration System



Billing System



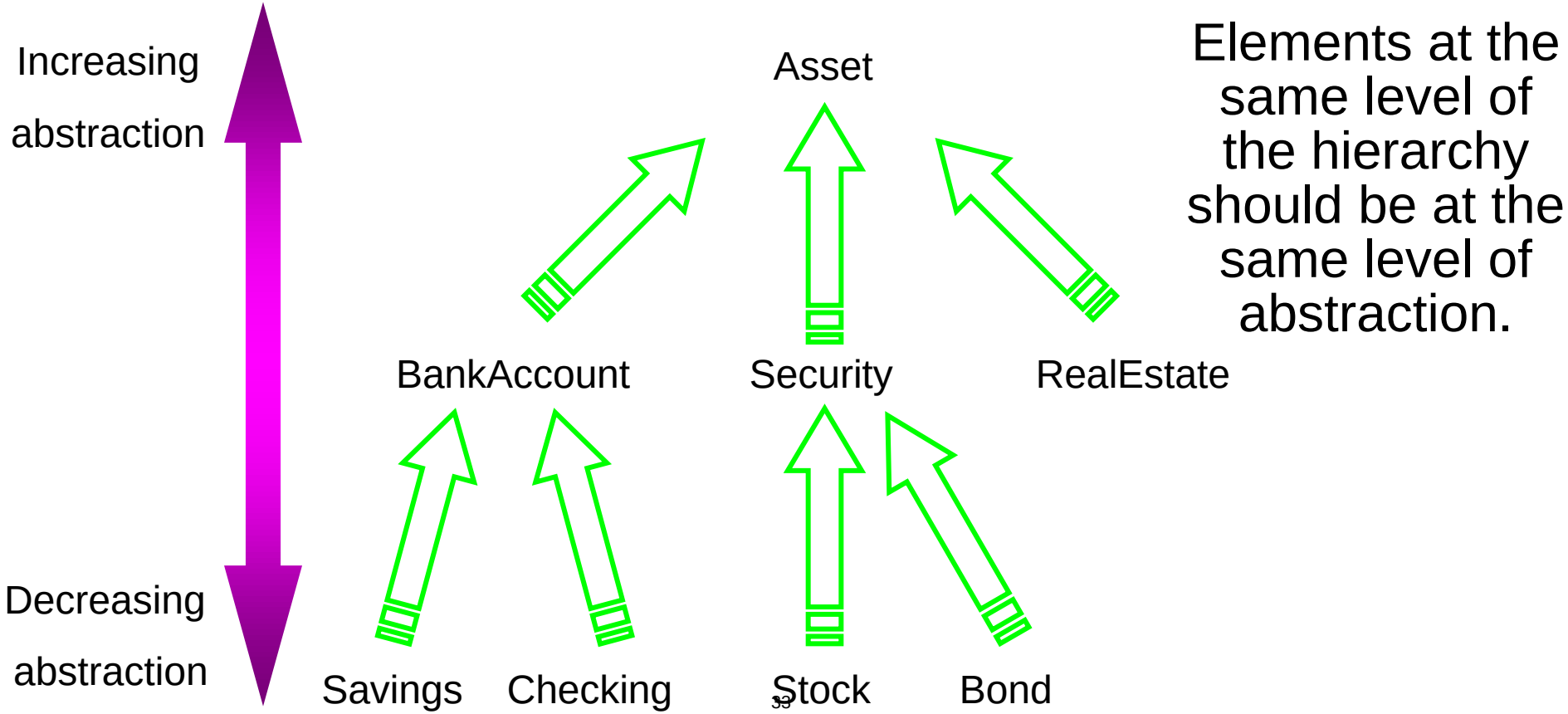
Course Catalog System



Student Management System

Hierarchy

- Any ranking or ordering of abstractions into a tree-like structure.
  - . Kinds: Aggregation hierarchy, class hierarchy, containment hierarchy, inheritance hierarchy, partition hierarchy, specialization hierarchy, type hierarchy.



Elements at the same level of the hierarchy should be at the same level of abstraction.

What Is Generalization?

- A specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). (*The Unified Modeling Language User Guide*, Booch, 1999)

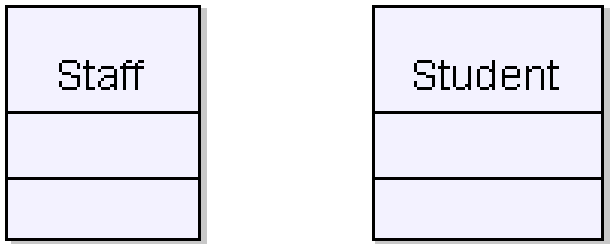
A synonym to inheritance

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

Generalization - 1

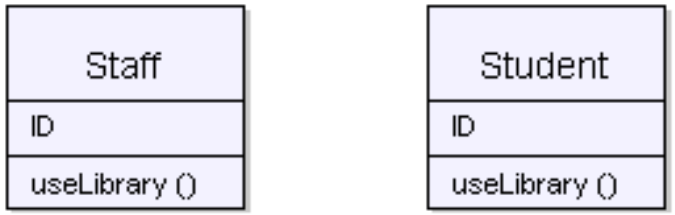
- Consider people who attend a university – in general they are either staff or students
  - Staff - employed by university, give lectures
  - Students - attend lectures

- Since there is a functional difference between the two types of people, we would represent them by two separate classes.



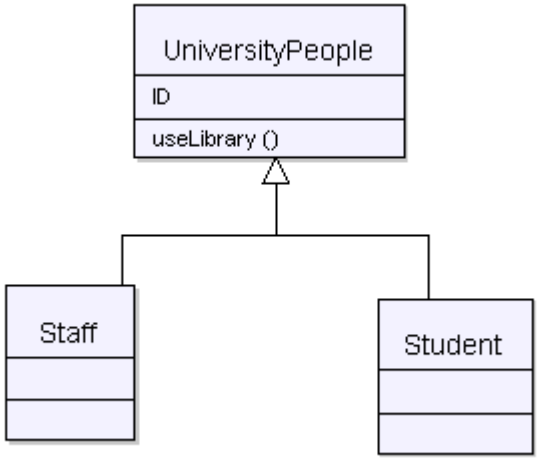
Generalization - 2

- However there can potentially be properties and operations that are common to both staff and students. For example:
  - Both staff and students have a unique ID number printed on their ID cards
  - Both staff and students can use the library



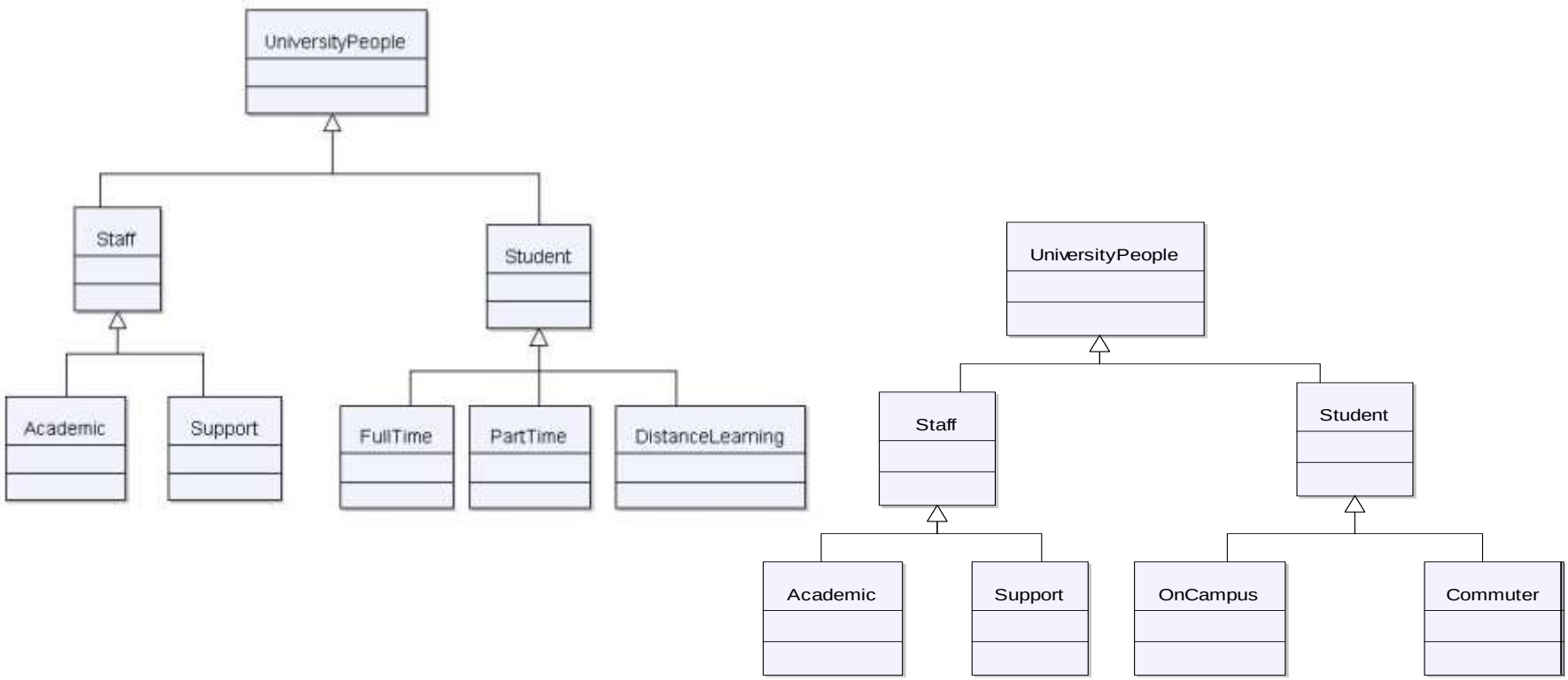
Generalization -3

- Create a class called UniversityPeople and put the common properties and operations into it
- Connect the Staff and Student classes to UniversityPeople using the generalization symbol
- The generalization symbol tells us that all the information held within the UniversityPeople class is common to the Staff and Student classes
- Generalization facilitates reuse and change control



Generalization - 4

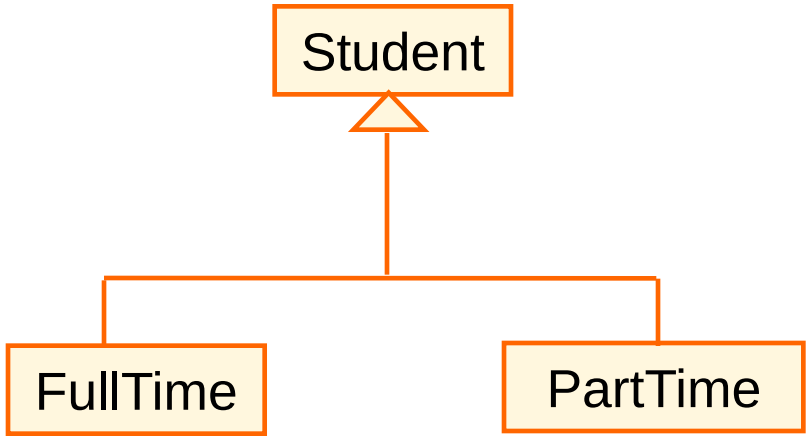
- We can continue to extend this structure based on the context of the university



- Properties and operations common to groups of classes are drawn down into the classes above

Generalization - Notes

- There are a number of important technical terms that are used when talking about generalization:
  - Student is a superclass to FullTime and PartTime. Superclasses define common (general) properties
  - FullTime is a subclass to Student (so is PartTime). The subclass adds incremental properties and operations
  - FullTime and PartTime inherit all the properties and operations from Student.



The Traditional OO Process

The Traditional OO process

- The Traditional OO Process has the following phases:
  - Requirements capture
  - Analysis
  - Design
  - Implementation
  - Test: not an afterthought. Testing is required at every phase
- This is the process that we use as an example
  - Simple to understand and to apply
  - Uses all UML concept
  - Real life processes could be different

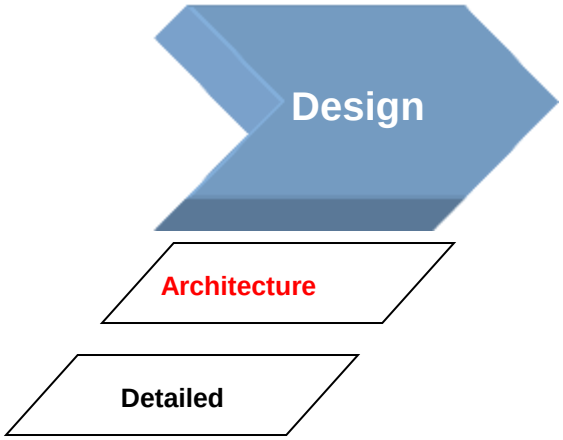
Requirements phase

- Understand the customer’s requirements and business need
- Develop the requirements of the system functionalities
- Identify both Functional and Non-Functional requirements
- Achieve agreement between stakeholders and suppliers of the system

Analysis phase

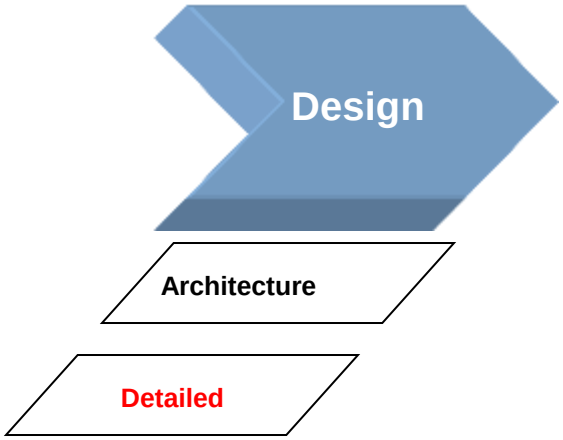
- Understand the problem domain
- Create a model of the problem domain
  - Free from any technical or implementation details
  - Detailed enough to allow simulation
- Common diagrams used in this phase:
  - Use Case diagrams
  - Sequence diagrams
  - Activity diagrams
  - Class diagrams

Architecture design phase



- Divide the system into logical subsystems / packages
- Define the interfaces to the subsystems
- Describe the interaction between subsystems
- Common diagrams used in this phase:
  - Package diagram
  - Class diagram
  - Composite Structure diagram
  - Sequence diagram

Detailed design phase



- Decompose the subsystems further
- Refine sequence diagrams
- Describe the detailed behavior of classes using statemachines
- Include all technical or implementation details
  - Detailed enough to allow easy implementation of the model
- Common diagrams used in this phase:
  - Class diagrams
  - Sequence diagrams
  - Composite Structure diagrams
  - State Machine diagrams
  - Deployment diagram

Implementation phase

- Implement the agreement between stakeholders and suppliers of the system
- Translate the UML model into a software language that can be compiled, and build the application for model verification
  - The more complete the model, the easier is the translation
  - Using a OO language would ease the task
  - Usually in an iterative fashion
- For systems and hardware projects, the UML model becomes the requirements/roadmap for actual implementation
- For software projects, the OO language allows translation into actual software code and application completion

Test Phase

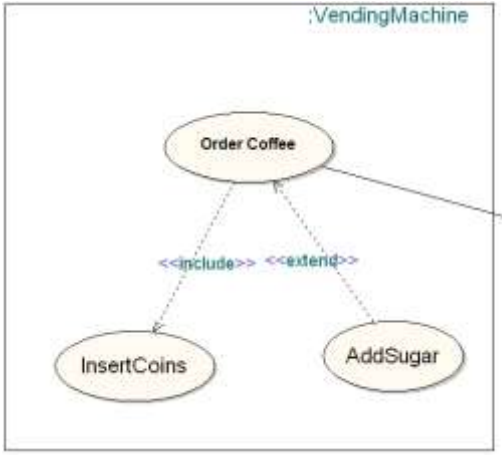
- Verify that the agreement between stakeholders and suppliers of the system is fulfilled
- Write the test cases and execute them against the implementation of the model
- Testing methods
  - Unit testing
  - Integration testing
  - System/Acceptance testing
  - Regression Testing
- This phase could and should be performed in parallel to the others

Analysis phase

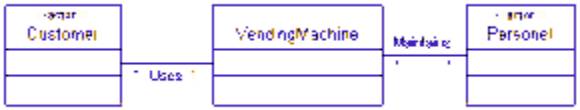
Analysis phase

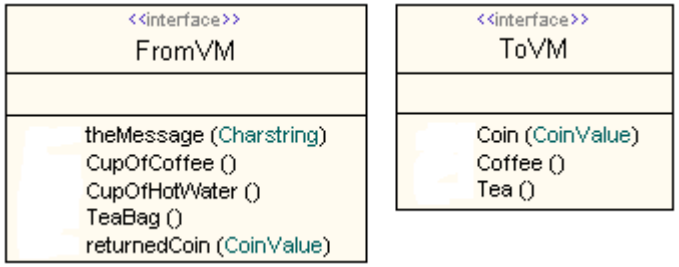
- Understand the problem domain
- Create a model describing the problem domain
  - Free from any technical or implementation details
  - Detailed enough to allow simulation on paper or by using a modeling tool
- Activities:
  - Acquire domain knowledge
  - Perform use case analysis
  - Define the necessary classes and relationships
  - Define the behavior and collaboration between classes
  - Test the resulting model

Deliverables



- Use Case Model with detailed use case descriptions
  - Natural language
  - Sequence diagram
  - Activity diagram
  - State Machine diagram
- Domain Model
  - Class diagram showing domain concepts and their relationships
- System Interface Class Diagram
  - Interface Classes
  - Signals that compose the interfaces
  - Data types used by the signals



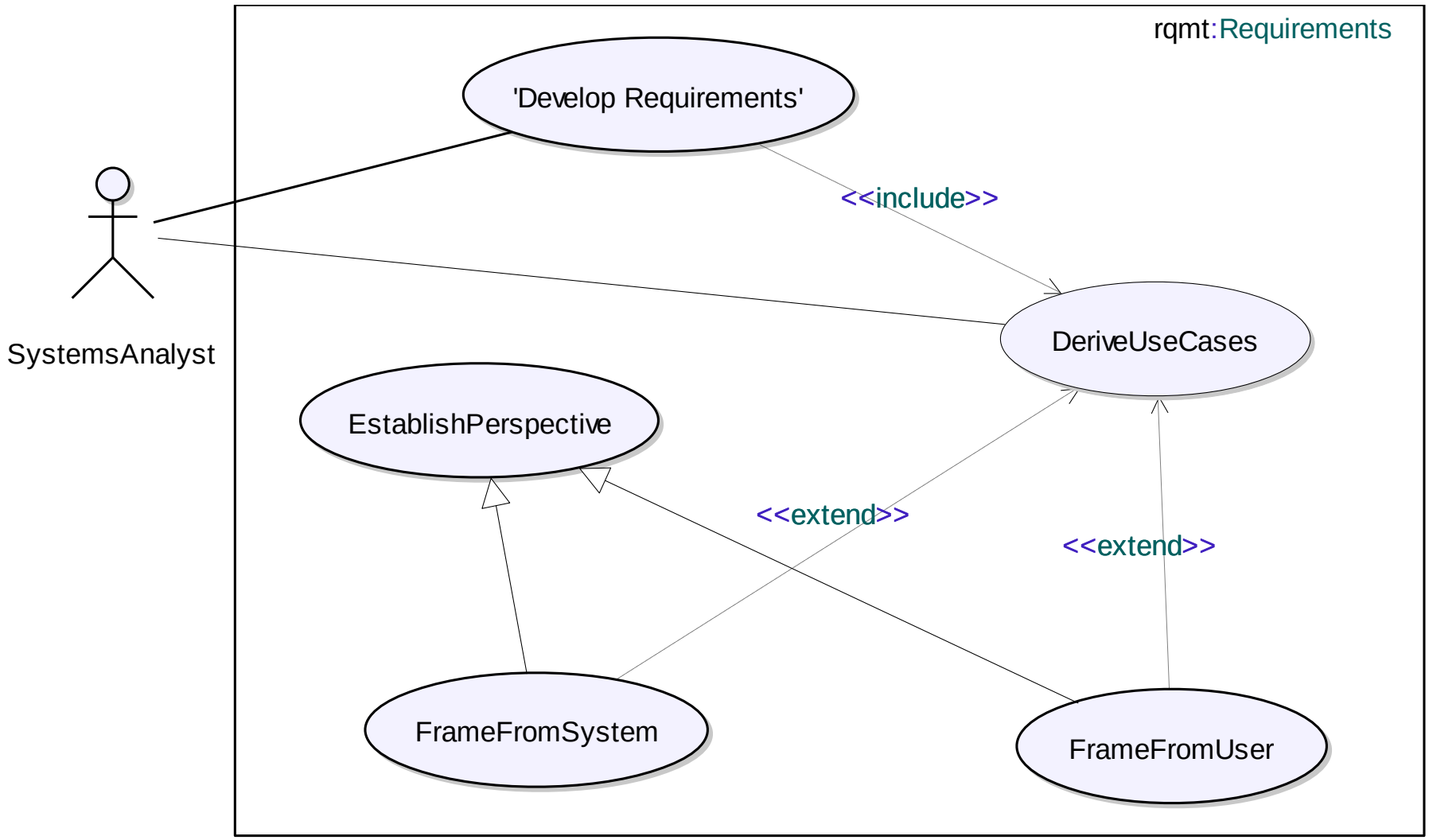


Use case model

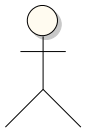
- The use case model is often used to allow the relevant stakeholders to approve the system to be constructed
- Therefore it must be written in a language that the stakeholders understand i.e. user domain language
- A view of a system that emphasizes the behavior as it appears to outside users. A use case model partitions system functionality into transactions (‘use cases’) that are meaningful to users (‘actors’)
- It will define the boundary between the system and external entities. The system is regarded as a black box and the functionalities are expressed from a user’s perspective (perspective (or from the system’s perspective, as long as the perspective is consistent)

51

Example


rqmt:Requirements
'Develop Requirements'
<<include>>
SystemsAnalyst
DeriveUseCases
EstablishPerspective
<<extend>>
<<extend>>
FrameFromSystem
FrameFromUser

Main concepts



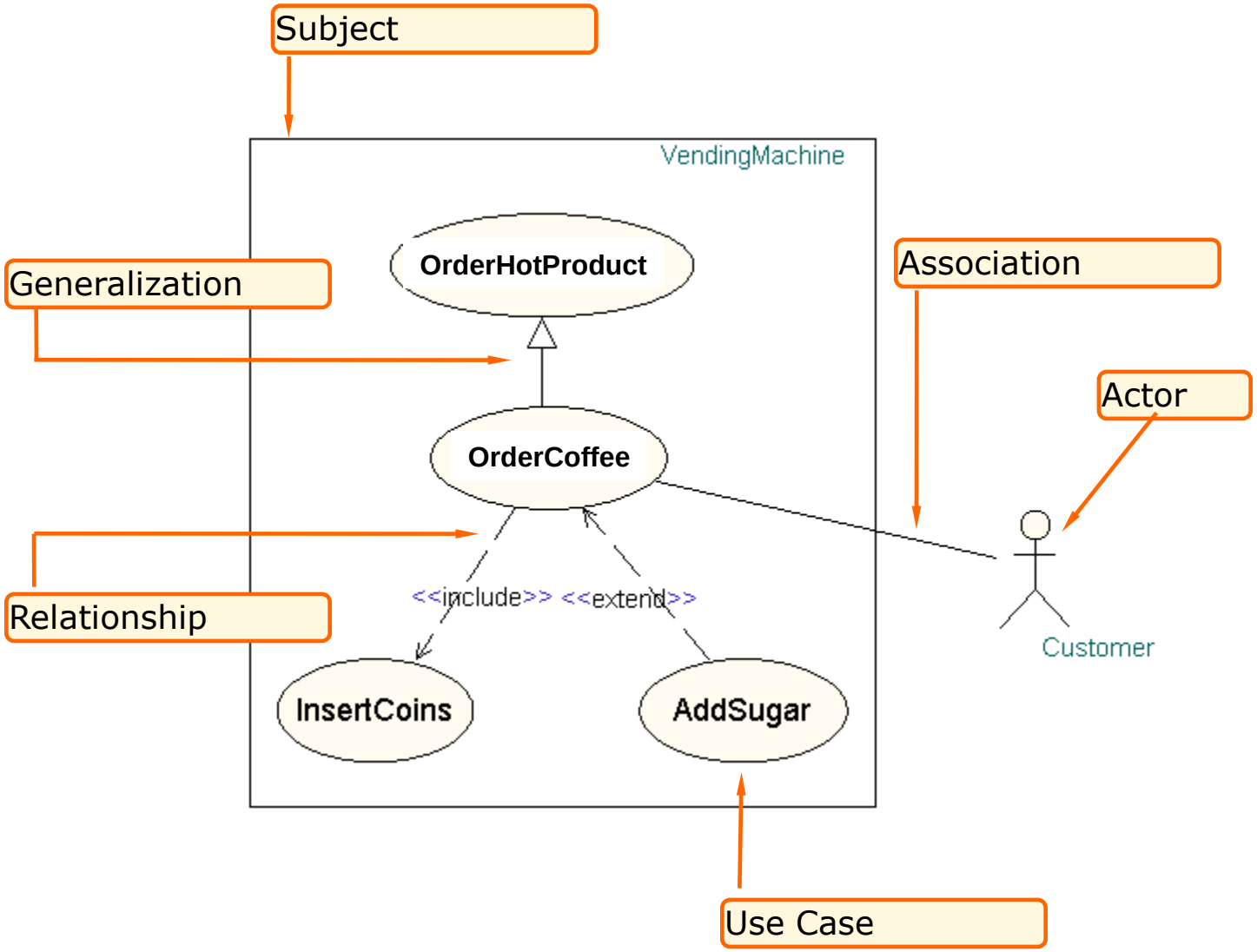
customer

- An actor represents the role of a user or a machine which interacts with a system.

- A Use Case is a piece of functionality that the system shall offer to an actor. The use case describes the interaction between one or more actors and the system.

Use Case Diagram

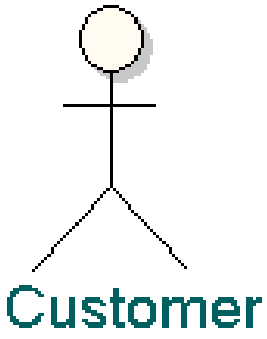
Use case diagram - Elements

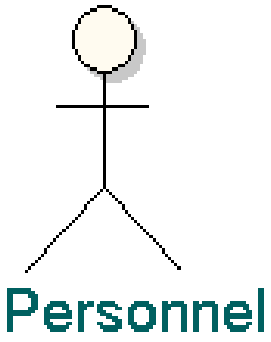


55

Actors

- A coherent set of roles that users of use cases play when interacting with the system
- Only depict actors that interact with the system

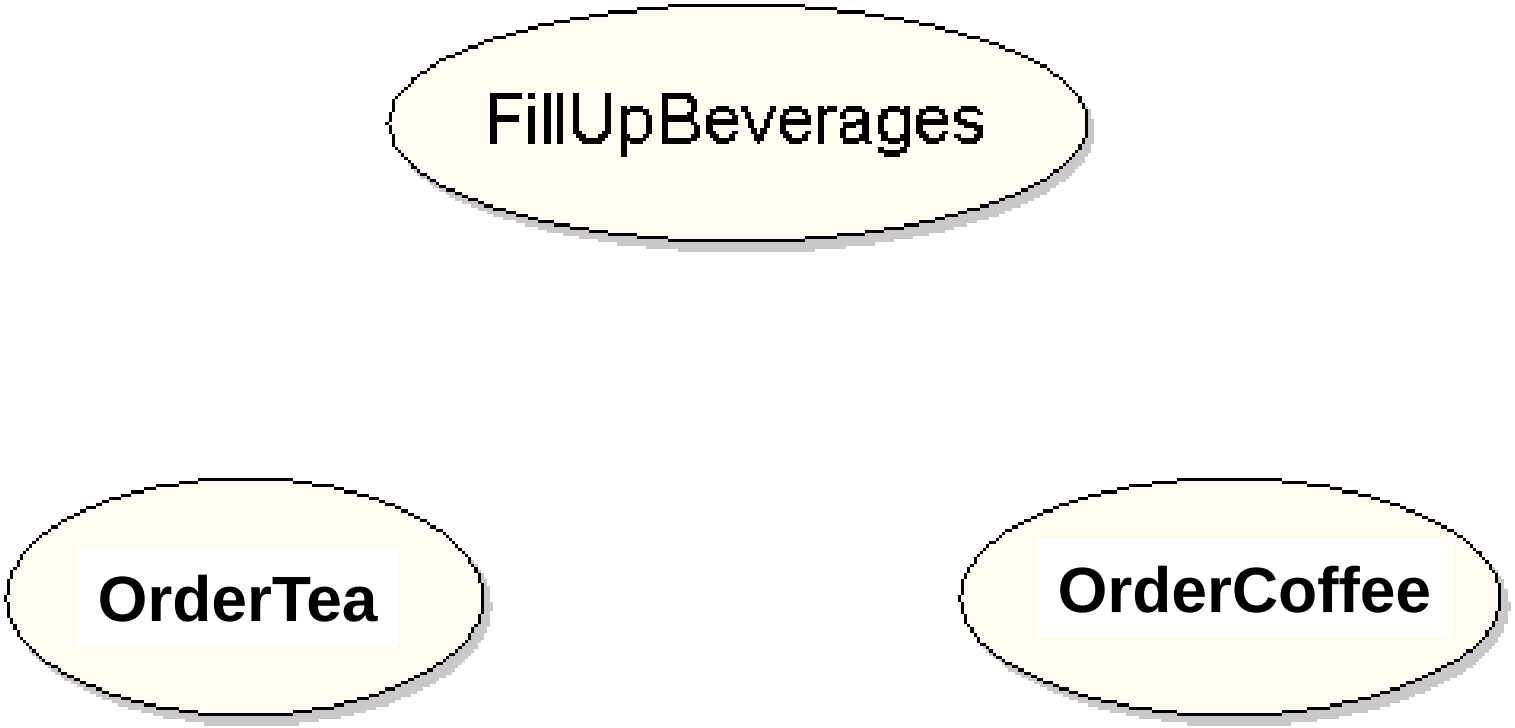




56

Use case

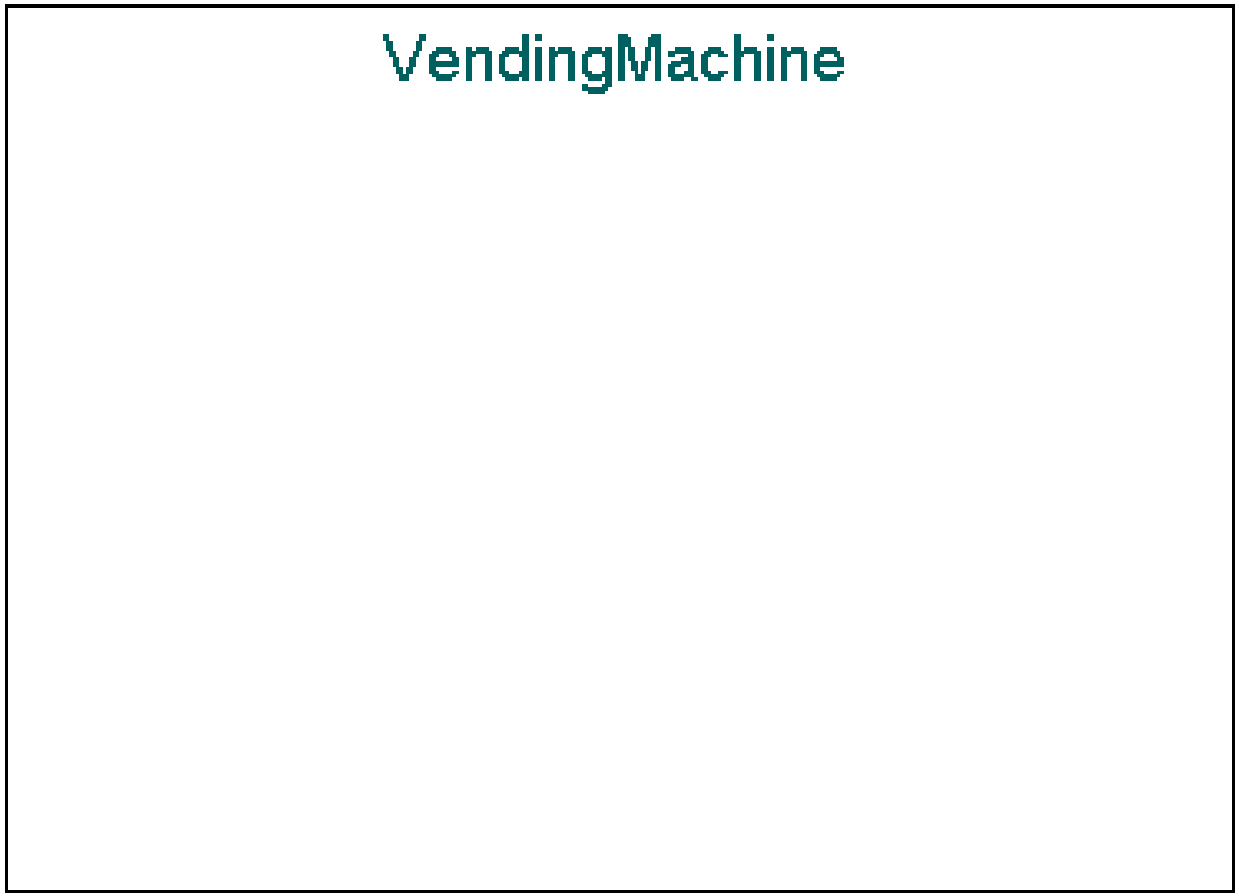
- A sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system



57

Subject (System boundary)

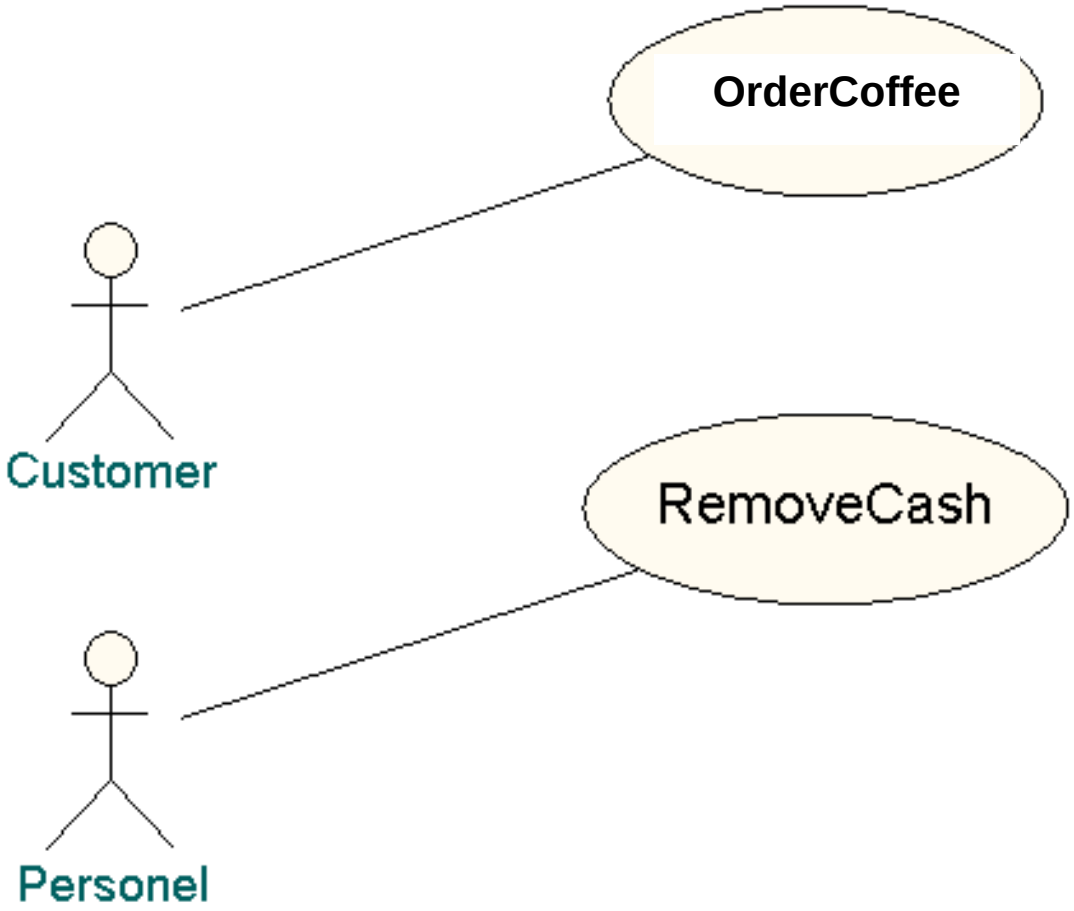
- Represents the boundary between the physical system and the actors who interact with the physical system.



58

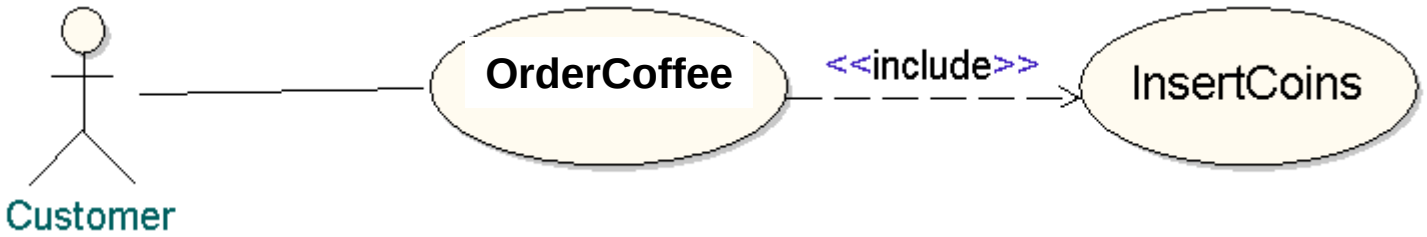
Association

- The participation of an actor in a use case. i.e., instance of an actor and instances of a use case communicate with each other.



Dependency – Include

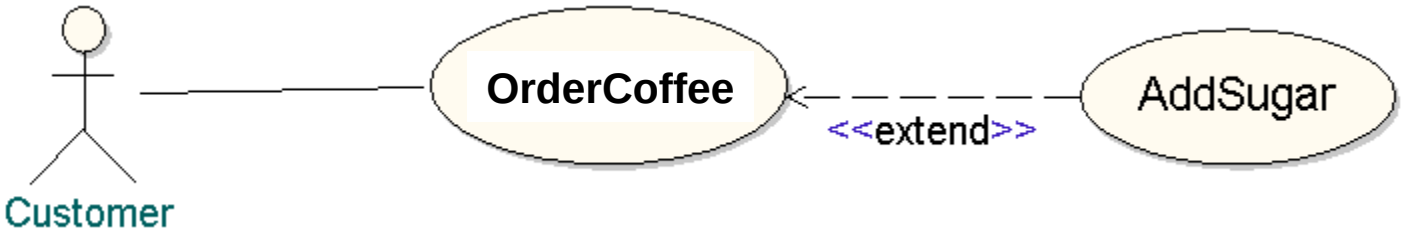
- A relationship from a base use case to an inclusion use case
- Implies that a use case “calls” another use case
- Primarily for reuse behavior common to several use cases
- The “calling” Use Case references the “called”. All conditions for the usage are stated in the “calling” Use Case, for example:
  - ”… The system then prints the report according to <Print> …”
  - ”… The system collects the coins according to <InsertCoins>, the system is now ready to …”



60

Dependency - Extend

- A relationship from an extension use case to a base use case
- Implies that behavior in one use case is an extension of the behavior in an other use case.
- Is used to model optional behavior.
- Conditions for how and when the extension is done is stated in the extending Use Case:
  - ”… if the user pushes the “add sugar” button, sugar is added according to <AddSugar> …”

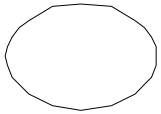


61

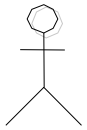
Summary symbols (Use cases)

- Subject (System Boundary)

- Use Case



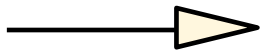
- Actor



- Association line

- Dependency (<<include>>/<<extend>>)

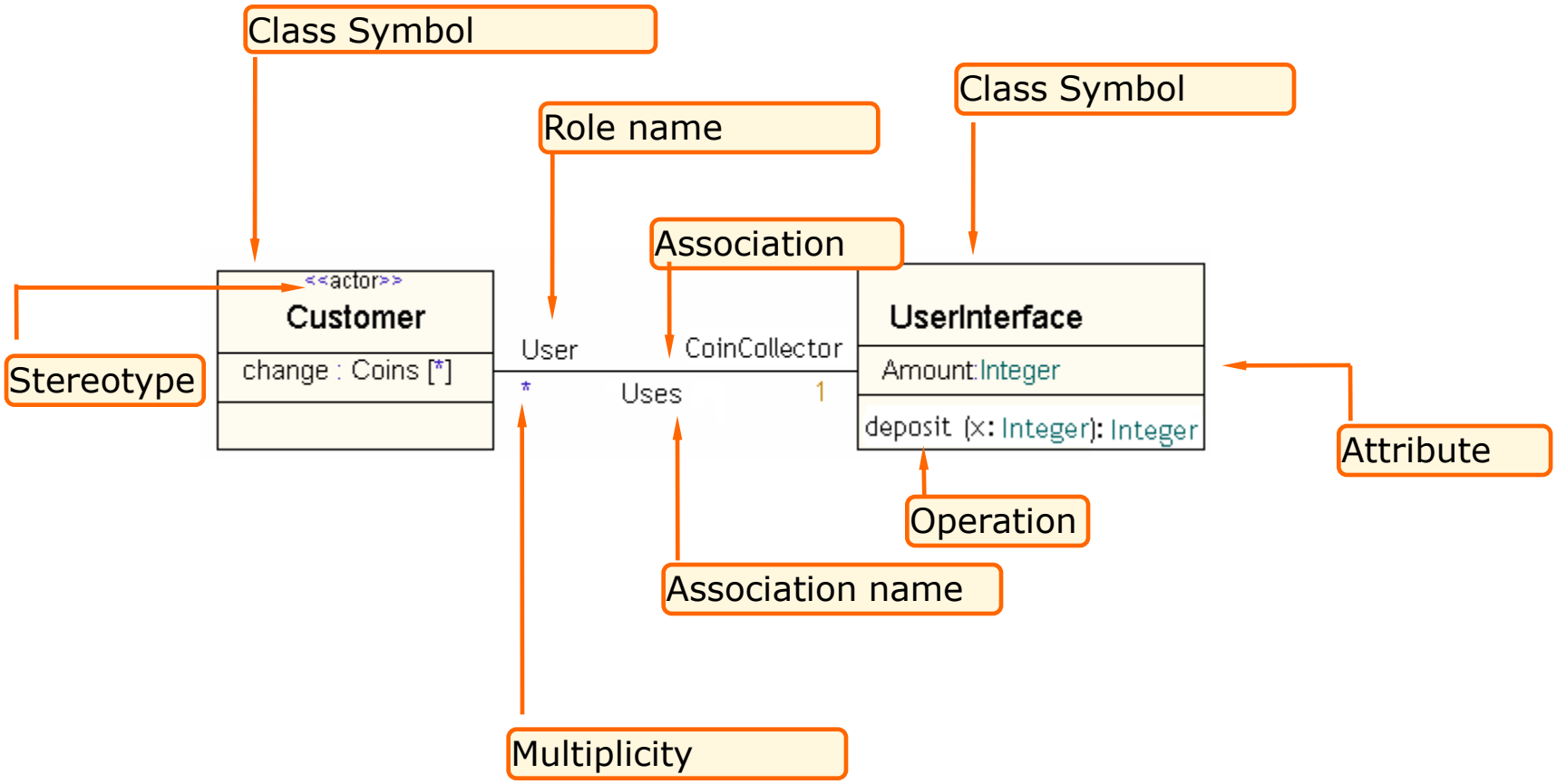
- Generalization



62

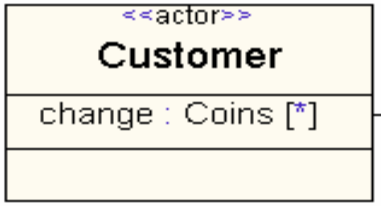
Class Diagram

Class diagram – Elements

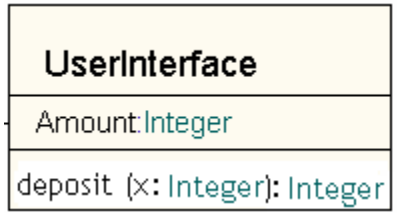


Attributes

- The static structure of a class is represented by properties that are called attributes

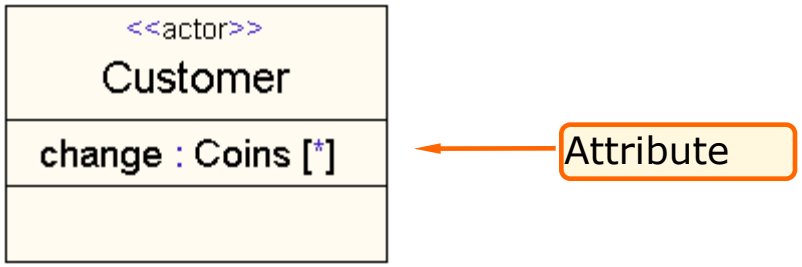


- Each customer has to have a set of coins

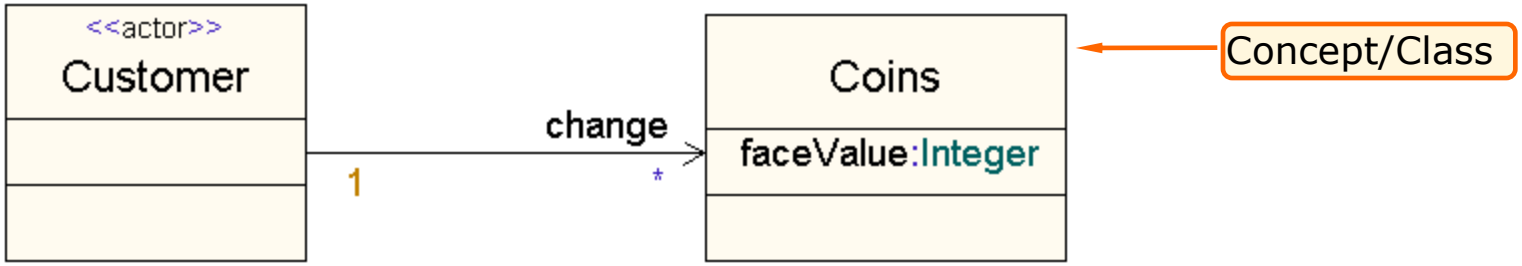


- The User Interface keeps track of the inserted amount

Attributes or classes?

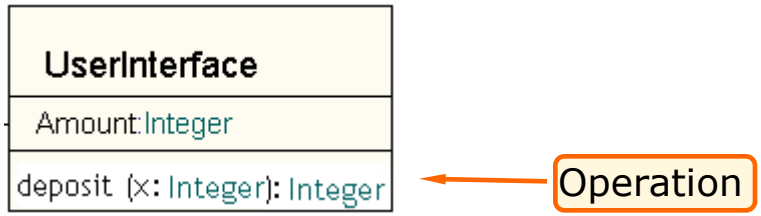


- Is the attribute itself a class or a simple attribute?
  - Think of the attribute first as a class
  - Search for its attributes or operations
  - If impossible to find attributes or operations, then it's a simple attribute
  - But if in doubt, make the attribute a class



Operations

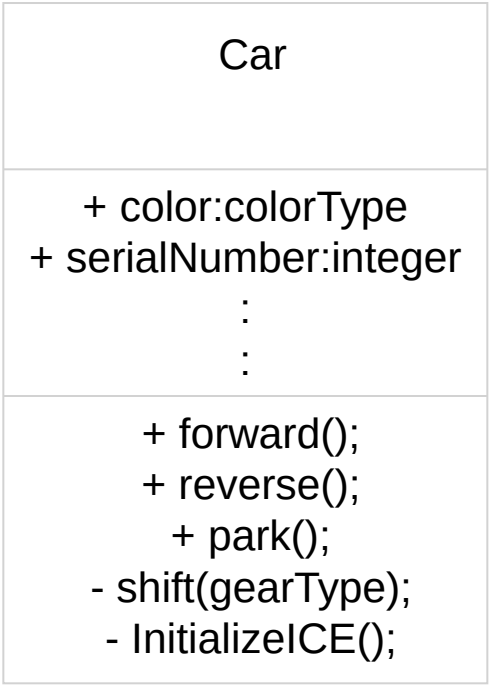
- The behavior of a class is represented by a set of operations.
- Operations represent the services that can be performed by an instance of a class.
- The format for operations is:
  - visibility name (parameters): returntype



Visibility of class features (attributes and operations)

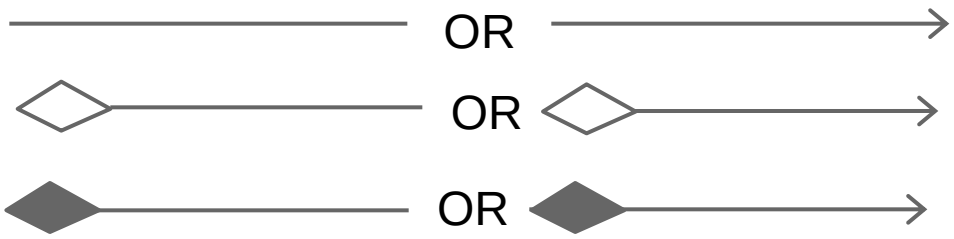
- defines if the attribute or operation can be accessed from outside the class where it is defined.
- **Public** (+)
  - This feature can be referenced from any place where its contained class is visible.
- **Protected** (#)
  - This feature can be referenced from any descendant (by specialization) of the class that defines the feature.
- **Private** (-)
  - Only the class that defines a private feature can use the feature.
- **Package** (~)
  - Only the class that defines the feature and its next enclosing scope can use the feature.

Visibility example

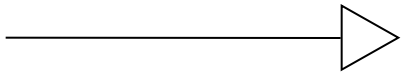


Class Relationships

- "The semantic connection between classes" ~ Grady Booch
- Class diagrams may contain the following relationships:
  - Association
    - Aggregation
    - Composition

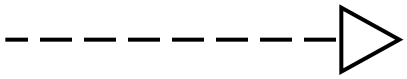


  - Generalization



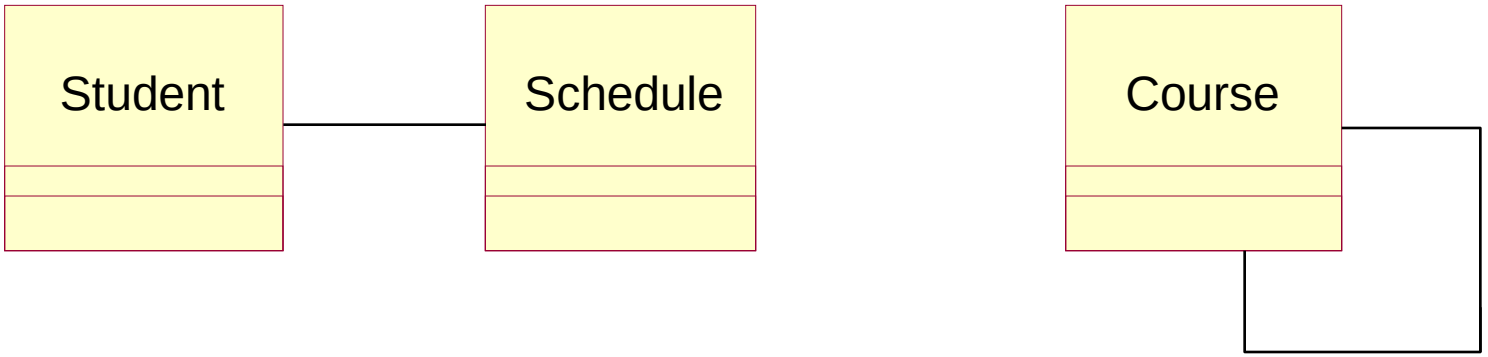
  - Dependency

  - Realization

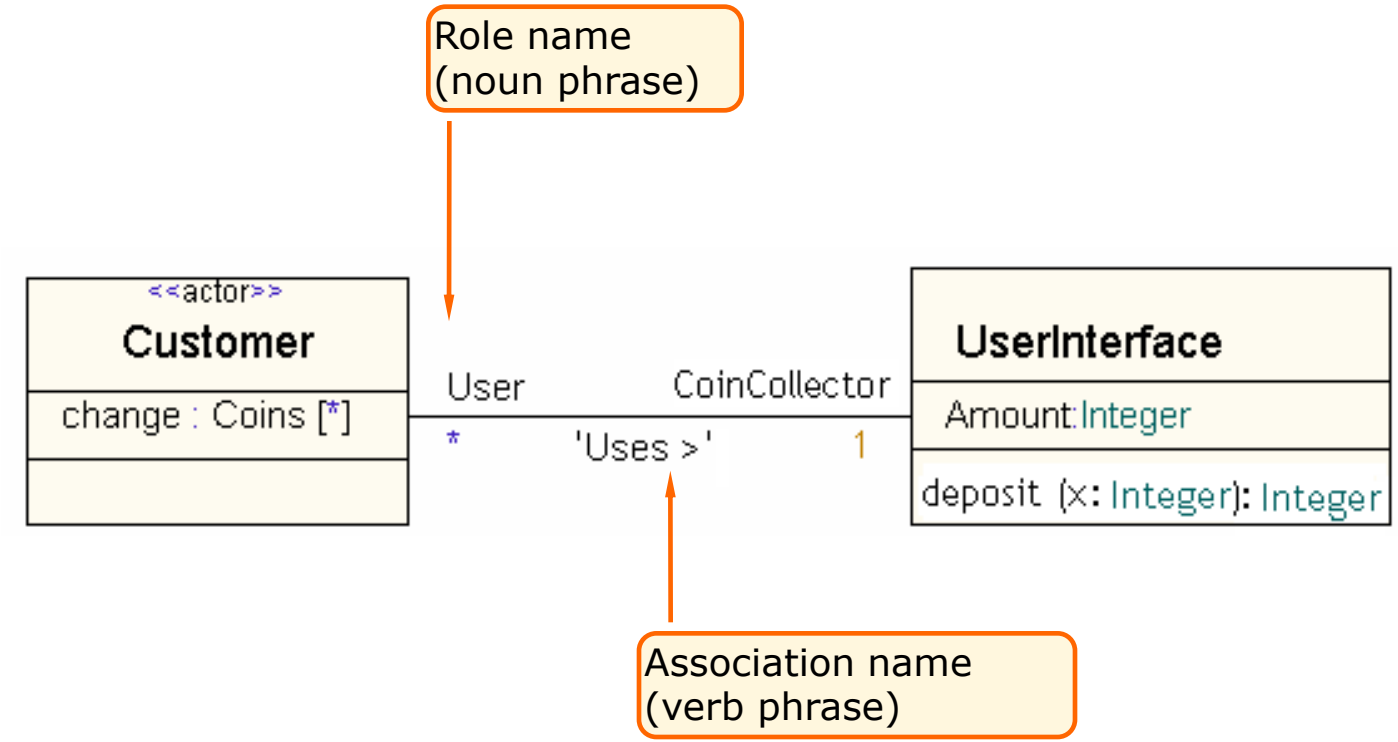


What Is an Association?

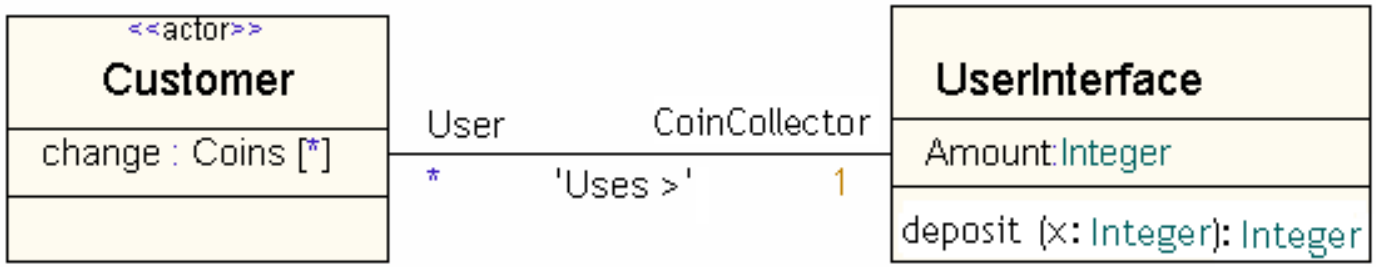
- The semantic relationship between two or more classifiers that specifies connections among their instances
  - A structural relationship, specifying that objects of one thing are connected to objects of another



Naming associations and roles



An association is a blueprint for a set of links



"a Customer uses a CoinCollector"

or

"a UserInterface is used by multiple Users"

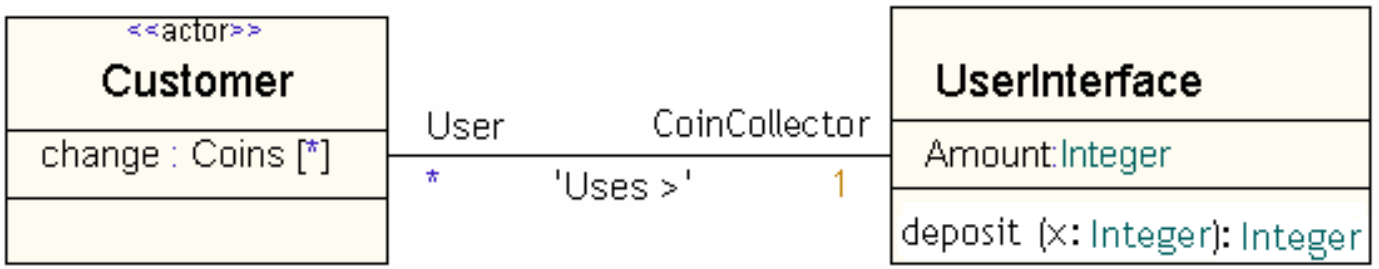
Associations are instantiated as links
(Conversely, a link is an instance of an association)

Multiplicity

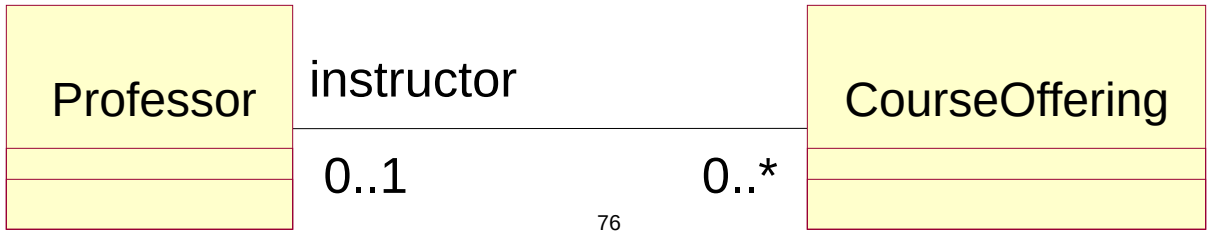
- Specifies how many objects at one role relate to a single object at the other role

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional scalar role)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

Reading multiplicity

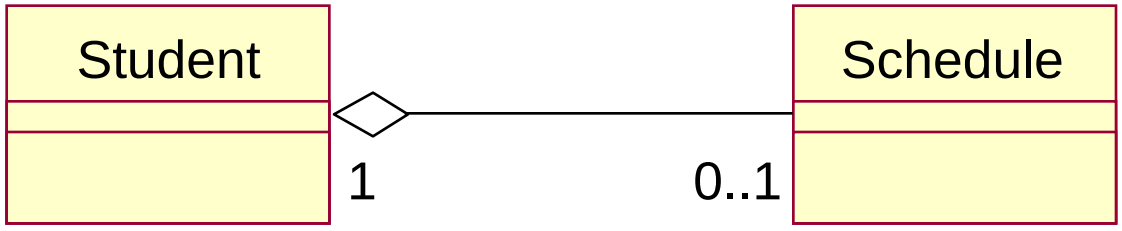


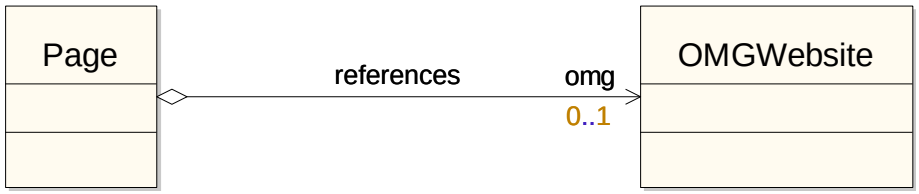
- Associations can be read in both directions
- Left-to-right
  - A single Customer object links to one UserInterface object
- Right-to-left
  - A single UserInterface object links to zero or more Customer objects



What Is Aggregation?

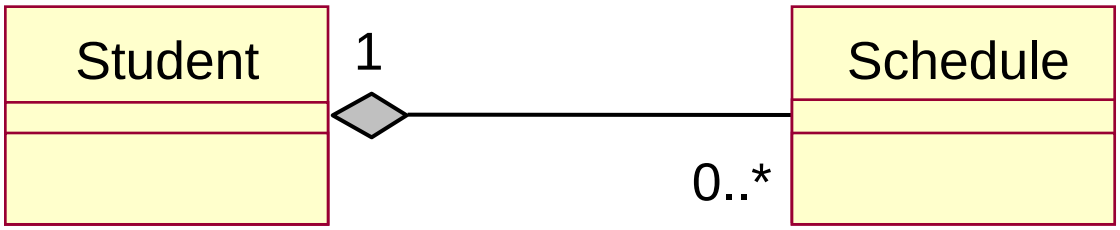
- An aggregation is a special form of association where the target of the relationship is a reference.
  - An aggregation is an “owns” relationship.
- Multiplicity is represented like other associations.



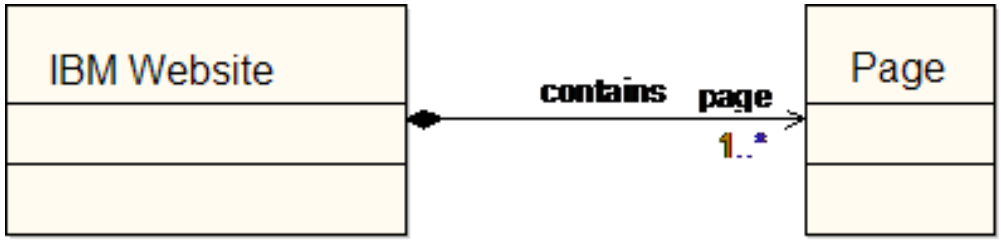


What Is Composition?

- A composition is a stronger form of association in which the composite has sole responsibility for managing its parts – such as their allocation and deallocation.
  - Is is a “is part of” relationship
- It is shown by a diamond filled adornment on the opposite end.



By definition, composition is a non-shared aggregation.



Class Diagram Modeling

Finding classes

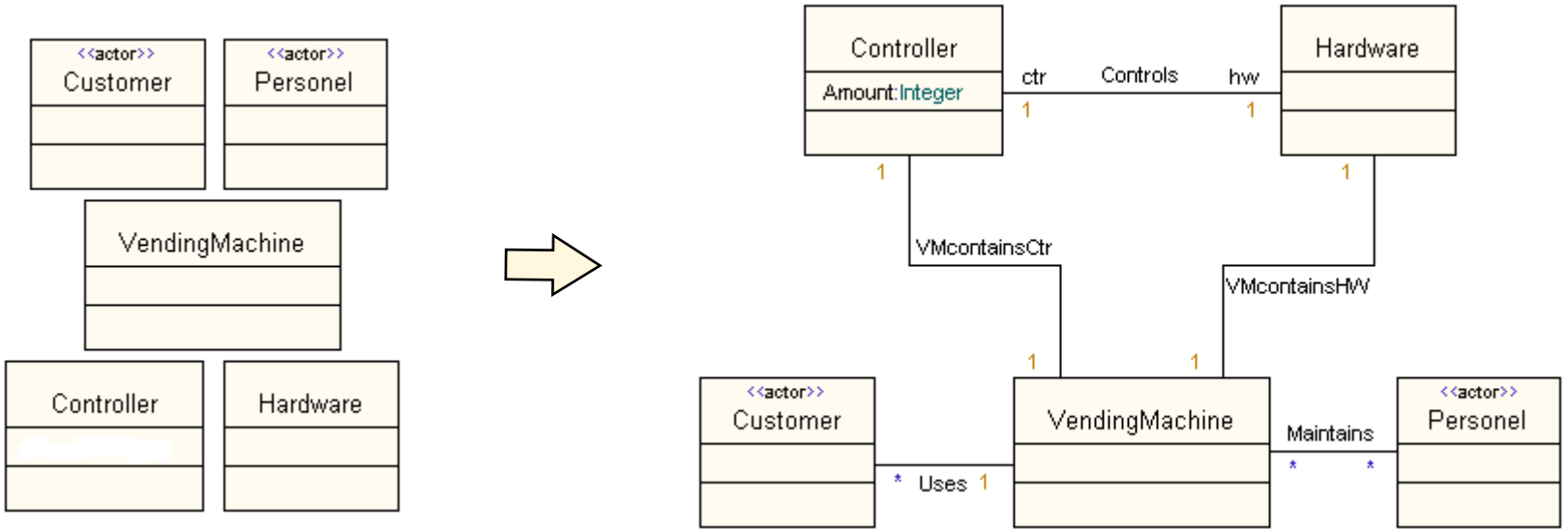
- Extract concepts (nouns) from requirements documents
  - Physical or tangible objects
  - Places
  - Transactions
  - Roles of people (for example, Users, ...)
  - Containers for other concepts
  - Other systems external to the system (for example, DB, ...)
  - Abstract nouns
  - Organizations
  - Event
  - Rules/Policies
  - Records/Logs

Finding attributes

- Attributes may be found by examining class definitions, the problem requirements, and by applying domain knowledge
- The name of an attribute should be a noun
- Try to limit the number of attribute types by reusing the already existing ones
- If an attribute needs be to shared by several classes, the attribute needs to be defined as its own class
- Don’t argue too much on what should be an attribute or a class. When doubts, make it a class

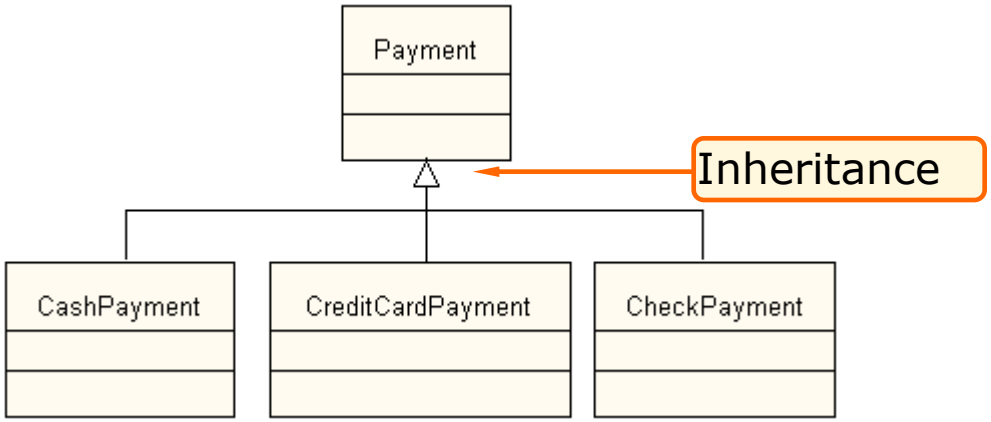
Finding associations

- "Fix" one concept class and consider the other ones in turn
- Are these two concepts related?
- If so, decide on the name of the association, the multiplicity…
- Do not just draw a line and leave the association unspecified
- Missing associations will be picked up during design



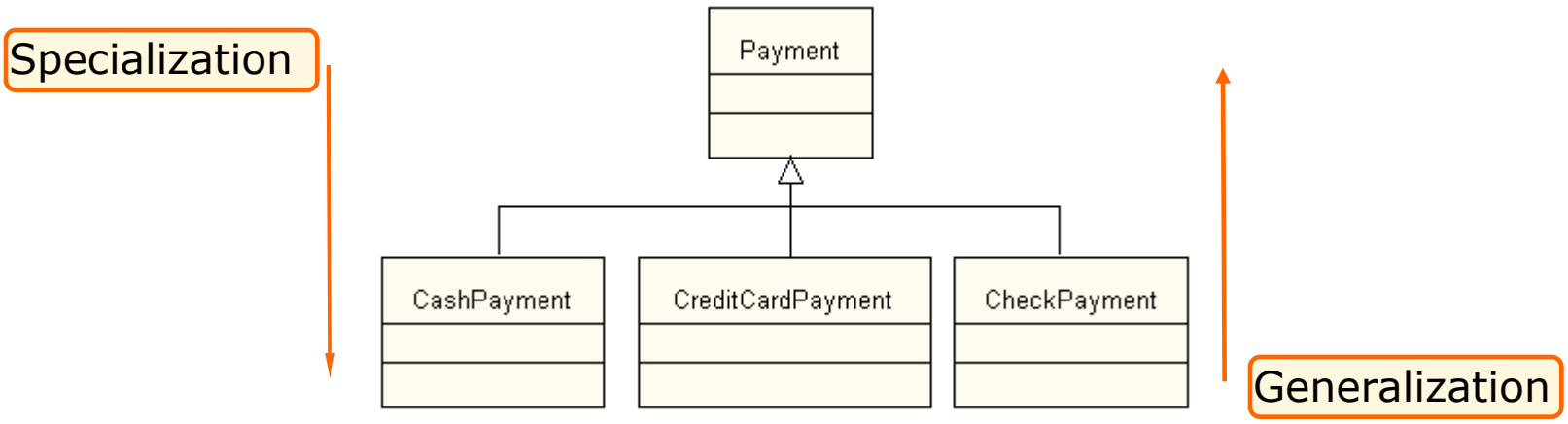
Inheritance

- Most important feature of Object-Oriented concept: Reusability
- Allow factoring common characteristics (attributes and operations) from different specific concepts into a general concept
  - Cash, Credit Card, Check payments require an amount, and a currency
- All attributes and operations of the general class are part of the inheriting class
- The hollow arrow points to the more general concept



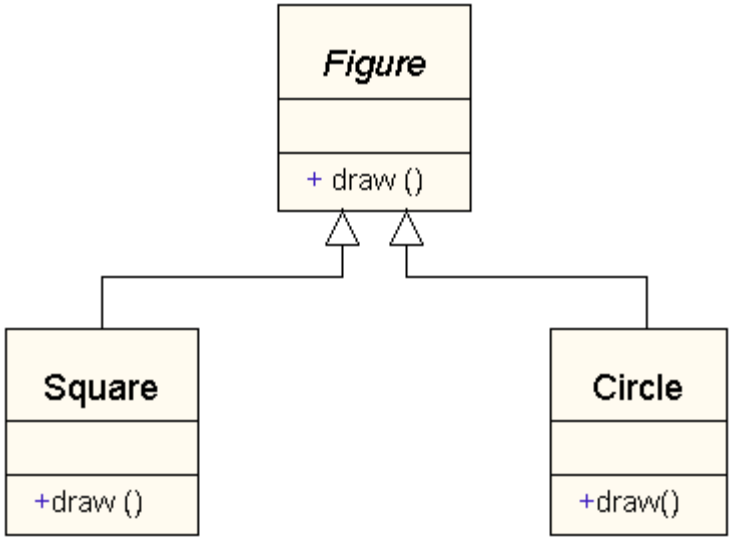
Generalization/specialization

- A generalization is the process of building a more general class from a set of specific classes
- A specialization is the process of creating specialized classes based on a more general class



Polymorphism

- A specialized element can add features and impose restrictions on existing ones, for example, attributes, operations, and multiplicity
- Two specialized concepts could inherit from the same general class, and implement the inherited operations differently



OO Principle:
Encapsulation

Inheritance rules

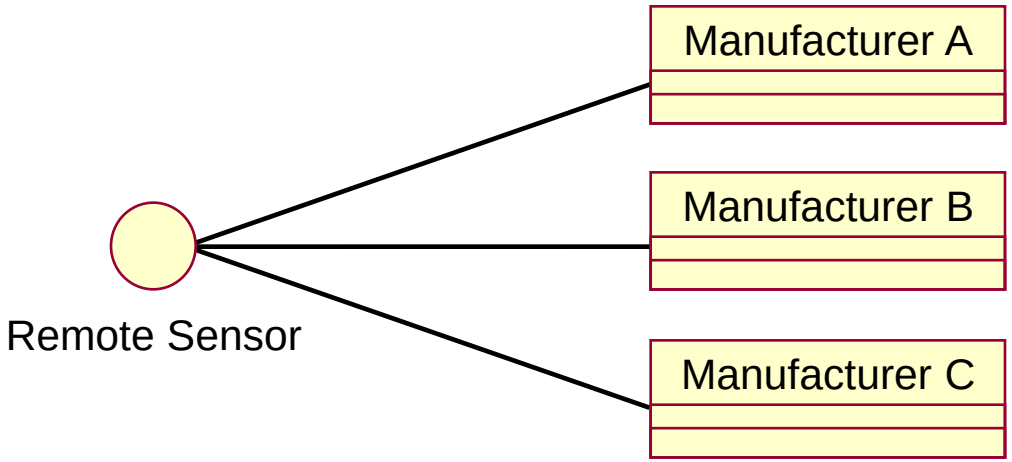
- Inheritance should only be used as a generalization mechanism
- The 100% Rule
  - All attributes and operations of the base class are applicable to the specialized class
- The ‘is-a-kind-of’ or ‘is-a’ Rule
  - The statement “<derived class> is a <base class>” should be true
  - Every instance of the <derived class> can be viewed as an instance of the <base class>

What Is an Interface?

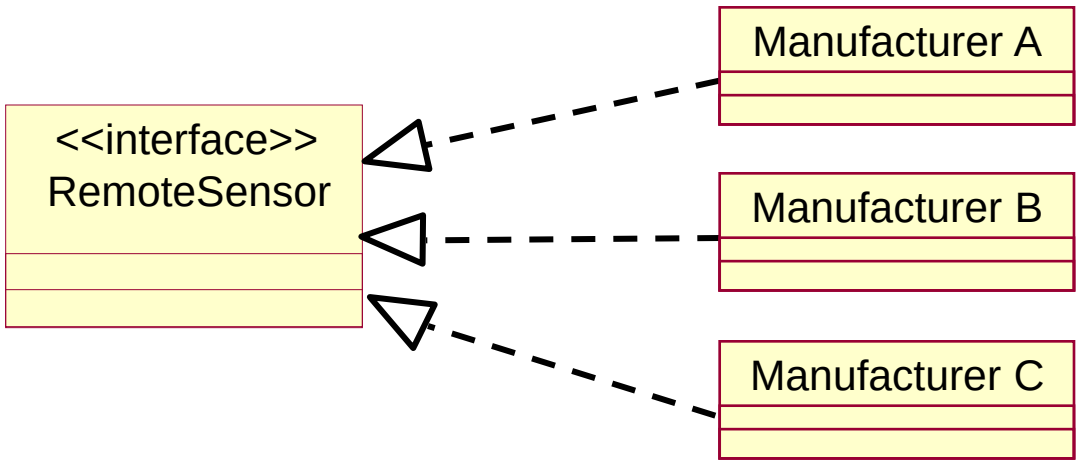
- A declaration of a coherent set of public features and obligations.
  - A contract between providers and consumers of services. Examples of interfaces are:
    - Provided interface - The interfaces that the element exposes to its environment.
    - Required interface - The interfaces that the element requires from other elements in its environment in order to be able to offer its full set of provided functionality.

A Provided Interface

Elided/Iconic Representation (“ball” or ”lollypop”)

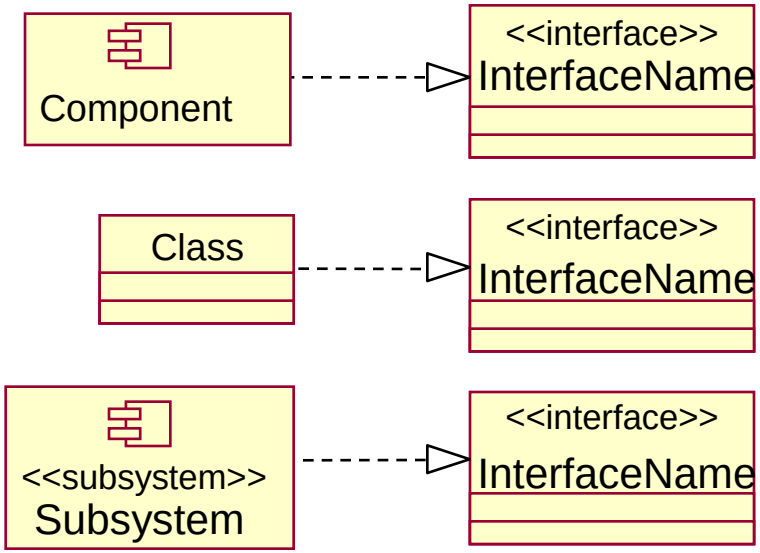


Canonical (Class/Stereotype) Representation

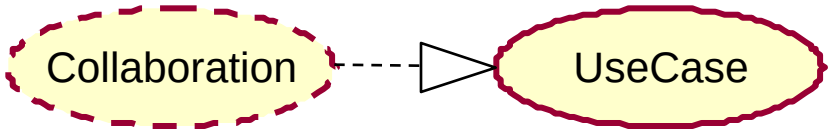


Provided interface and the **realization** relationship

- One classifier serves as the contract that the other classifier agrees to carry out, found between:
  - Interfaces and the classifiers that realize them

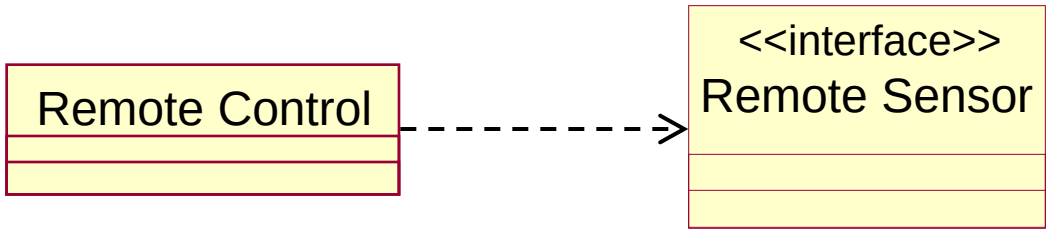


  - Use cases and the collaborations that realize them



A Required Interface

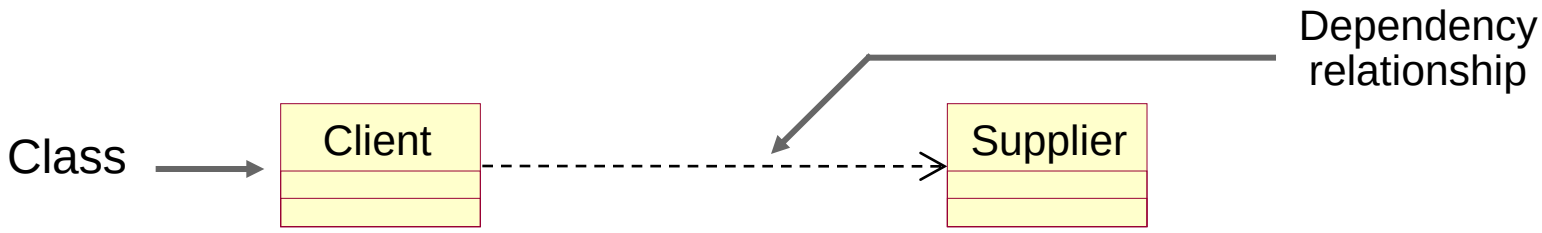
Elided/Iconic Representation (“socket”)



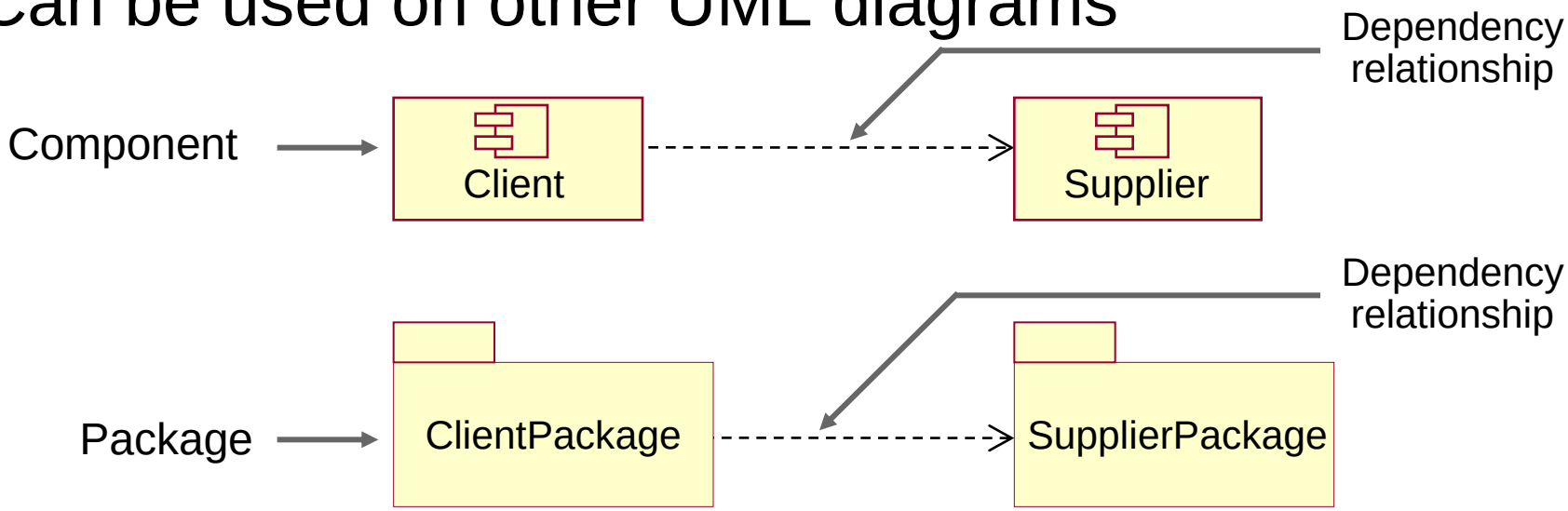
Canonical (Class/Stereotype) Representation

Required interface and **dependency** relationship

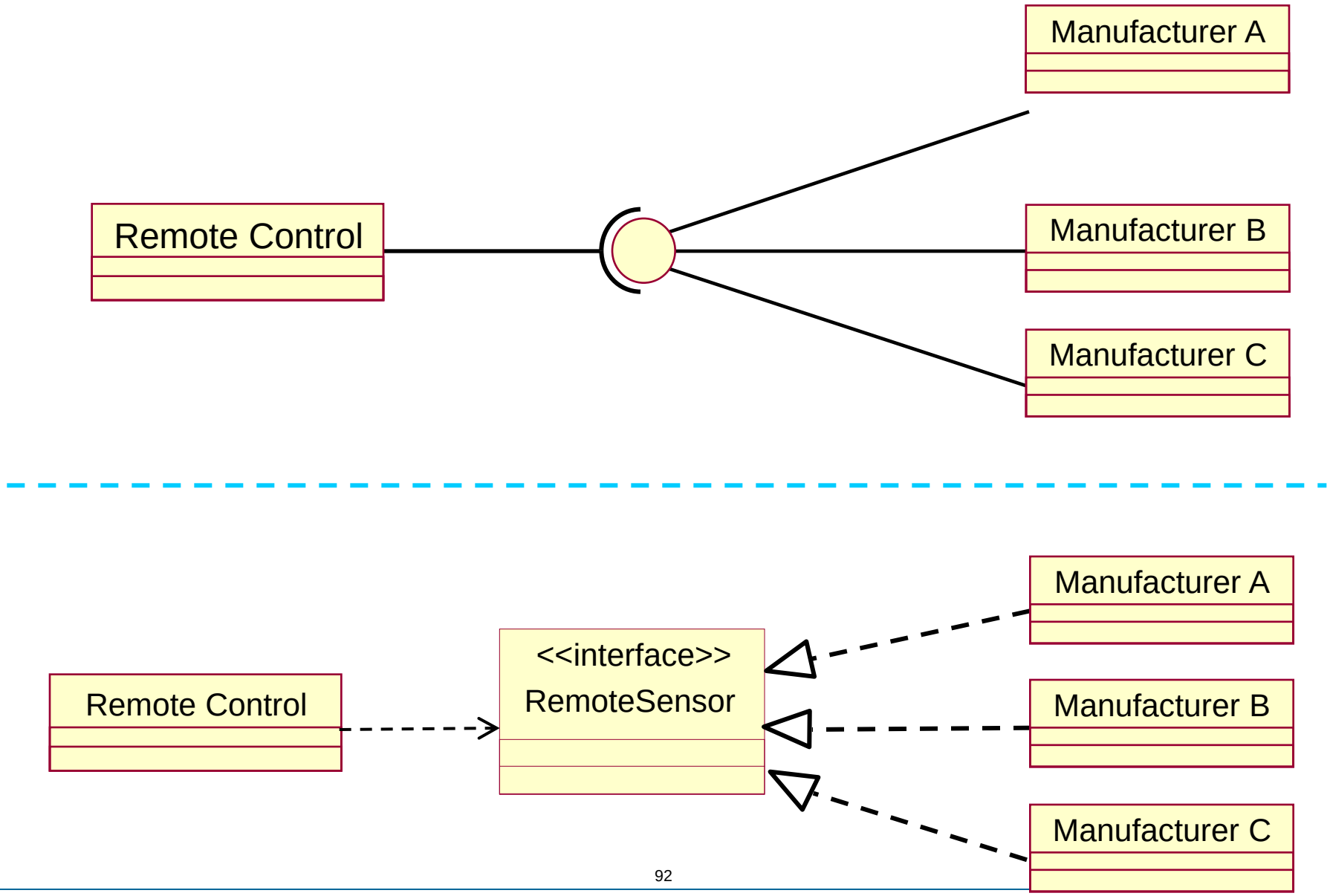
- A relationship between two model elements where a change in one may cause a change in the other
- Non-structural, “using” relationship



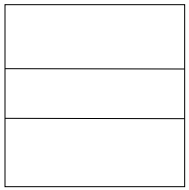
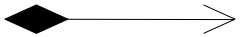
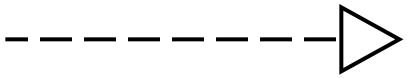
- Can be used on other UML diagrams



Connecting Interfaces



Summary symbols (Class)

- Class 
- Association
- Aggregation 
- Composition 
- Generalization 
- Dependency 
- Realization 

Interaction Modeling

Interaction model

- A view of a system that emphasizes the behavior of the system as a whole (as it appears to outside users)
- Uses:
  - Sequence Diagrams
  - Activity Diagrams
  - State machine Diagrams
- The system is regarded as a black box and the functionalities are expressed from a user's perspective
- The model must capture the requirements of the system, not the implementation solution

High Level Behavior - Sequence Diagram

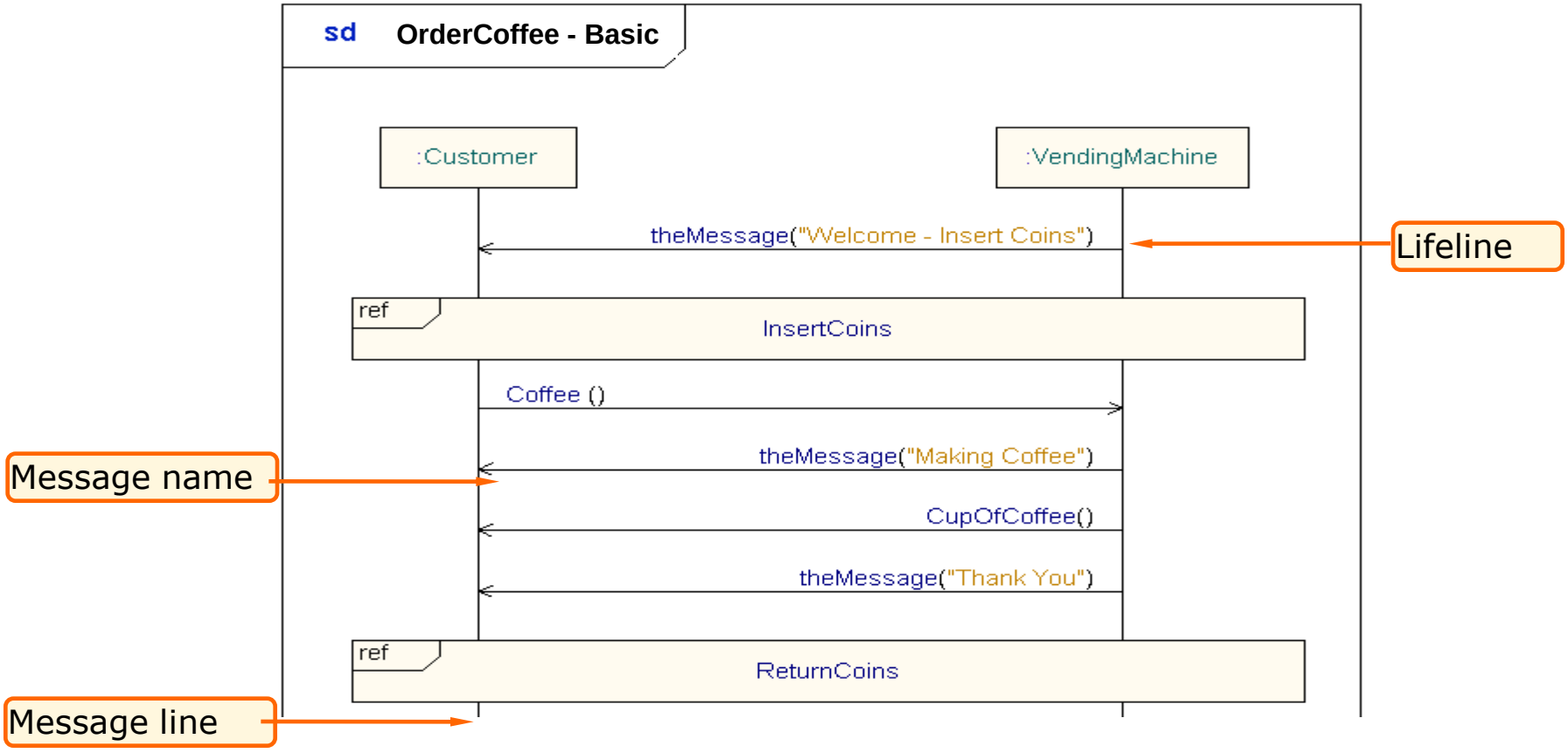
Sequence diagrams

- Sequence diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task.
- Sequence diagrams show sequences of messages (“interactions”) between instances in the system
- Each diagram depicts a possible set of messages only
  (a ”scenario”) - they do not specify all possible messages
- Sequence diagrams are read left to right and descending
- Sequence diagrams emphasize time ordering

System level sequence diagrams

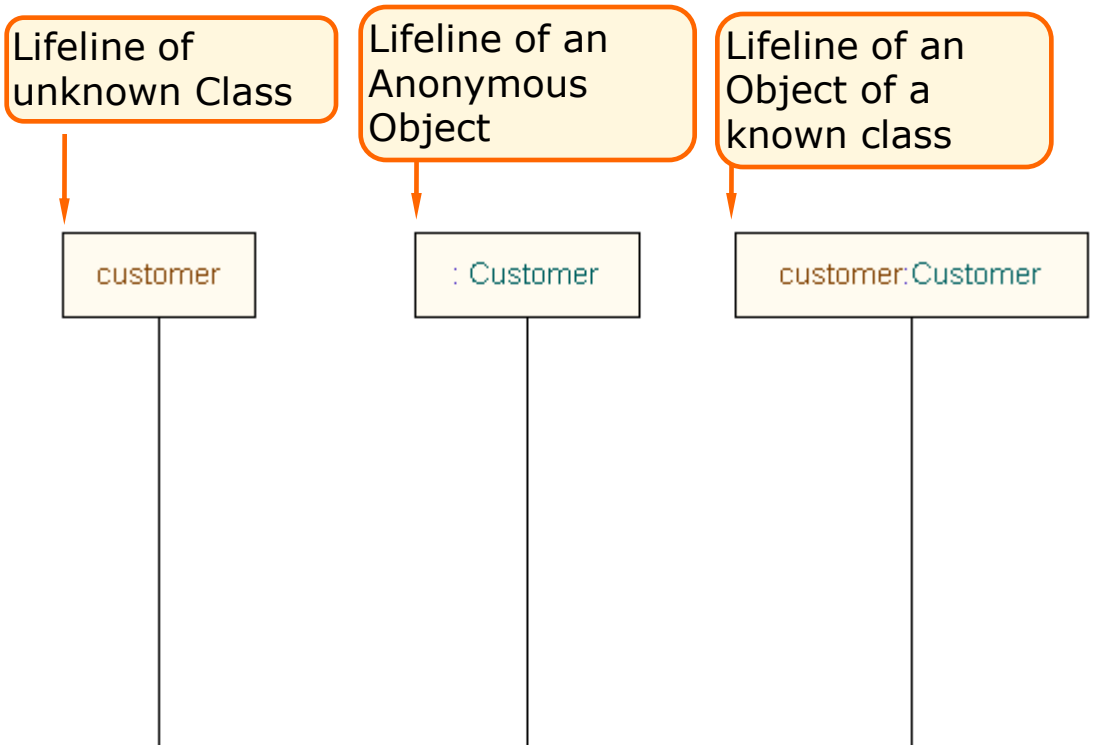
- They describe the interaction between the Actors and the System
  - Remember: Black box
- They define the System Interface
- They can be used as System Level Test Cases

Sequence diagram – Elements



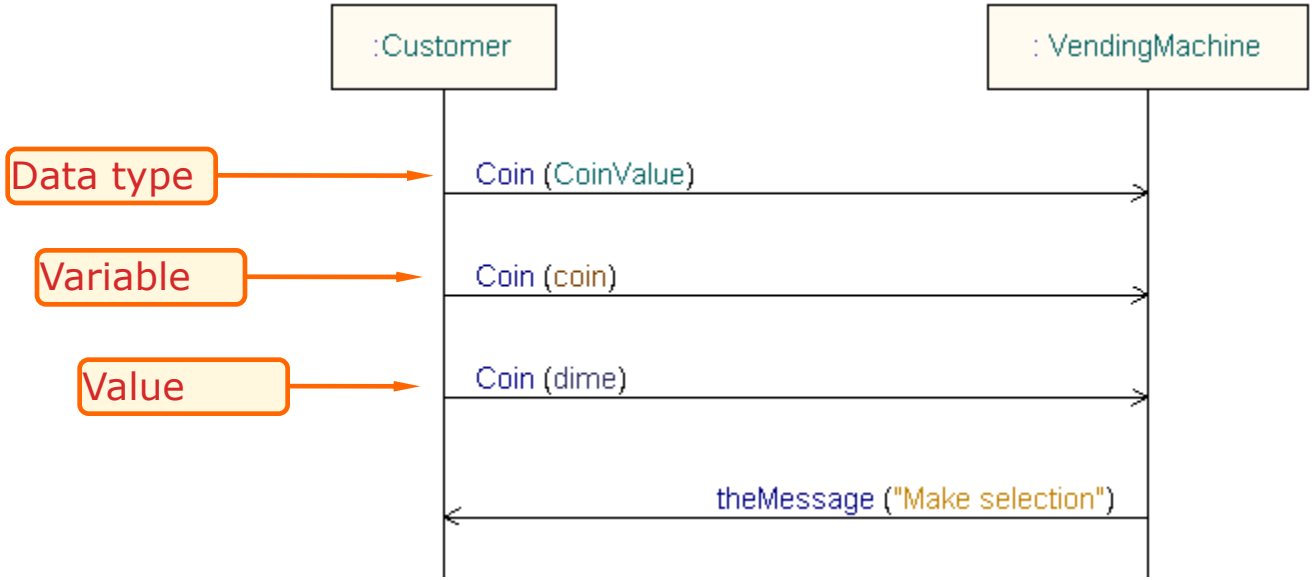
Lifelines

- A Lifeline consists of a rectangle that identifies the connectable element (such as a part or component) and a vertical line indicating its time of existence



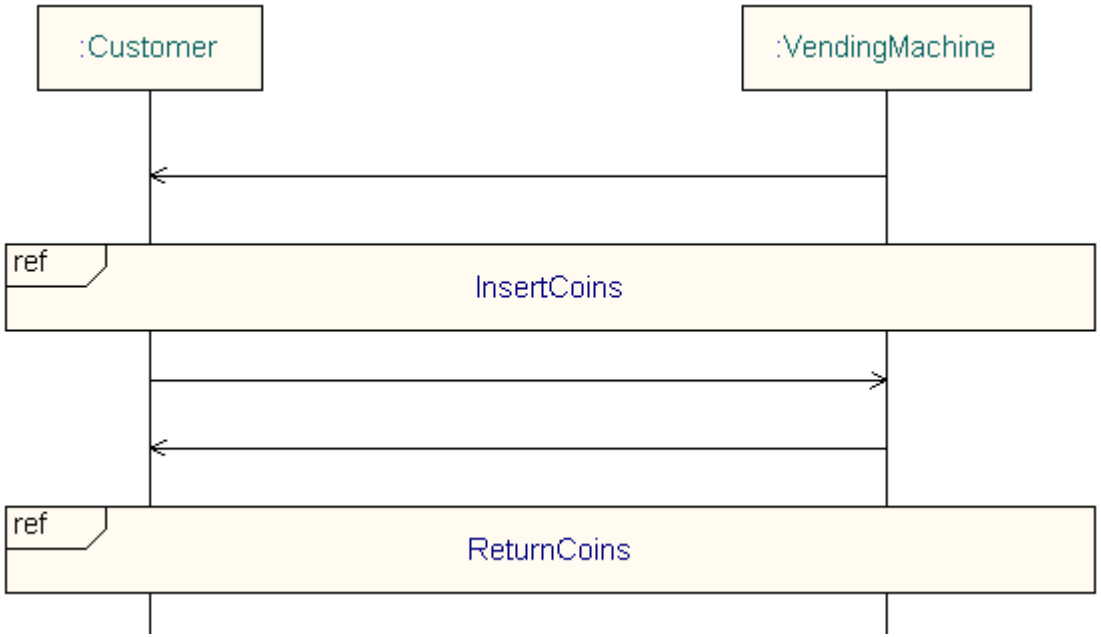
Object communication

- Messages are labeled with the name of the message (operation or signal) and its argument values.
- Message parameters can be a data type, a variable or a constant.



Referencing

- To avoid unnecessary duplication it is possible to reuse already existing Sequence Diagrams

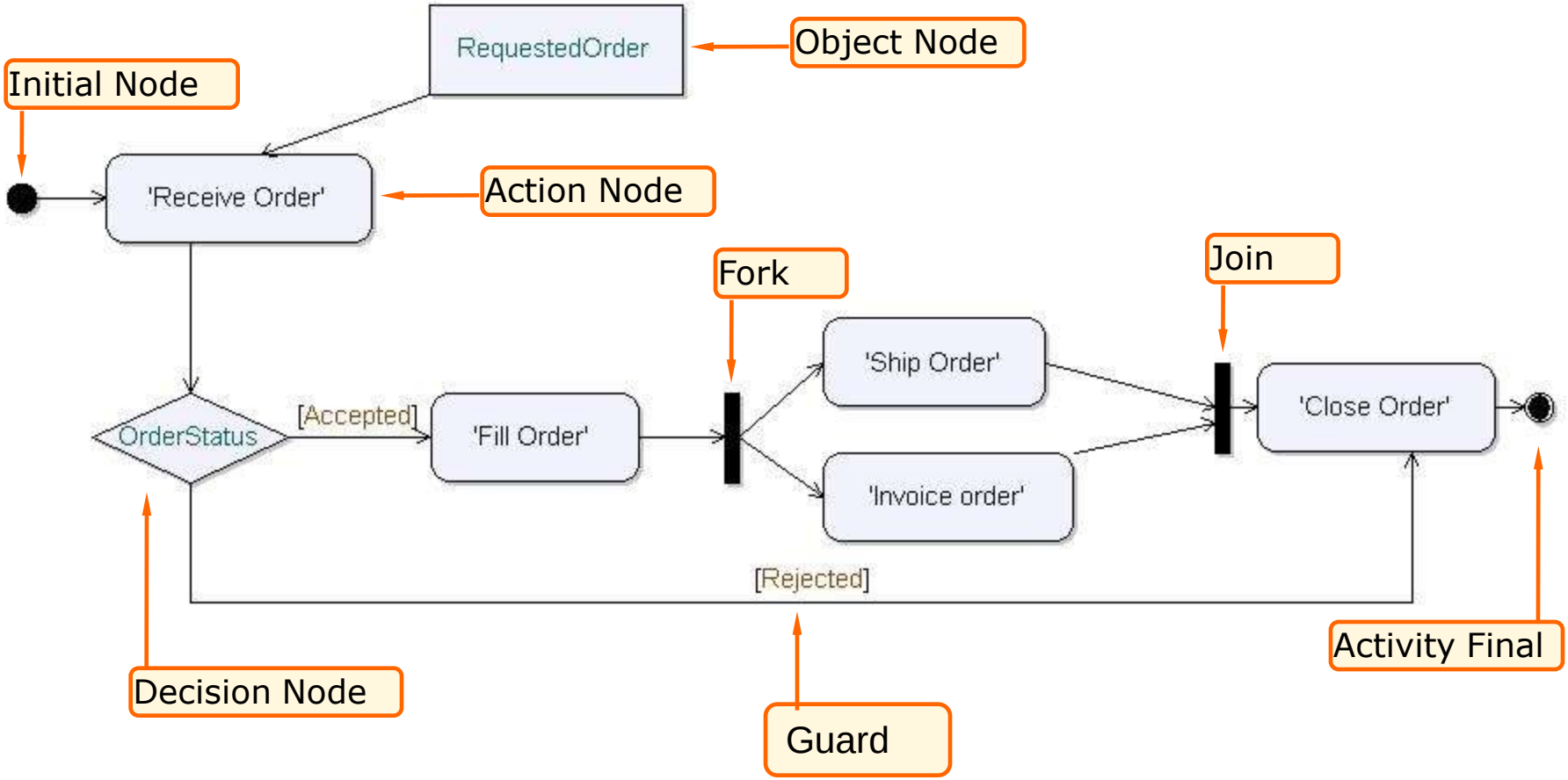


High Level Behavior - Activity Diagram

Activity diagrams

- Activity diagrams describe the workflow behavior of a system
- Activity diagrams can show activities that are conditional or parallel.
- Activity Diagrams are useful for:
  - analyzing a use case by describing what actions need to take place and when they should occur
  - describing a complicated sequential algorithm
  - modeling applications with parallel processes
  - modeling business workflow

Activity diagram - elements



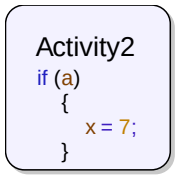
Activity diagram symbols - 1

Initial node

Starting point for invocing other activities. An activity may have several starting points.

Activity final symbol

Aborts all flows in the containing activity.



Action/Activity

An action is an executable unit. Can also refer to a new activity diagram.

Action states and activity states

- Each state in an activity diagrams is either:
  - an action state
    - describes an atomic action
    - the outgoing transitions are implicitly triggered by the completion of the action in the state
  - or an activity state
    - describes an enduring activity which can be interrupted
    - invokes an activity graph
    - the activity state is not exited until the final state of the nested graph is reached.

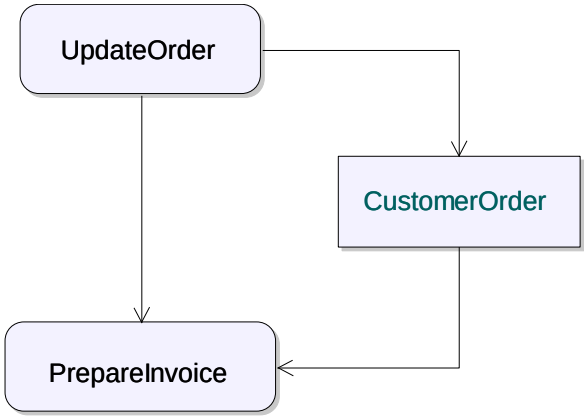
Activity diagram symbols - 2

Object node

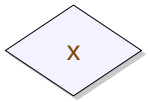
Represents an instance of a clasifier, usually a class

Used to show input to or output from an action.

An **object flow** shows objects being generated or used by actions or activities in activity diagrams



Activity diagram symbols - 3

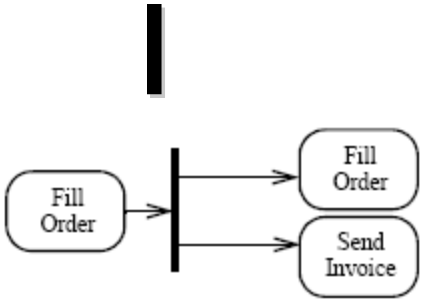


Decision node

A decision node is a control node that chooses between outgoing flows.

Each branch has its own guard condition

“Else” may be defined for at most one outgoing transition



Fork/Join symbol

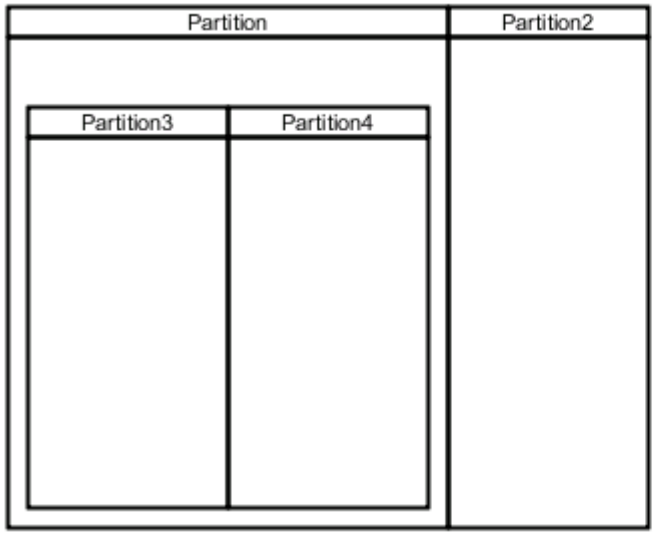
Divides a flow into multiple concurrent flows. Flows can be split and synchronized again.

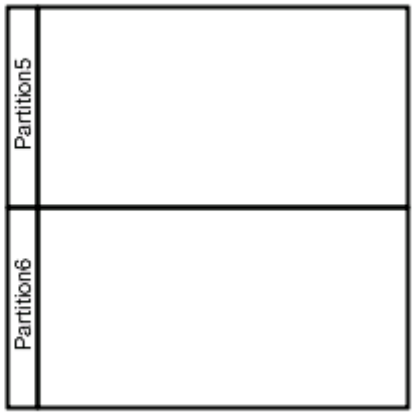
Activity diagram symbols - 4

Activity Partition (Swimlanes)

Swimlanes visually group the action states. They have no semantics but are often used to show a relation between activities, for example activities performed by a person, business unit etc.

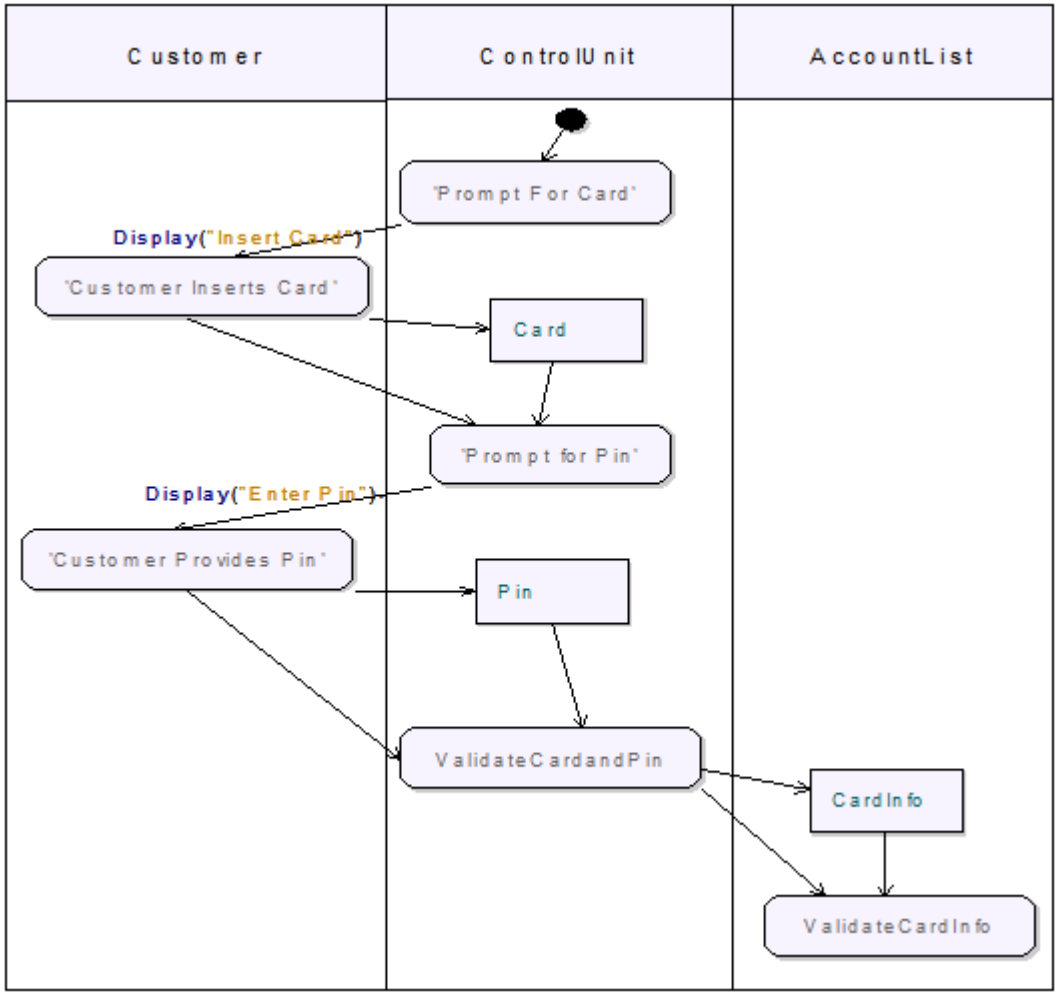
Swimlanes can also contain other swimlanes.





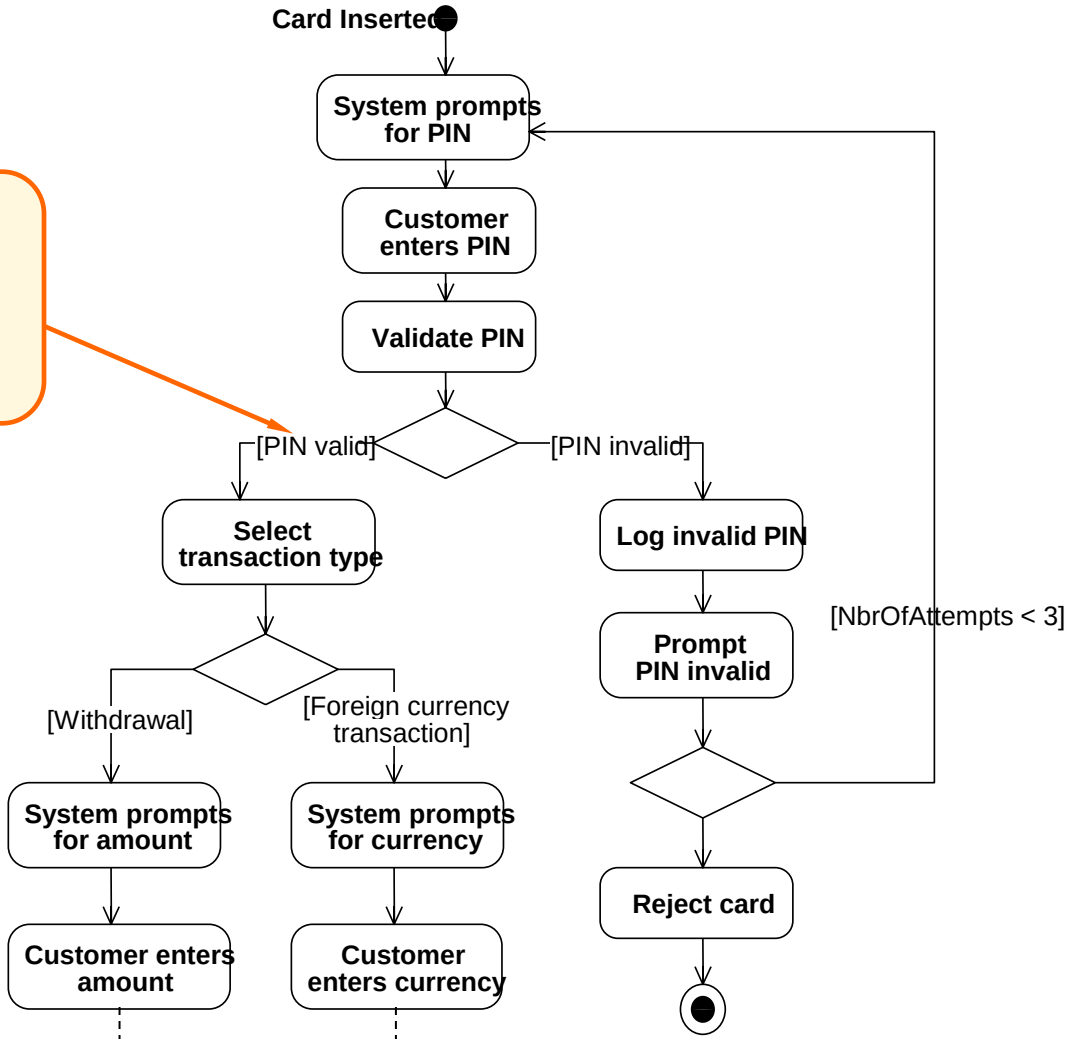
Activity diagram – Swimlane example

- Transitions can take place from one swim lane to another



Activity diagram – Example

Guard: Follow this path, only if the text in the brackets **[ ]** evaluates to true.



High Level Behavior - State Machine Diagram

State machine diagram usage

- To elaborate use case specifications
- To specify high-level system behavior during analysis
- Capture significant events that can act on an object

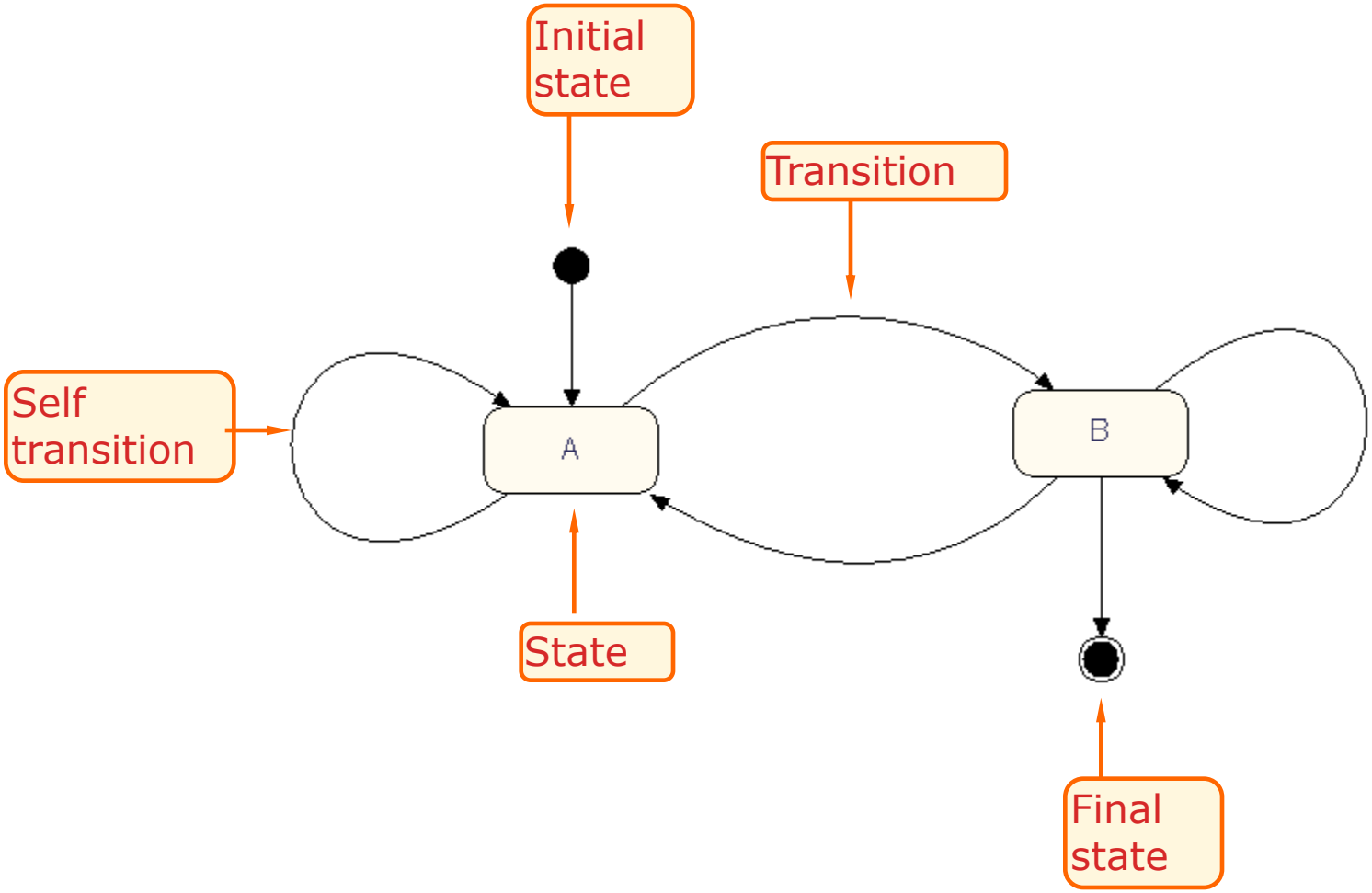
System level state machine diagram

- Defining a System Level State Machine that includes all Use Cases would be unrealistic
  - Too many system states
  - Too many possible transitions
- Instead, a System Level State Machine should be created to show only some aspects of the functionality
- Each of these State Machines could be used to do partial simulation of the System
  - On paper
  - Using a UML tool

State machine diagram

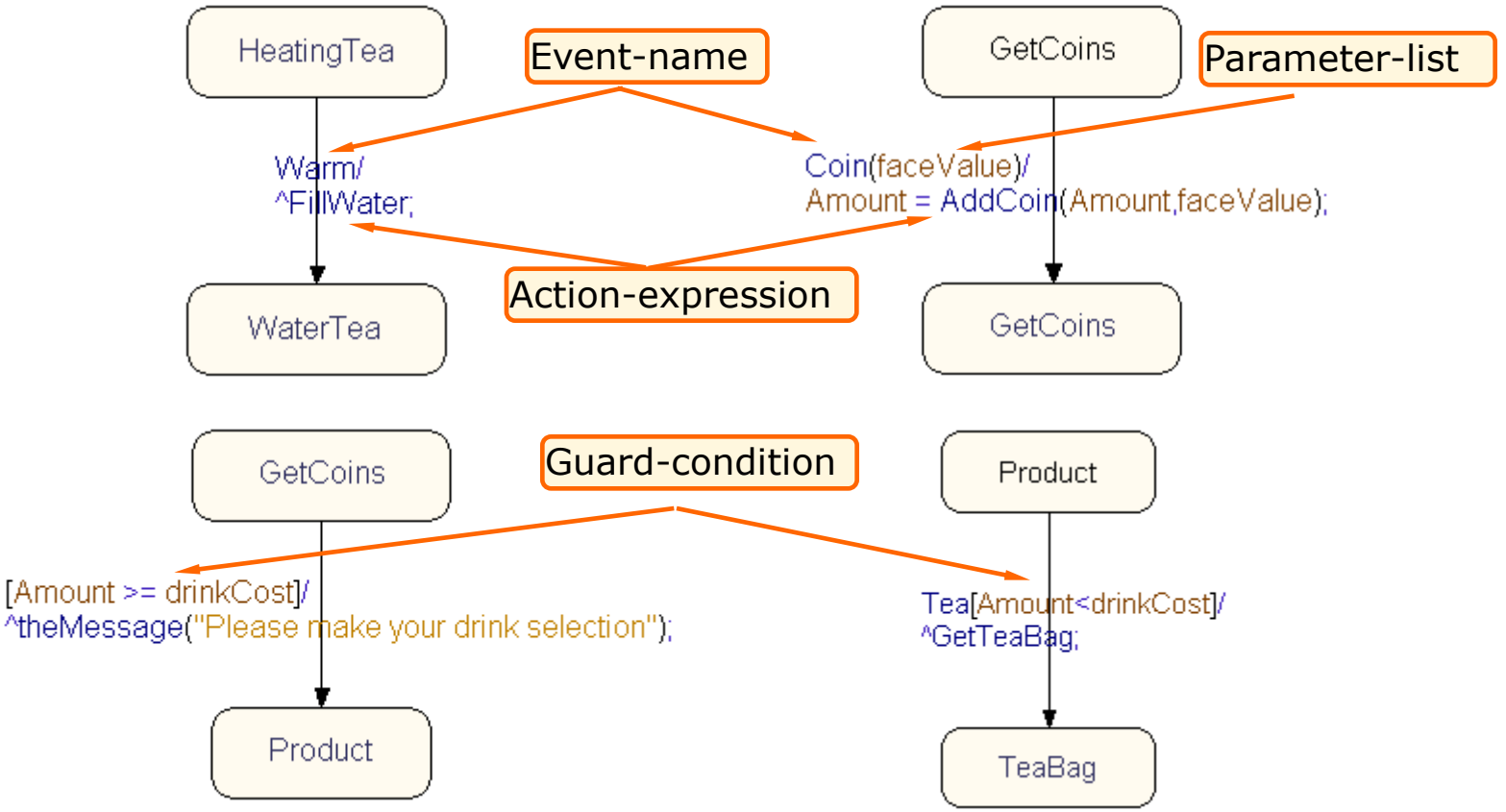
- A state machine diagram specifies the **dynamic behavior** of an element in a reactive, event-driven way
- Specifies a finite number of **states** of an element and how **transitions** between states are performed in response to **events**

State machine diagram - Elements



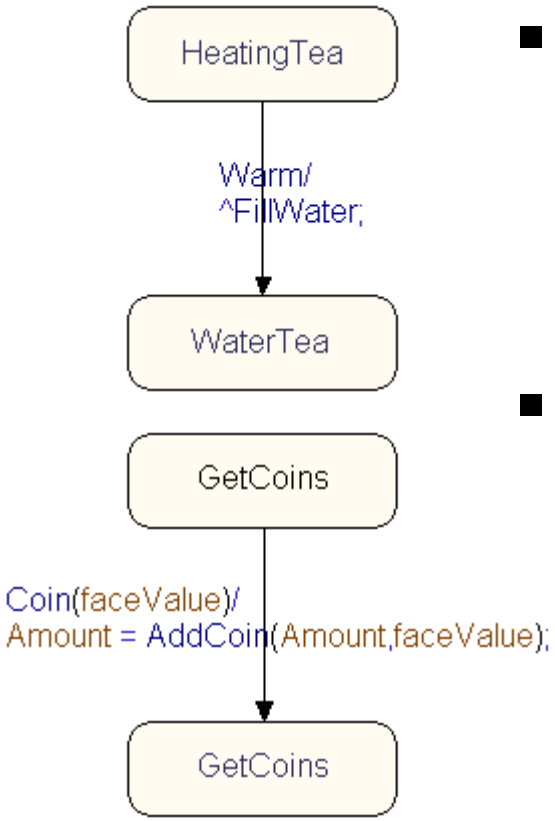
Transition (1)

event-name (*parameter-list*) [*guard-condition*] / *action-expression*;



Transition (2)

event-name (*parameter-list*) [*guard-condition*] / *action-expression*;

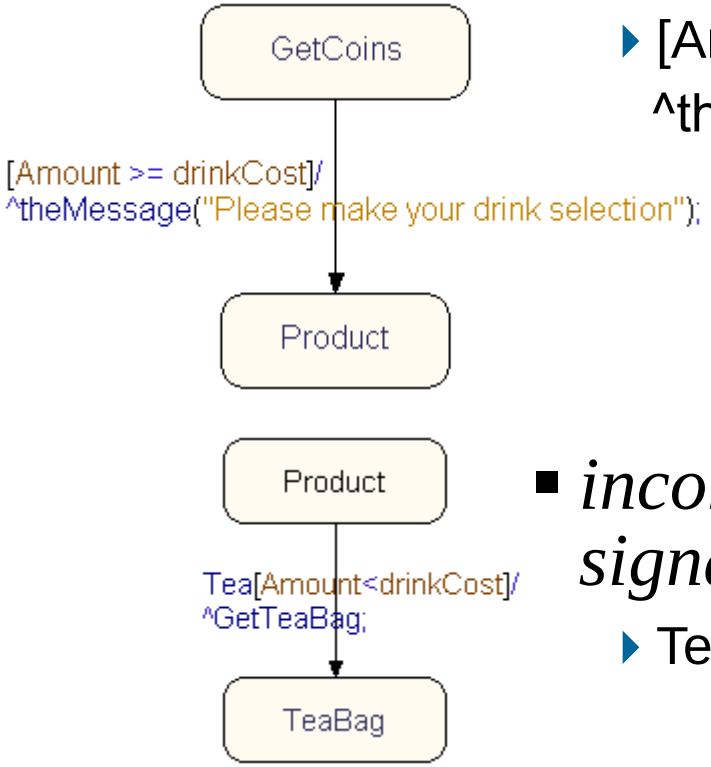


- *incoming-signal-name / signal-output*;
  - Warm / ^FillWater;
- *incoming-signal-name* (*parameter-list*) / *operation-call*;
  - Coin(faceValue) / Amount = AddCoin (Amount , faceValue);
  - The variable that will contain the parameter value, must be declared and visible within the scope of the state machine

Transition (3)

event-name (*parameter-list*) [*guard-condition*] / *action-expression*;

- [*guard-condition*] / *signal-output* (*parameters-list*);
  - [Amount >= drinkCost] / ^theMessage(“Please make your drink selection”);



- *incoming-signal-name* [*guard-condition*] / *signal-output*;
  - Tea [Amount < drinkCost] / ^GetTeaBag;

Links

https://www.uml-diagrams.org/

https://www.omg.org/spec/UML/2.5/

http://agilemodeling.com/