Integrarea sistemelor informatice

Suport curs nr. 1/p

Programator >> Arhitect

Modelare UML

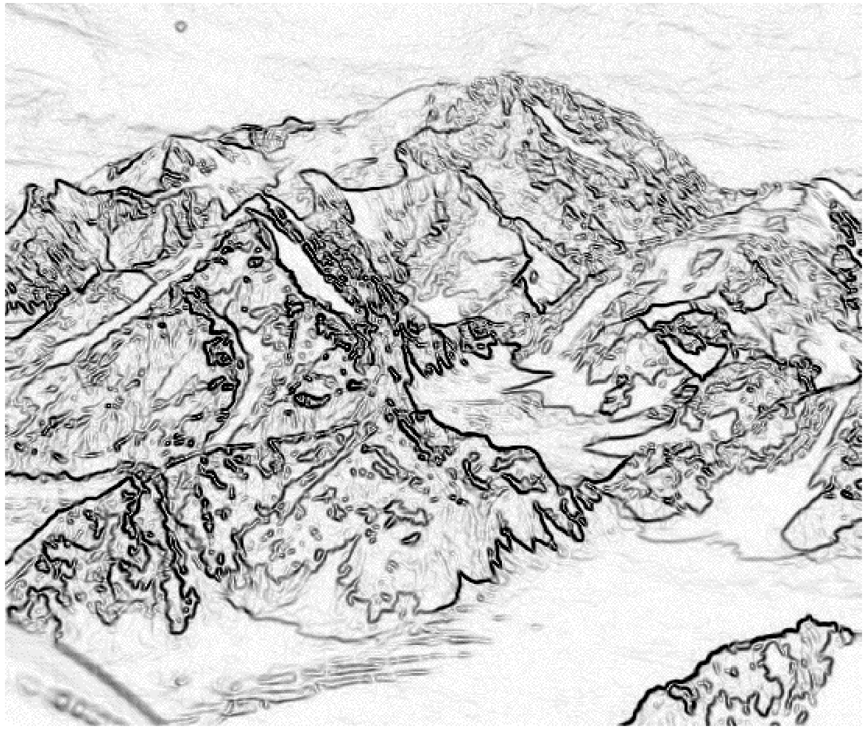
2024-2025

C1/p – Modelare UML

Obiective

- Introducere/recapitulare UML
- Identificarea diagramelor UML utile în modelarea sistemelor

Modeling with UML



Reference: Bernd Bruegge, Allen H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, Third Edition, Pearson, ISBN: 0-13-606125-7

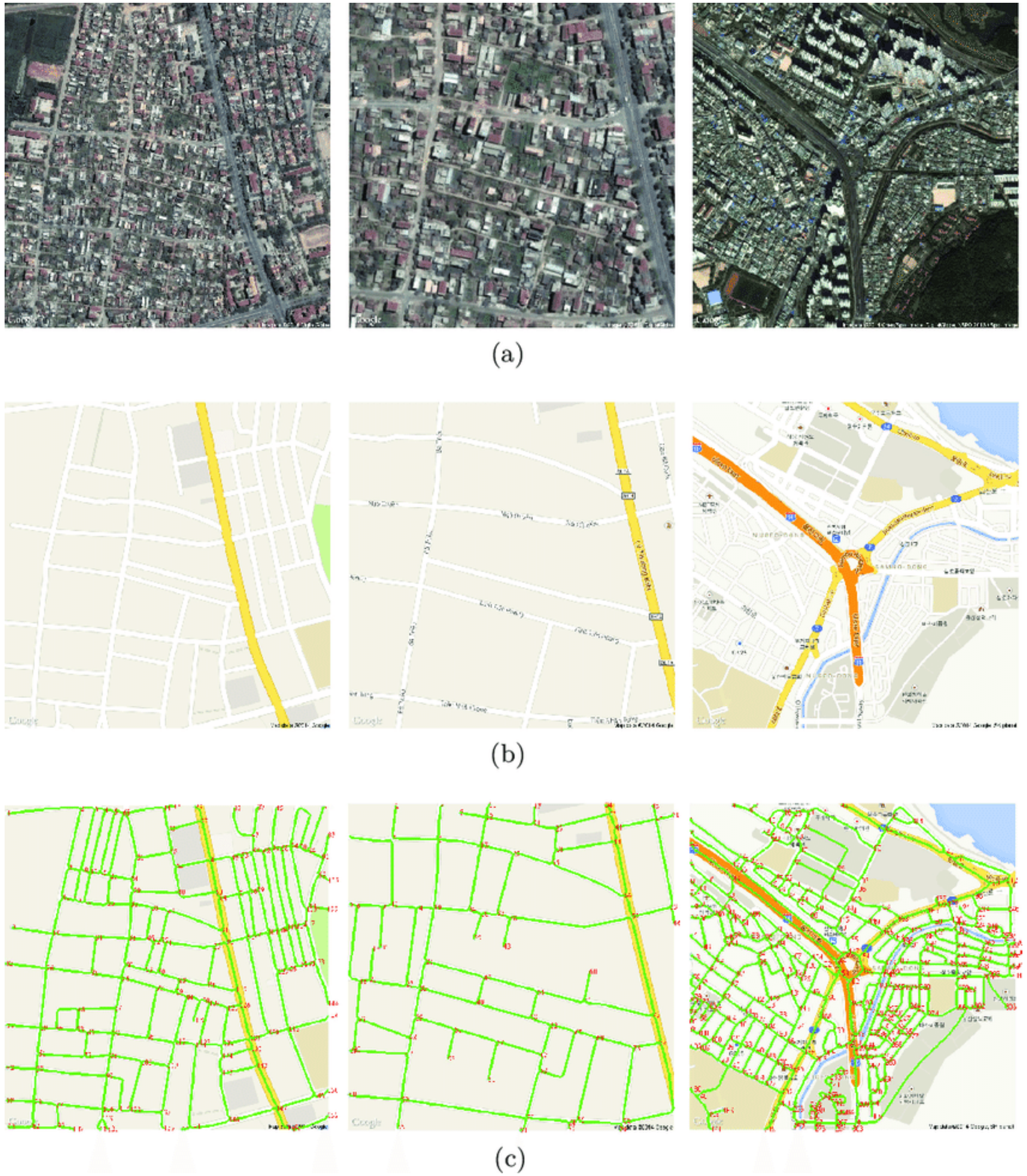
Overview: modeling with UML

- What is modeling?
- What is UML?
- UML diagrams
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - State (machine) diagrams
  - Activity diagrams (workflow / flowchart)

What is modeling?

- Modeling consists of building an abstraction of reality.
- Abstractions are simplifications because:
    - They ignore irrelevant details and
    - They only represent the relevant details.
- What is *relevant* or *irrelevant* depends on the purpose of the model.

Example: street map



Case Study

Path Planning for Unmanned Vehicle Motion Based on Road Detection Using Online Road Map and Satellite Image
http://dx.doi.org/10.1007/978-3-319-16631-5_32

Typical road detection: (a) Satellite images, (b) road map images, (c) road network estimation results with many additional detected road segments by our proposed method.

Why model software?

- Software is getting increasingly more complex
  - Windows 10 > 50 mil lines of code
  - A single programmer cannot manage this amount of code in its entirety.
- Code is not easily understandable by developers who did not write it
- Modeling is a mean for dealing with complexity
  - We need simpler representations for complex systems

Image by svstudioart on Freepik

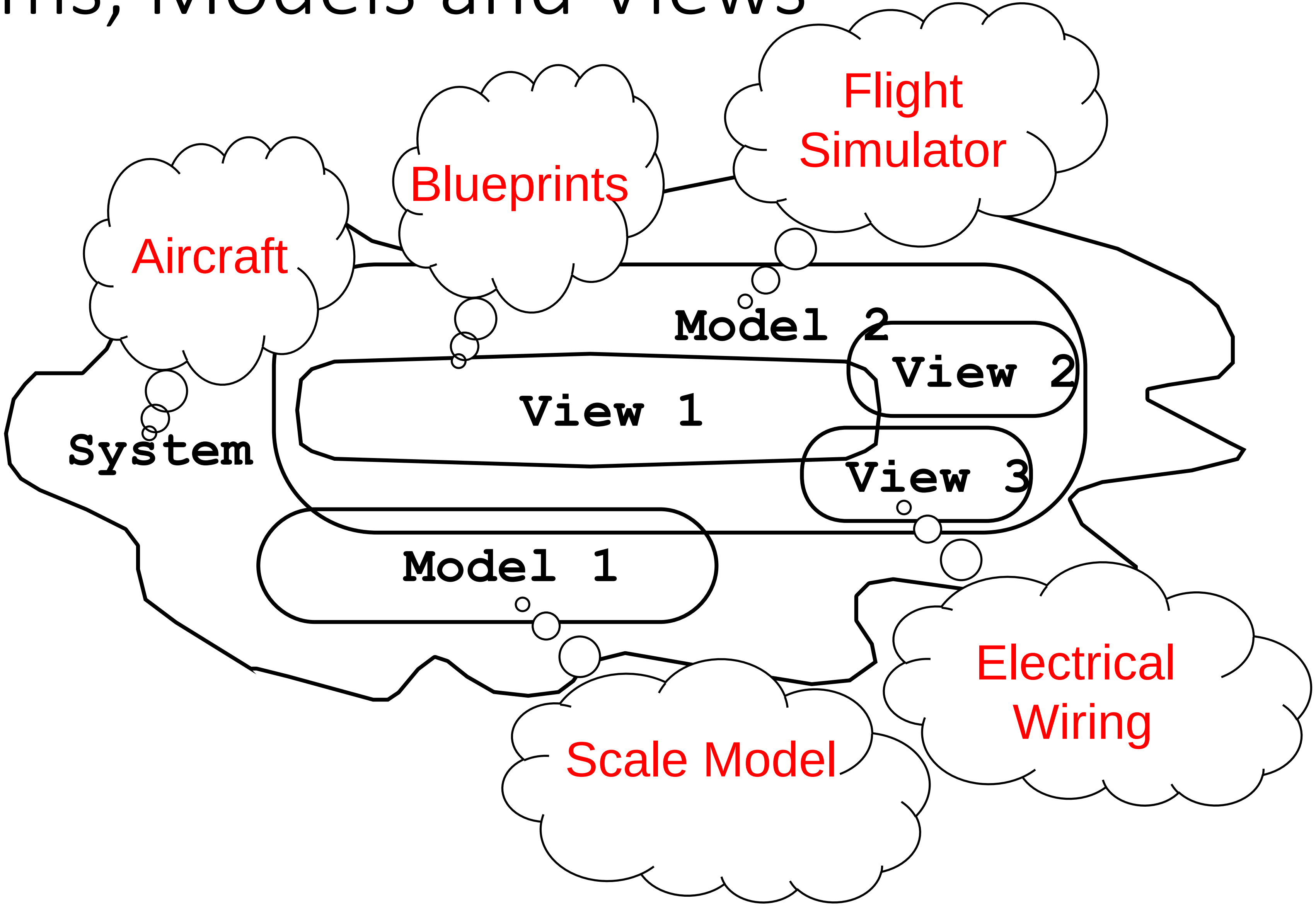
Systems, Models and Views

- A ***model*** is an abstraction describing a **subset** of a system
- A ***view*** depicts **selected aspects** of a model
- A ***notation*** is a set of graphical or textual **rules** for depicting views

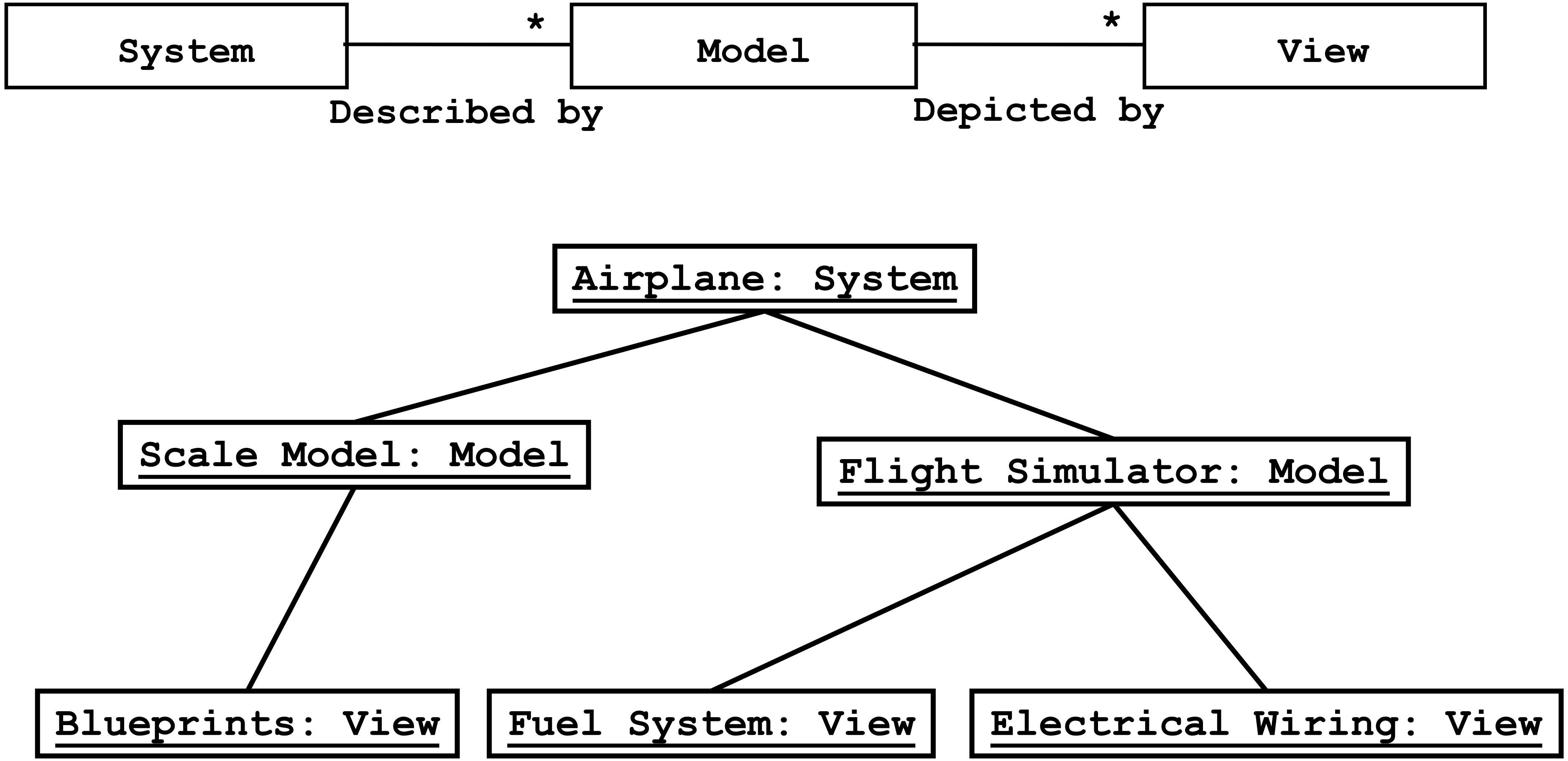
Examples:

- System: Aircraft
- Models: Flight simulator, scale model
- Views: blueprints, electrical wiring, fuel system

Systems, Models and Views



Systems, Models and Views



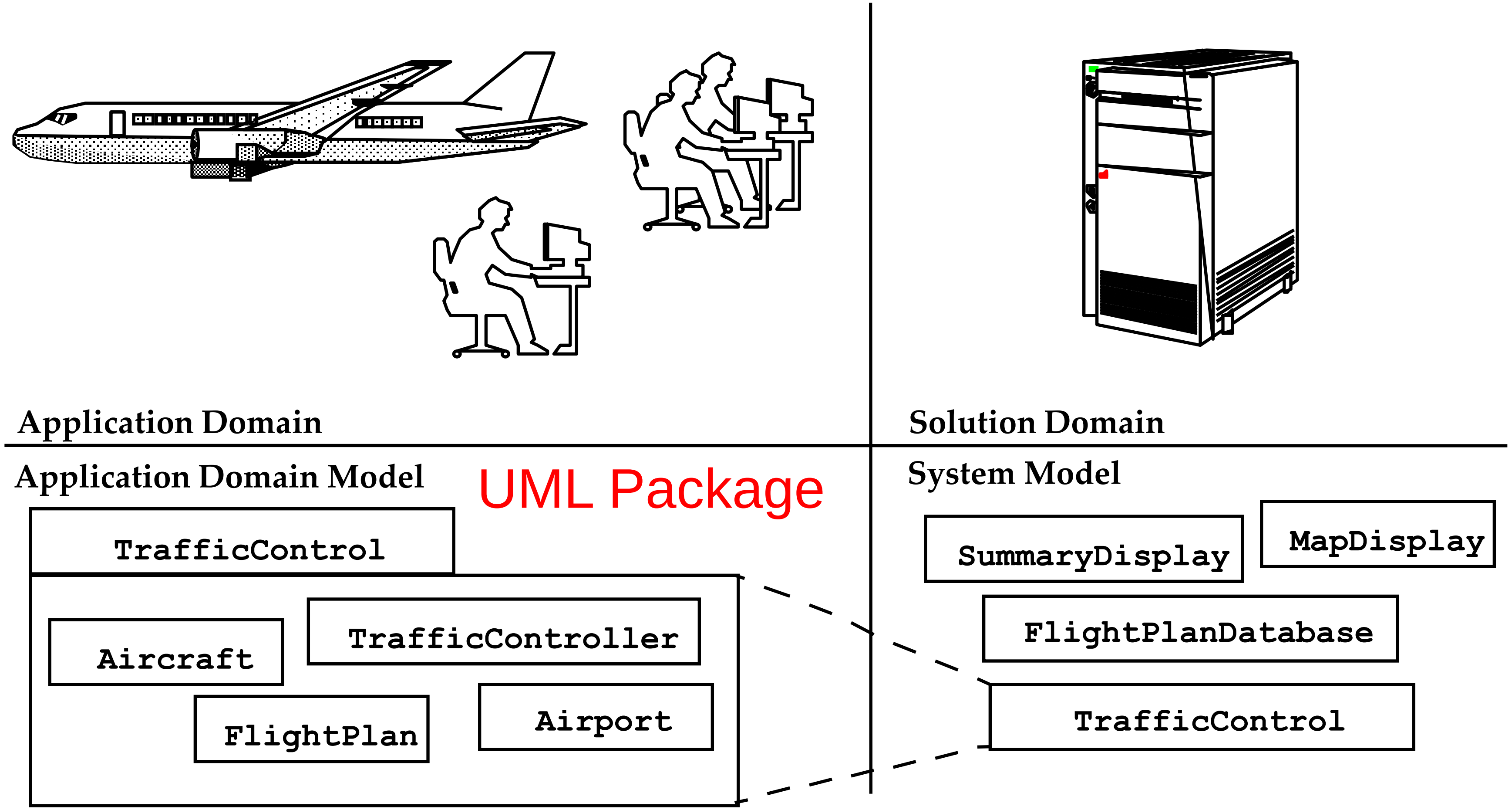
Application and Solution Domain

Modeling context

- Application Domain (Requirements Analysis)
  - The **environment** in which the system is operating

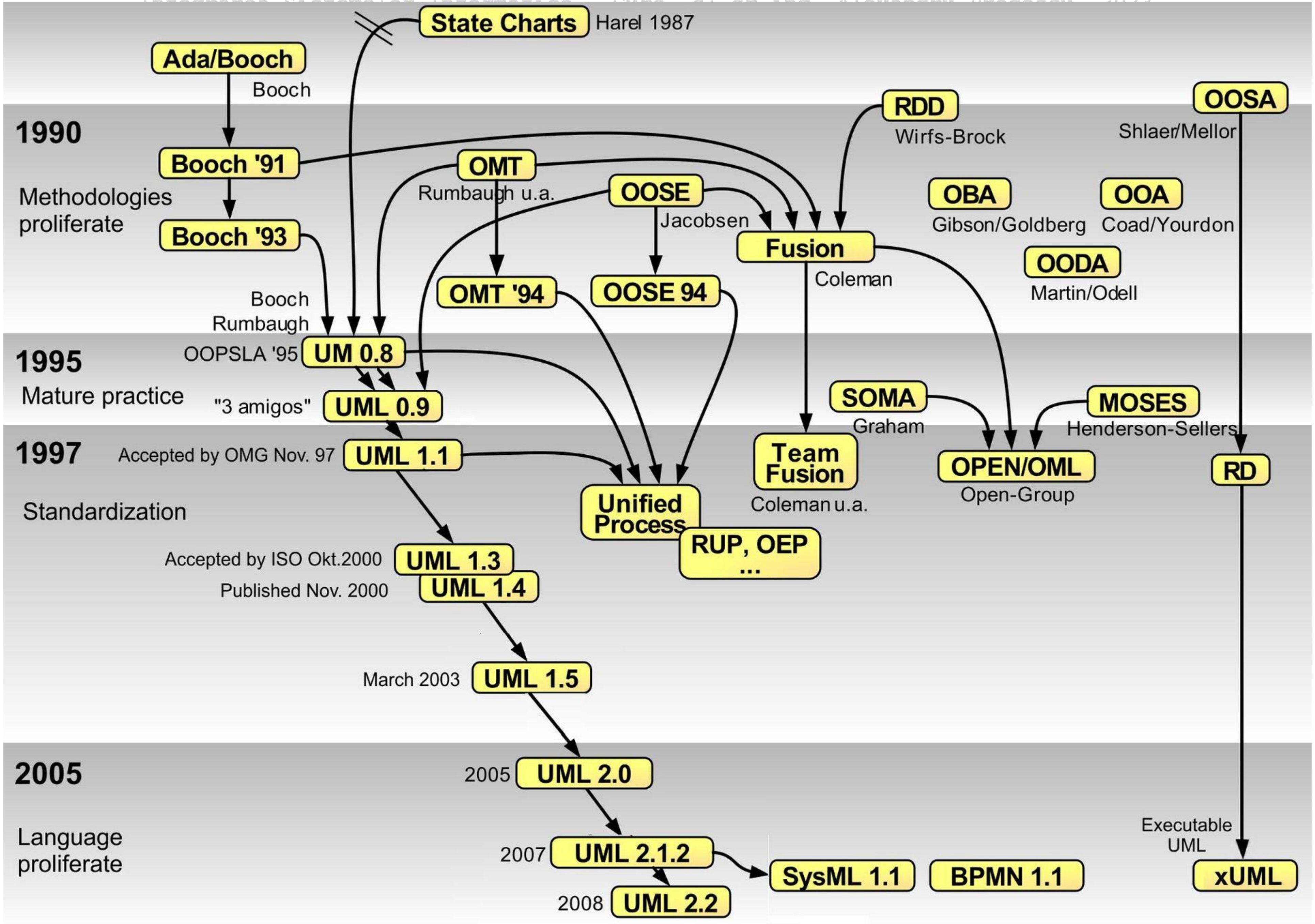
- Solution Domain (System Design, Object Design)
  - The available **technologies** to build the system

Object-oriented modeling

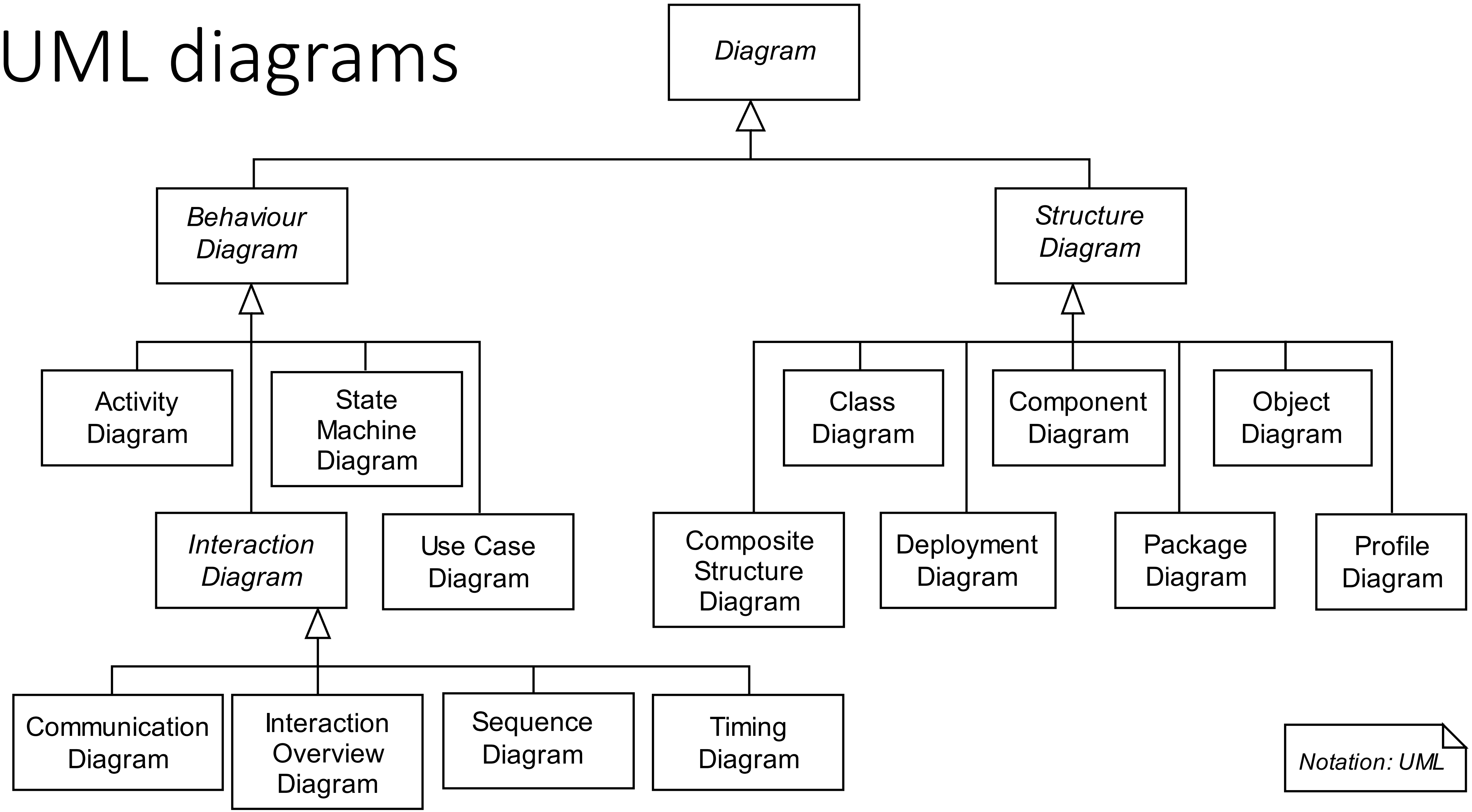


What is UML?

- UML (Unified Modeling Language)
  - An emerging standard for modeling object-oriented software.
  - Resulted from the convergence of notations from three leading object-oriented methods:
    - OMT (James Rumbaugh)
    - OOSE (Ivar Jacobson)
    - Booch (Grady Booch)
- Reference: “The Unified Modeling Language User Guide”, Addison Wesley, 1999.
- Supported by several CASE tools (Computer Aided Software Engineering)
  - Rational ROSE XDE
  - Rational Rhapsody
  - TogetherJ
  - etc.


State Charts
Harel 1987
Ada/Booch
Booch
1990
Methodologies proliferate
Booch '91
Booch '93
OMT
Rumbaugh u.a.
OOSE
Jacobsen
RDD
Wirfs-Brock
OOSA
Shlaer/Mellor
OBA
Gibson/Goldberg
OOA
Coad/Yourdon
Fusion
Coleman
OODA
Martin/Odell
OMT '94
OOSE 94
Booch
Rumbaugh
1995
Mature practice
OOPSLA '95
UM 0.8
"3 amigos"
UML 0.9
SOMA
Graham
MOSES
Henderson-Sellers
1997
Accepted by OMG Nov. 97
UML 1.1
Unified Process
Team Fusion
Coleman u.a.
OPEN/OML
Open-Group
RD
Standardization
RUP, OEP ...
Accepted by ISO Okt.2000
UML 1.3
Published Nov. 2000
UML 1.4
March 2003
UML 1.5
2005
2005
UML 2.0
Language proliferate
Executable UML
2007
UML 2.1.2
SysML 1.1
BPMN 1.1
xUML
2008
UML 2.2

UML diagrams



UML diagrams overview

Behavior (functional vs system)

- Use case diagrams
  - Describe the **functional behavior** of the system as seen by the **user**.
- Activity diagrams
  - Model the **dynamic behavior of a system**, in particular the **workflow**
  - Flowchart

Structure

- Class diagrams
  - Describe the **static structure** of the system
  - Objects, Attributes, Associations

UML diagrams overview

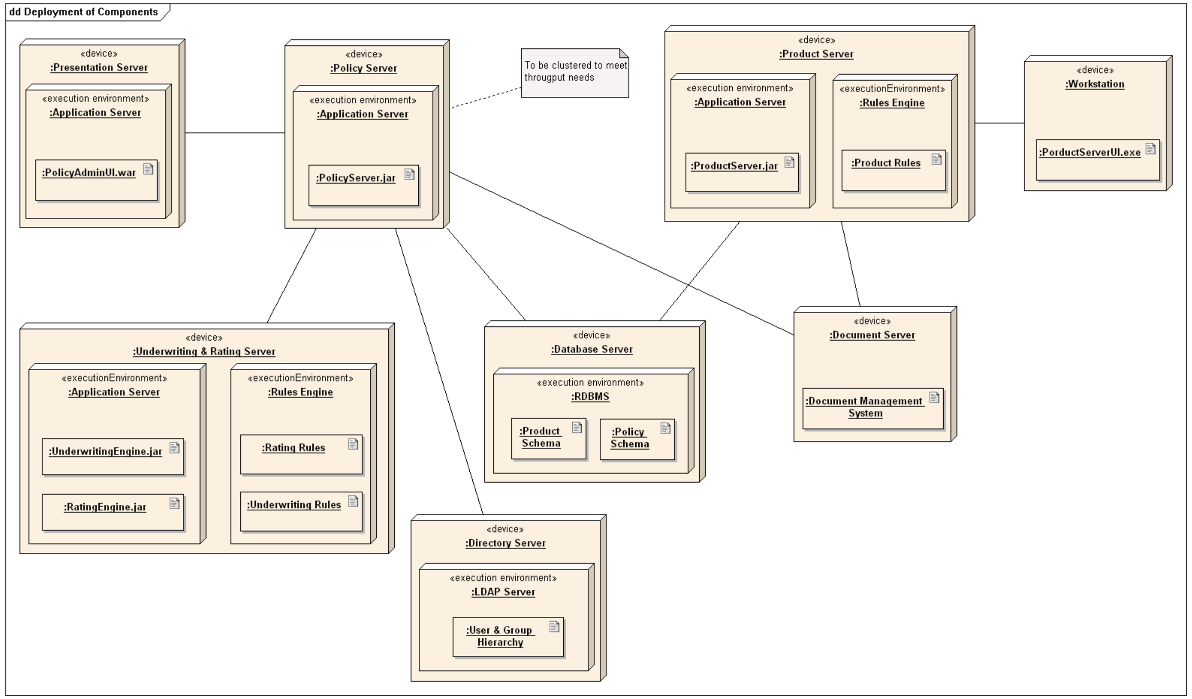
Behavior (system)

- Sequence diagrams
  - Describe the **dynamic behavior** between actors and the system
  - and between system components
- State machine diagrams
  - Describe the **dynamic behavior of an individual object**
  - Alternate name: Statechart Diagram
  - Finite State Automaton

UML diagrams overview

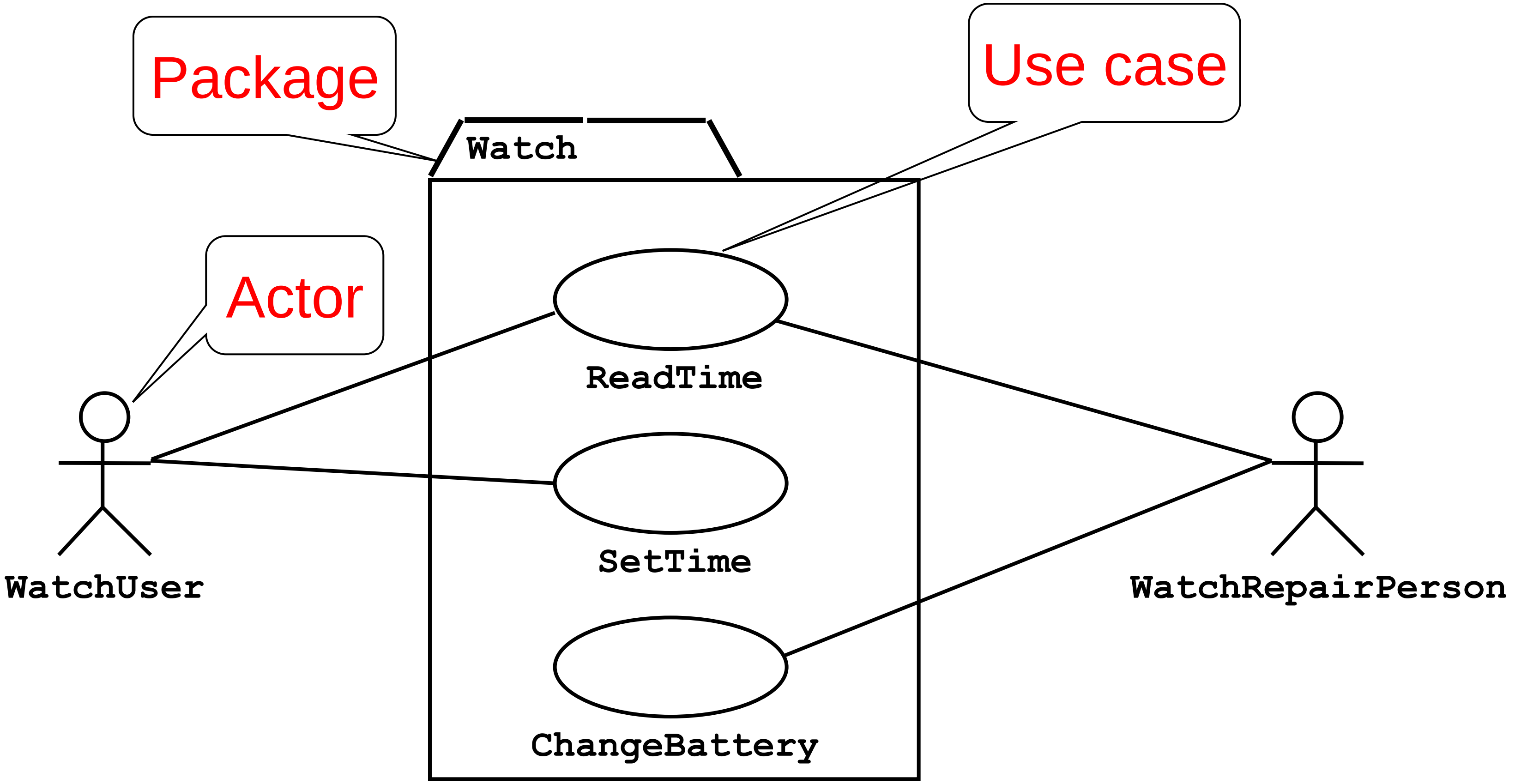
Structure (implementation)

- Component diagrams
- Deployment diagrams



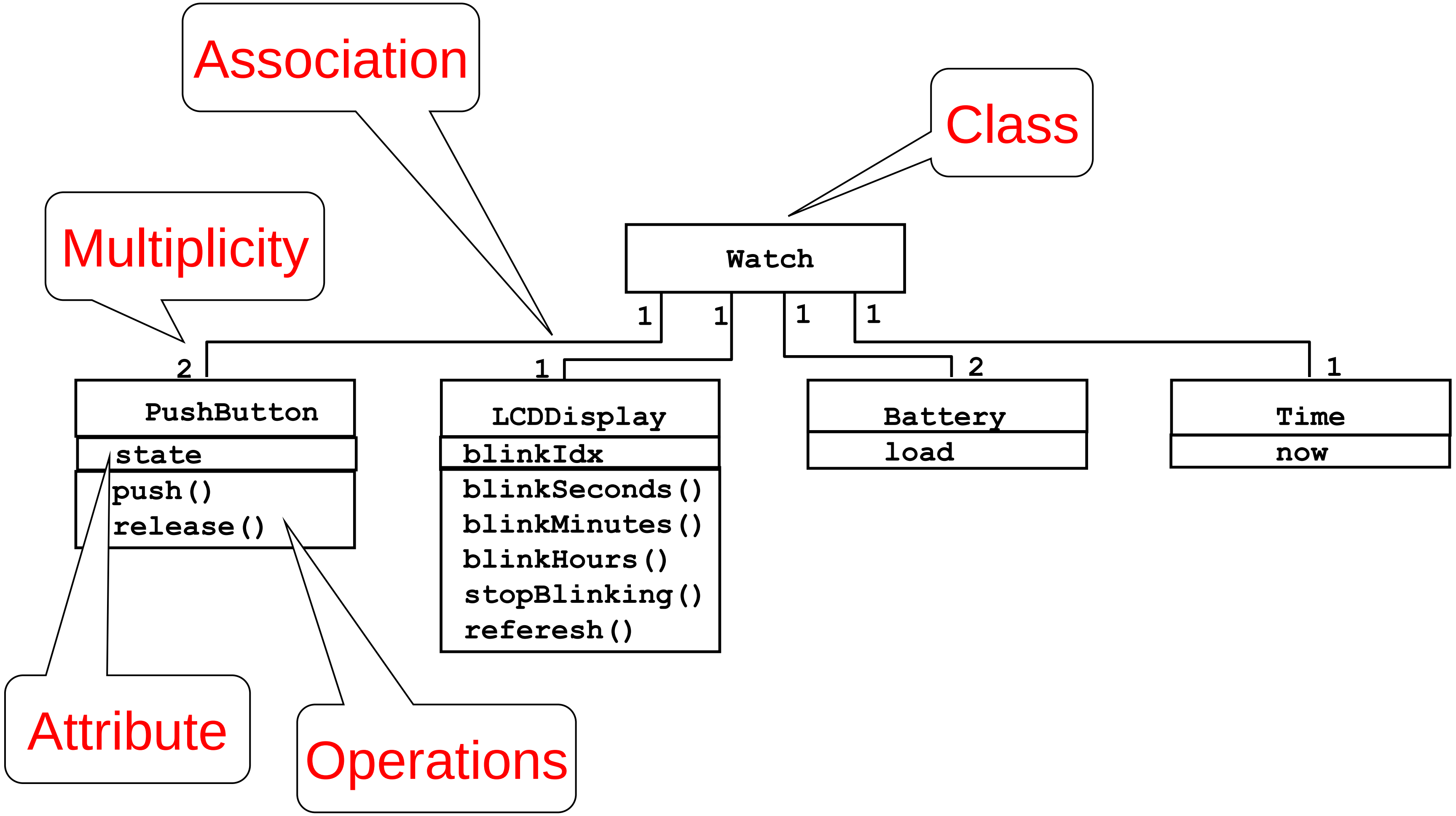
Example: Deployment diagram

UML overview: Use case diagrams



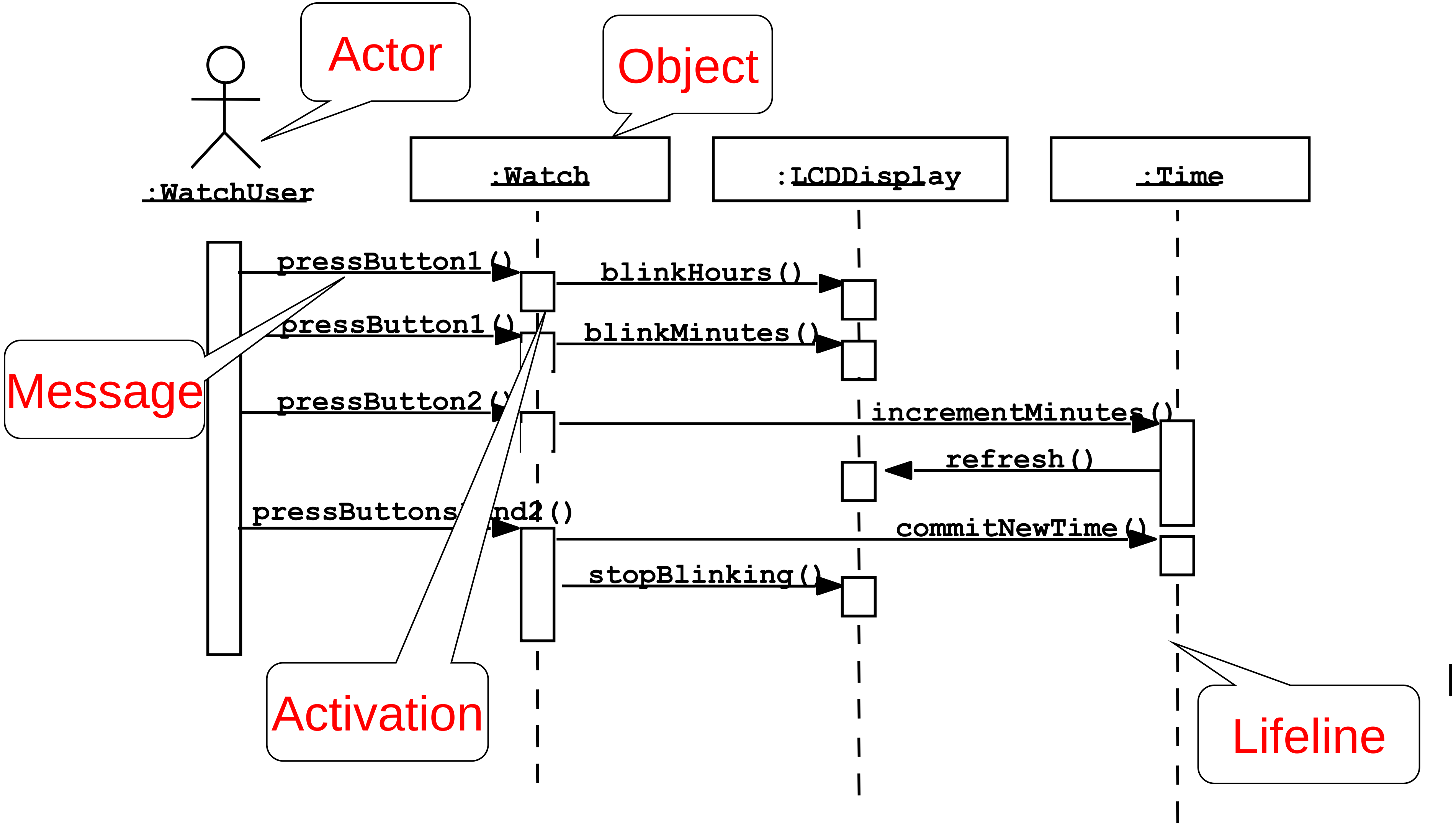
> represent the functionality of the system from the **user's point of view**

UML overview: Class diagrams



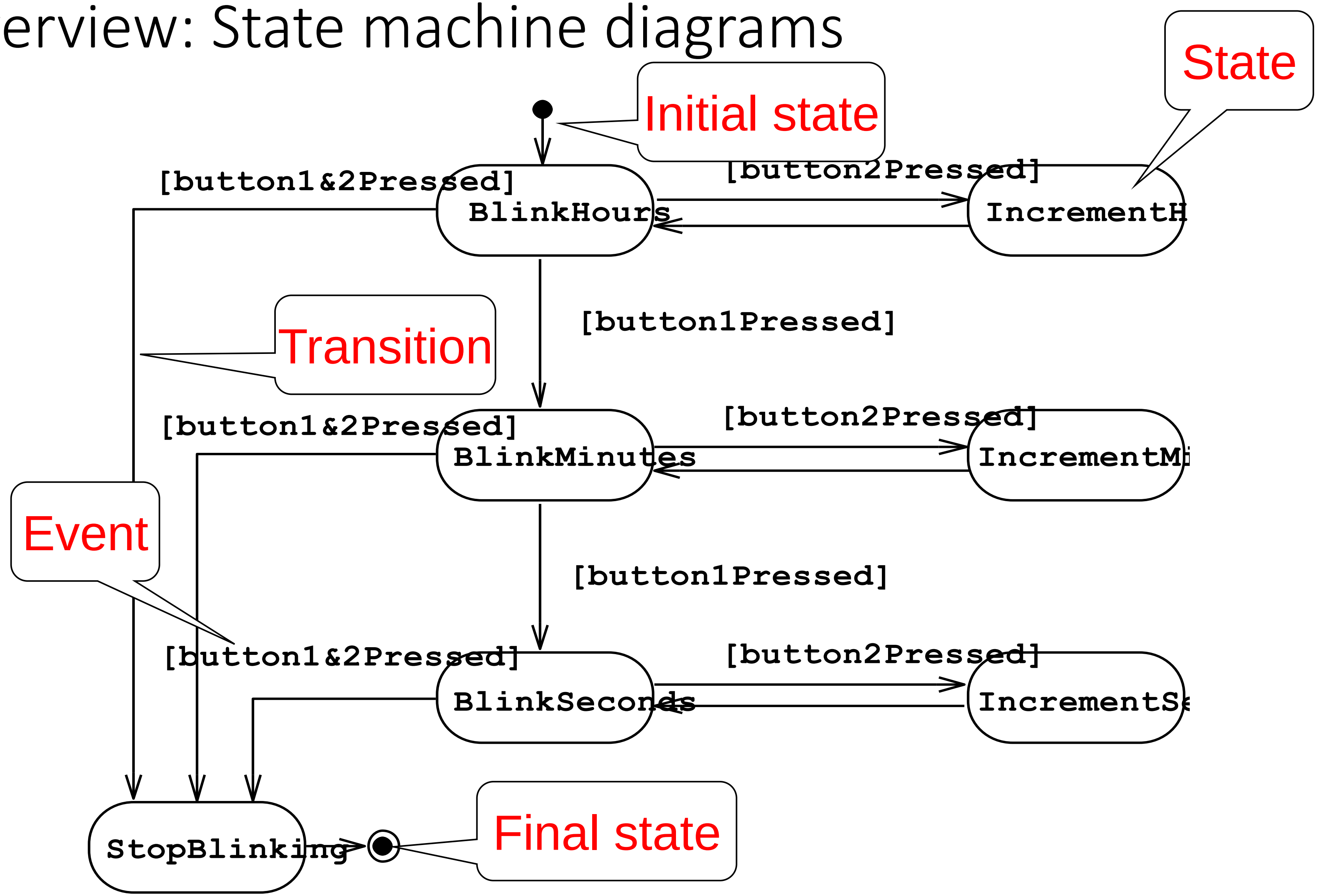
Class diagrams represent the structure of the system

UML overview: Sequence diagrams



Sequence diagrams represent the behavior as **interactions**

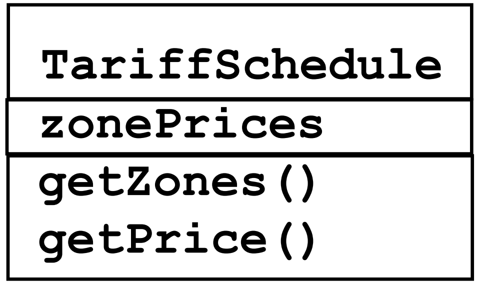
UML overview: State machine diagrams

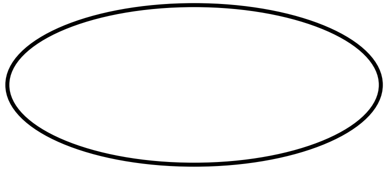


State machine diagrams represent behavior as **states and transitions**

UML core conventions (for all diagrams)

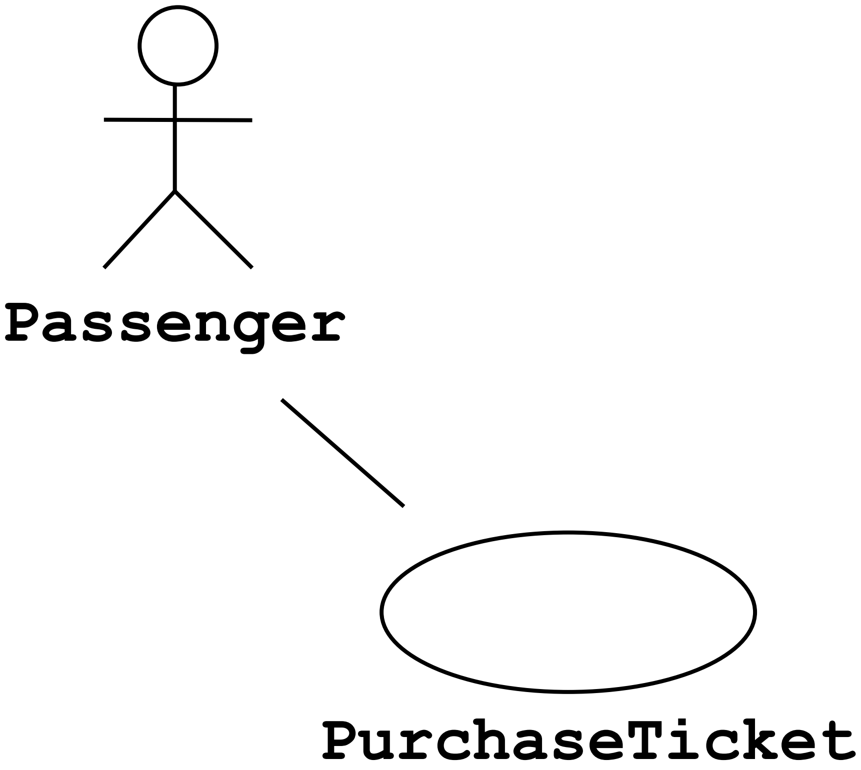
- Rectangles are **classes** or **instances**
- Ovals are **functions** or **use cases**
- Instances are denoted with an underlined names
  - `myWatch:SimpleWatch`
  - `Joe:Firefighter`
- Types are denoted with non underlined names
  - `SimpleWatch`
  - `Firefighter`
- Diagrams are graphs
  - **Nodes** are entities
  - **Arcs** are relationships between entities





PurchaseTicket

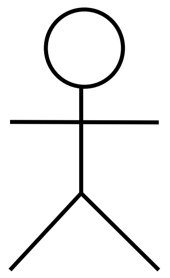
1. Use case diagrams



- Used during **requirements specification** to represent external behavior

- ***Actors*** represent roles – a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the **set of all use cases**. It is a complete description of the **functionality** of the system and its environment

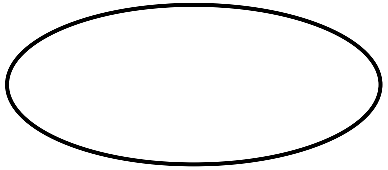
Use case diagrams: Actors



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

Use case diagrams: Use cases

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



PurchaseTicket

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use case diagrams: Example

Name: Purchase ticket

Participating actor: Passenger

Entry condition:

- Passenger standing in front of ticket distributor.
- Passenger has sufficient money to purchase ticket.

Exit condition:

- Passenger has ticket.

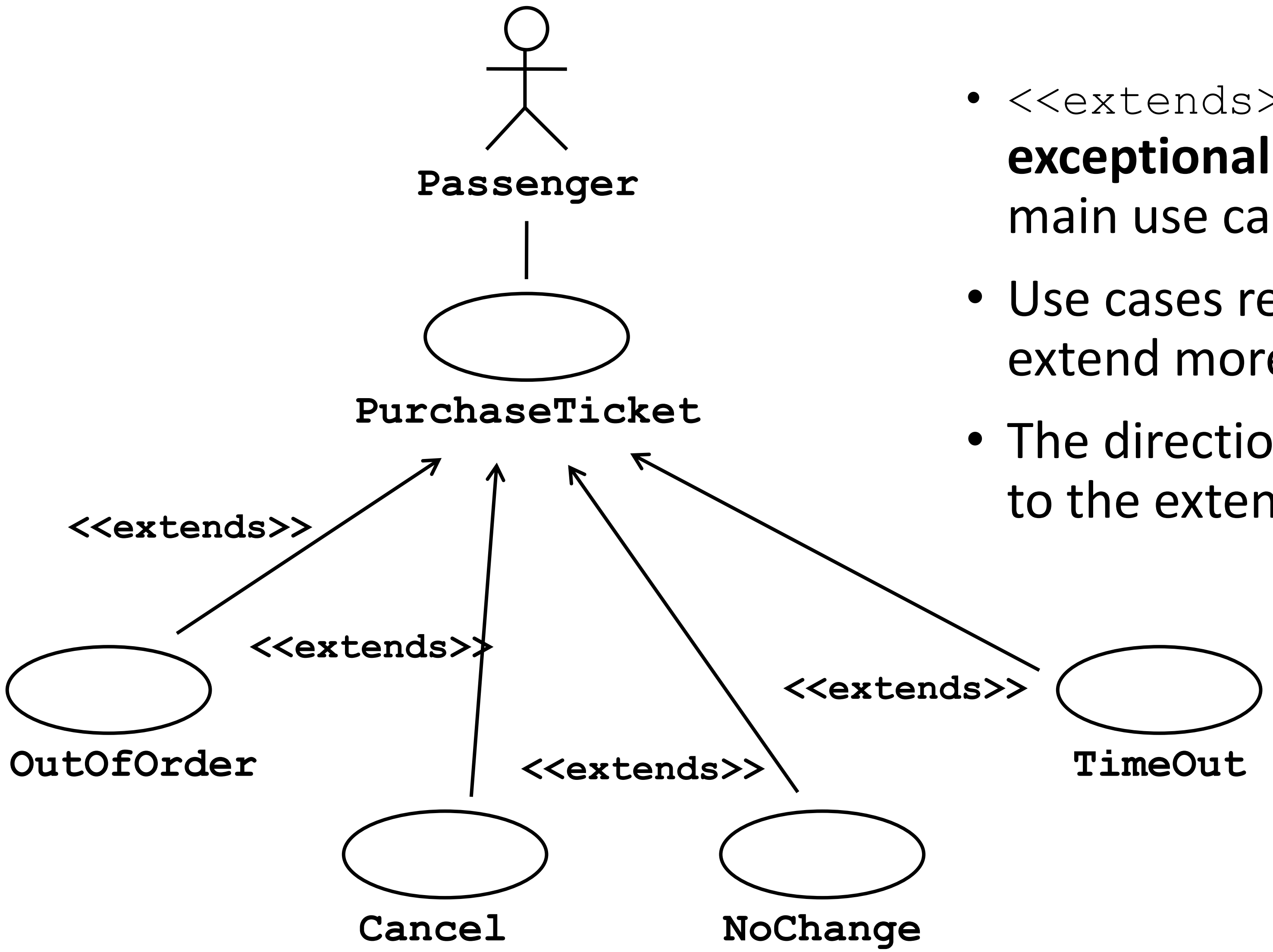
Event flow:

1. Passenger selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. Passenger inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

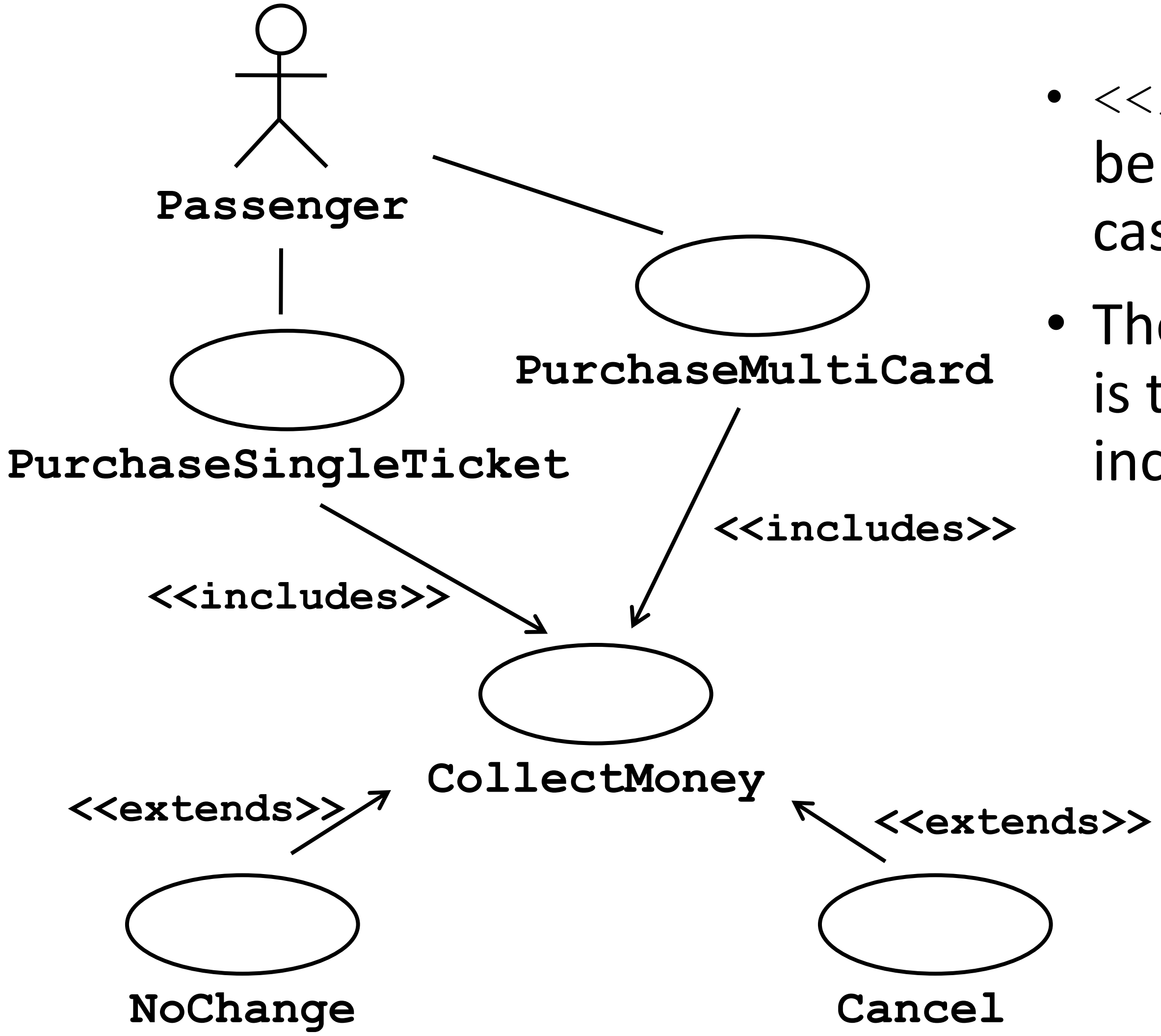
Exceptional cases!

The <<extends>> relationship



- `<<extends>>` relationships represent **exceptional cases** that are factored out of the main use case for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a `<<extends>>` relationship is to the extended use case

The <<includes>> relationship



- `<<includes>>` relationship represents behavior that is separated from the main use case for the purpose of **reuse**.
- The direction of a `<<includes>>` relationship is to the <using> use case (that is using the included use case).

Use case diagrams: summary

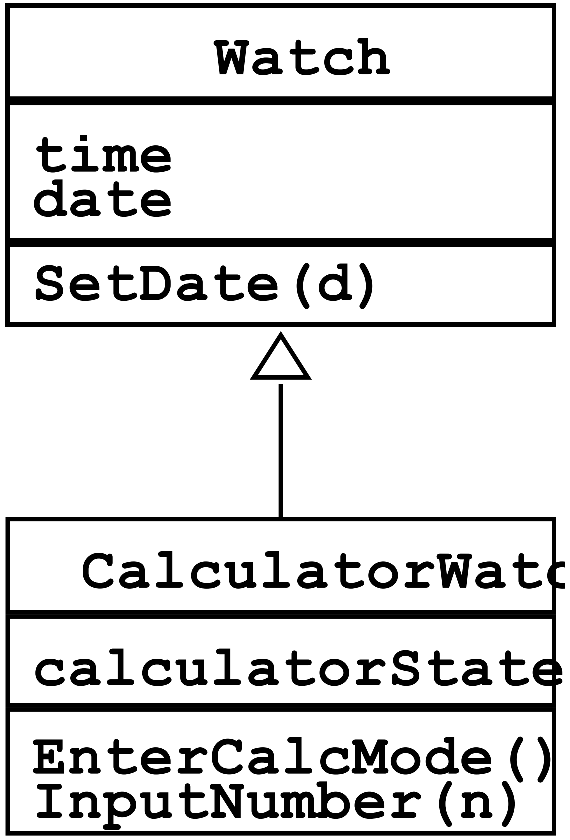
- Use case diagrams represent external behavior
- Use case diagrams are useful as an **index** into the use cases (see them all on a diagram)
- All use cases need to be described for the model to be useful

2. Class diagrams

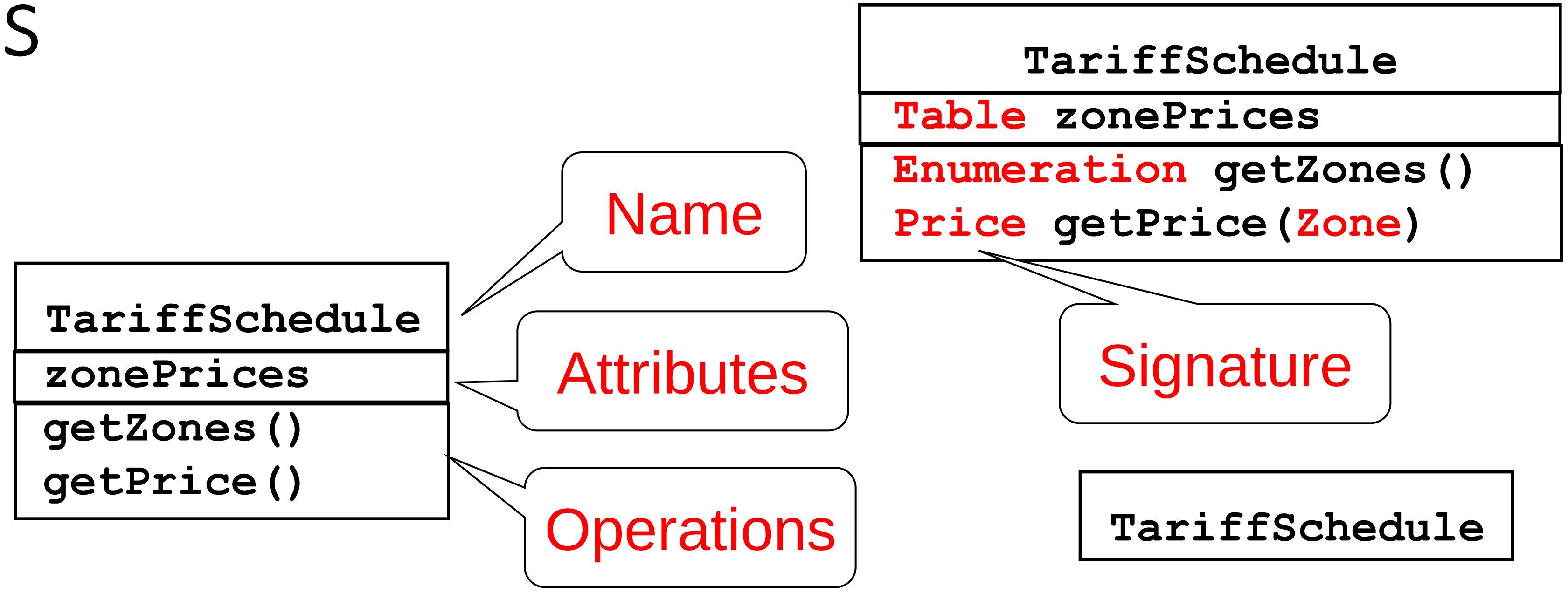
- Class diagrams represent the structure of the system.
- Used
  - during **requirements analysis** to model problem domain concepts
  - during **system design** to model subsystems and interfaces
  - during **object design** to model classes.

Abstract Data Types & Classes

- Abstract data type
  - Special type whose implementation is hidden from the rest of the system.
- Class:
  - An abstraction in the context of object-oriented languages
  - A class encapsulates both state (variables) and behavior (methods)



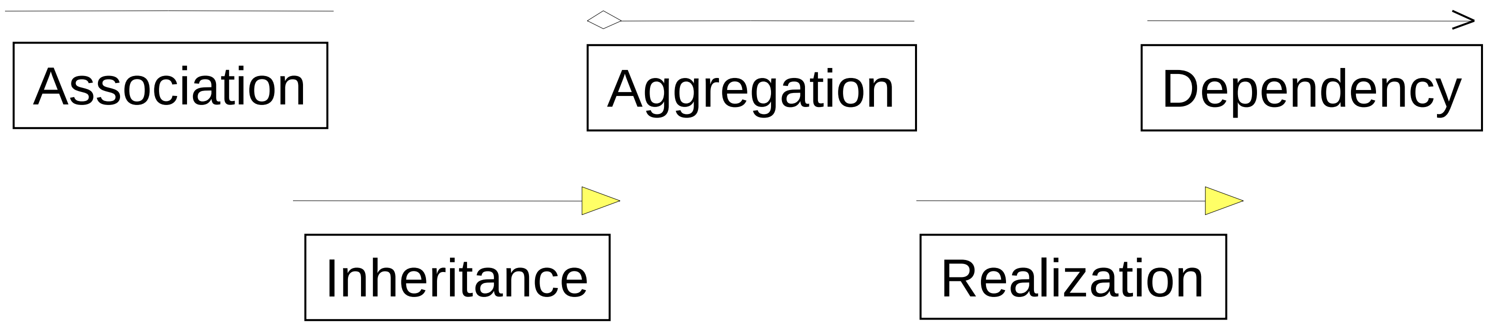
Classes



- A ***class*** represents a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations).***
- Each attribute has a ***type***.
- Each operation has a ***signature***.
- The class name is the only mandatory information.

Relationships

- Class diagrams may contain the following relationships:
  - Association, aggregation, dependency, realization/implementation, and inheritance
- Notation:

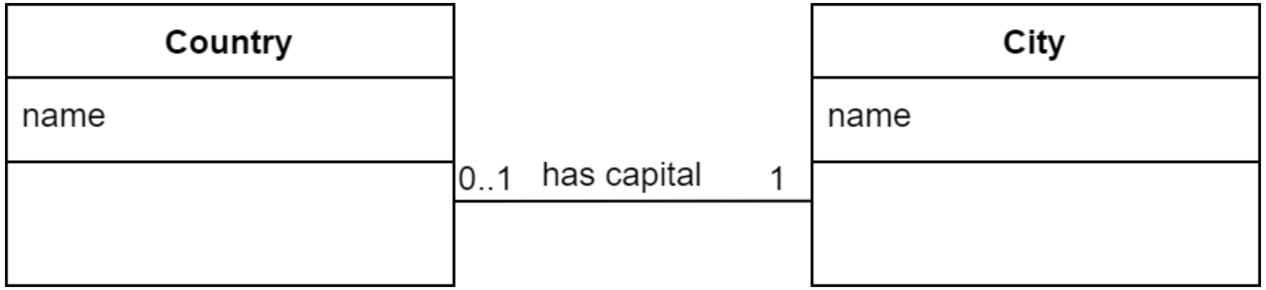


Associations

- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

1	**Exactly one**
0..*	**Zero or more**
1..*	**One or more**
0..1	**Zero or one**
2..7	**Specified range**

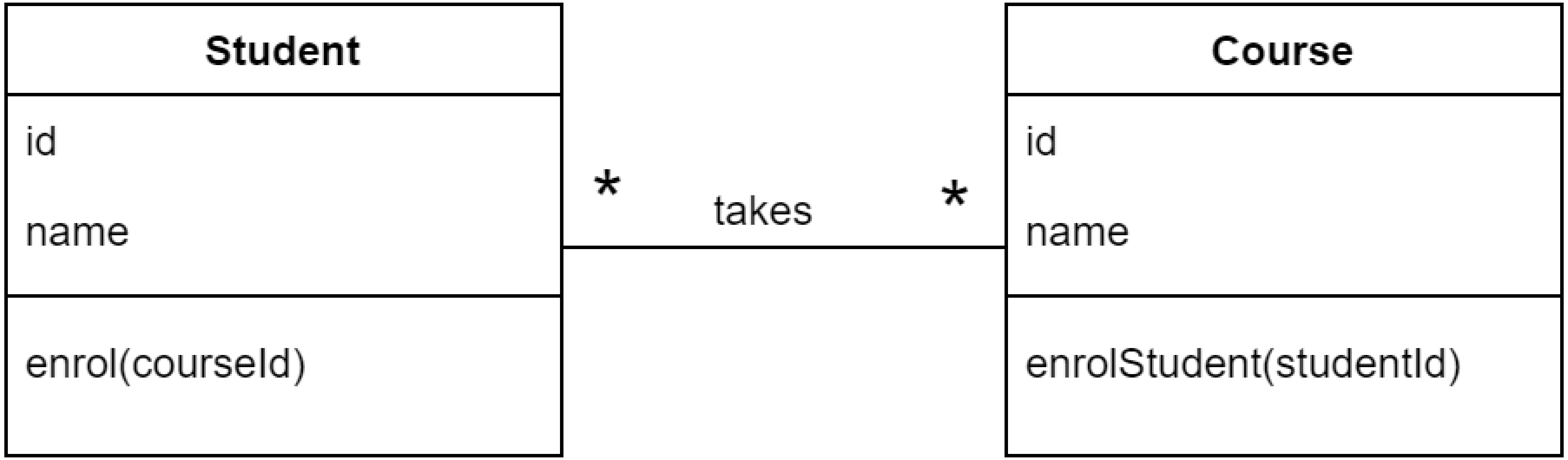
Associations



One-to-one association

One-to-many association

Associations

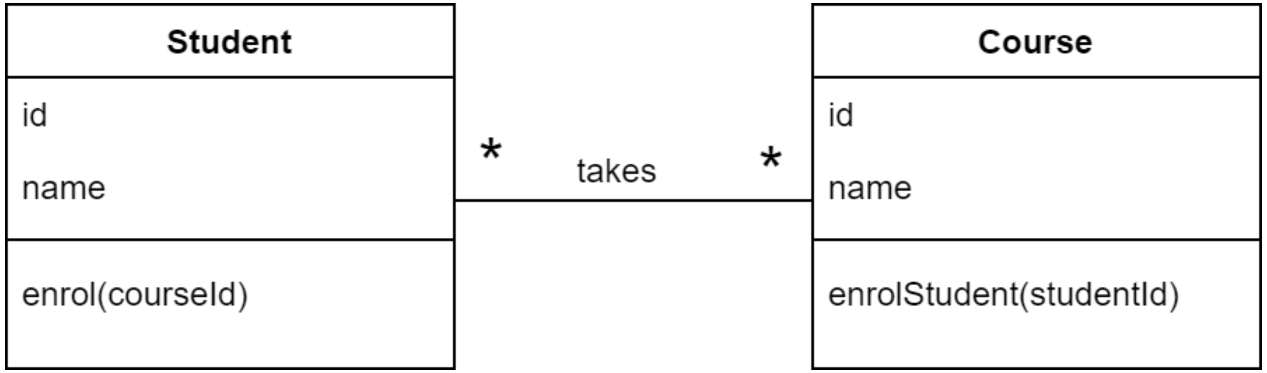


Many-to-many association

From Problem Statement to Object Model

Problem Statement: A course enrols many students. Each student can enrol to a course and is uniquely identified by a student ID.

Class diagram



From Problem Statement to Object Model

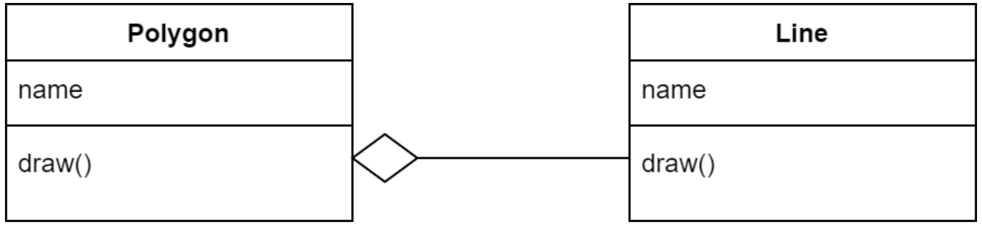
Problem Statement: A course enrols many students. Each student can enrol to a course and is uniquely identified by a student ID.

Java Code

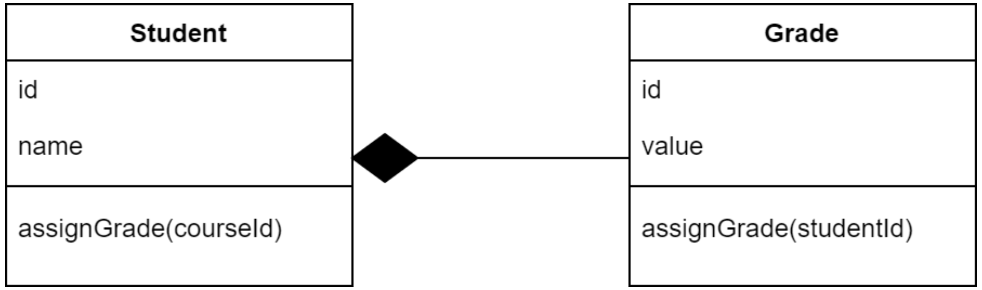
```
public class Course {
	private ArrayList<Student> students = new ArrayList<Student>();
	// ...
}

Public class Student {
	private int id;
	private ArrayList<Course> courses = new ArrayList(Course)();
	public boolean enrol(int courseId){
		// ...
	}
}
```

Aggregation

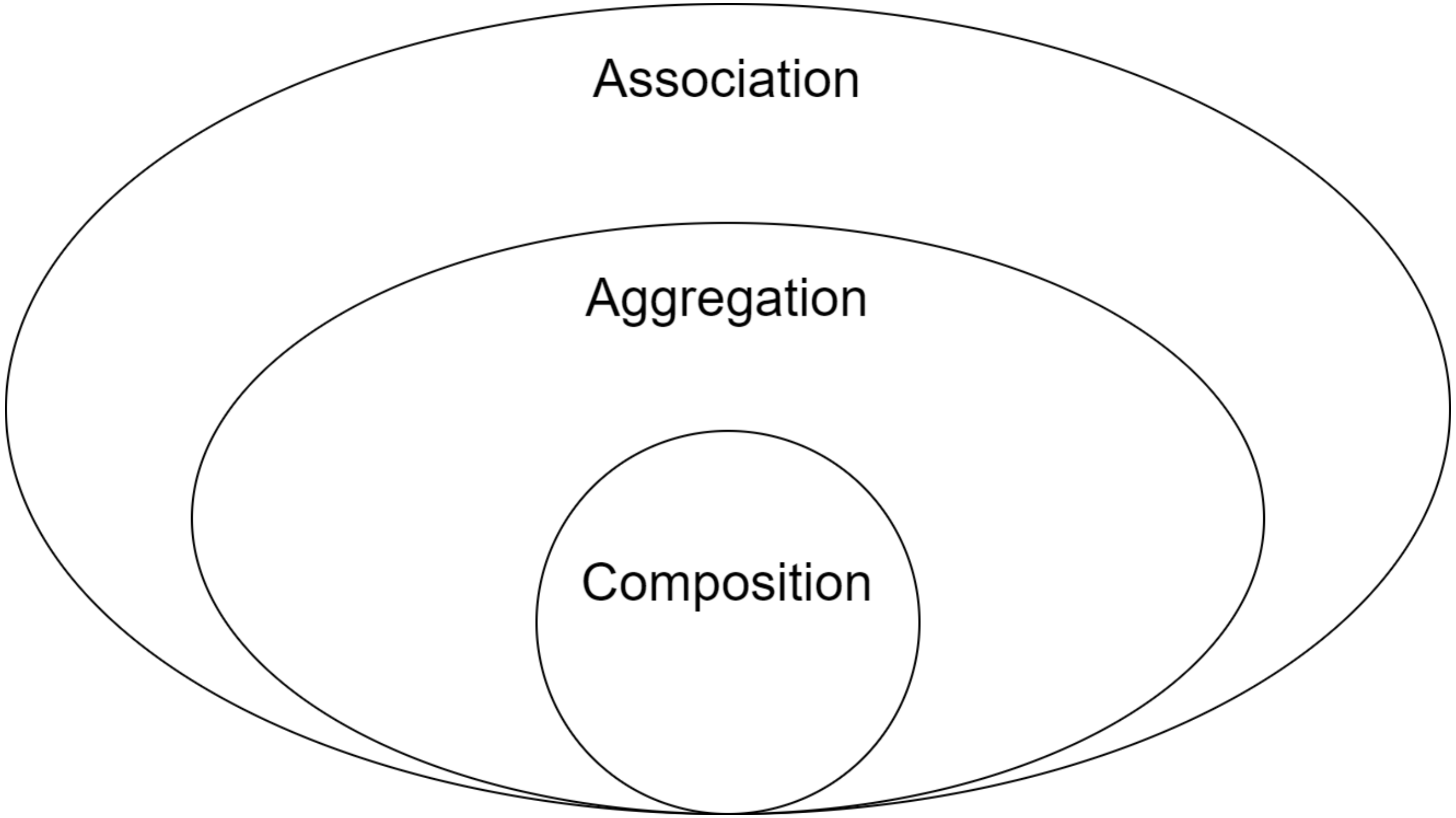
- An ***aggregation*** is a special case of association denoting a "consists of" hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children classes.



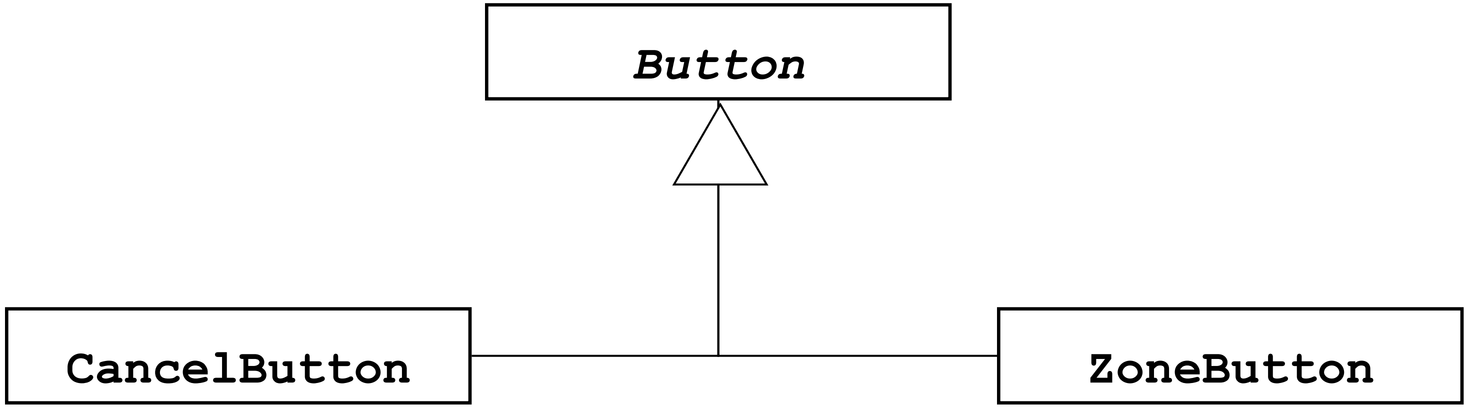
- A solid diamond denotes ***composition***, a strong form of aggregation where **components cannot exist without the aggregate**. (e.g., Bill of Materials)



Association > Aggregation > Composition



Inheritance

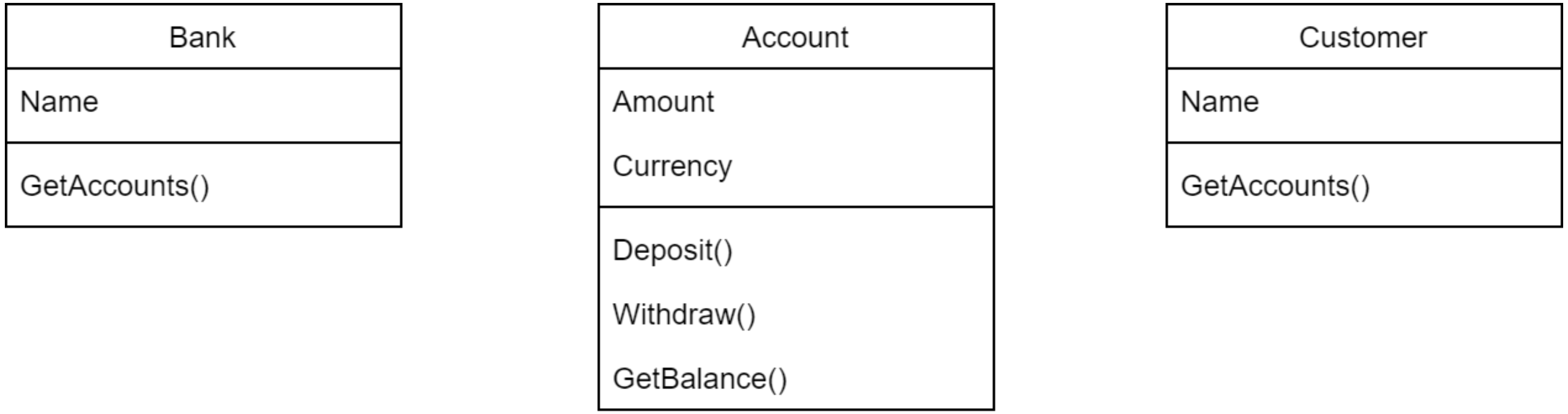


- The **children classes** inherit the attributes and operations of the **parent class**.
- Inheritance simplifies the model by eliminating redundancy.

Object Modeling in Practice: A Banking System

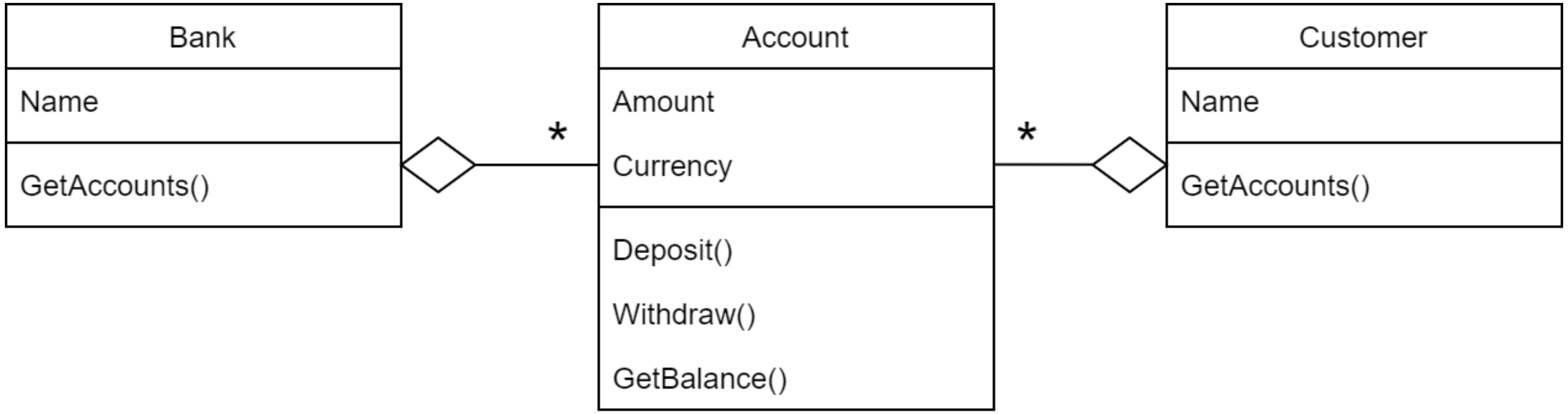
- Find new objects
- Define names, attributes and methods
- Find associations between objects
- Label the associations
- Determine the multiplicity of the associations

Object Modeling in Practice: A Banking System



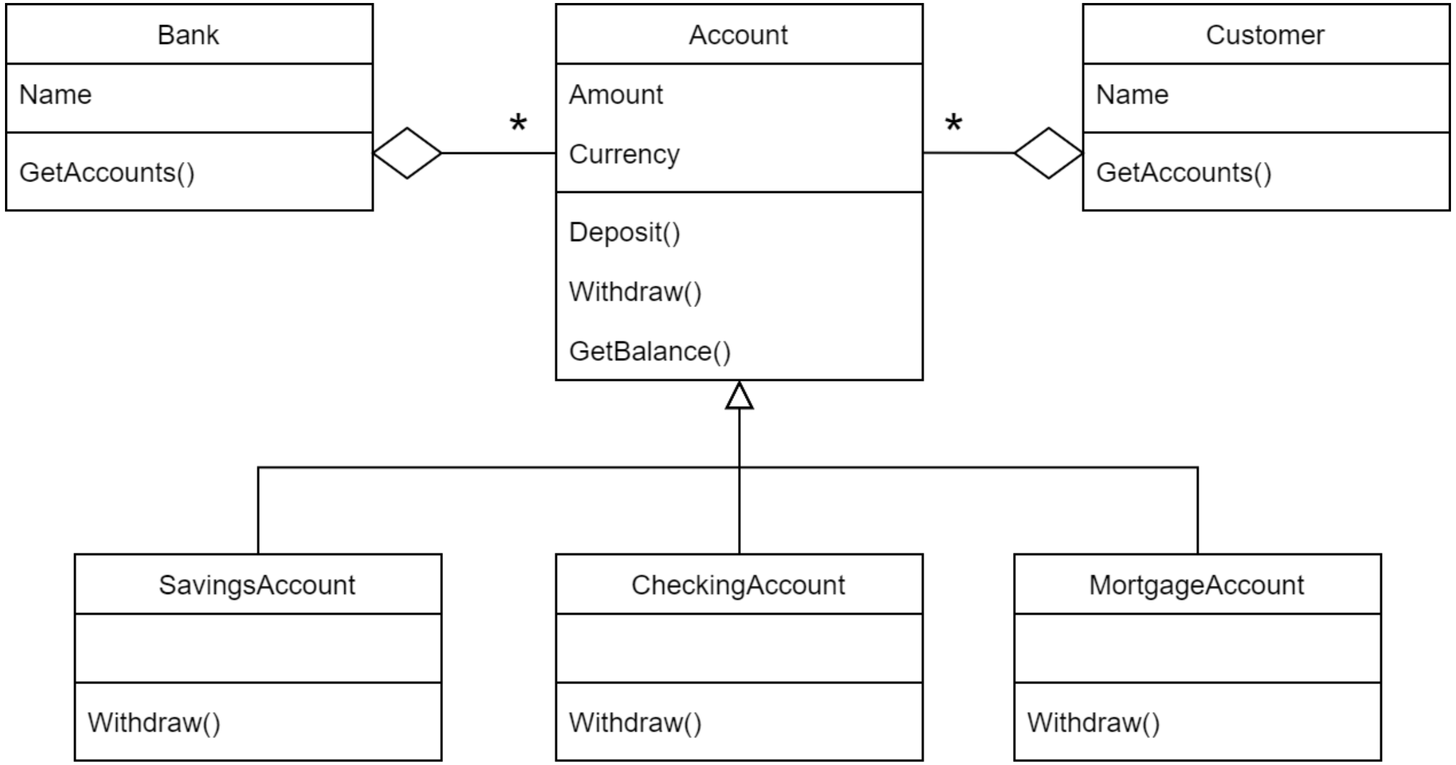
- **Find new objects**
- **Define names, attributes and methods**
- Find associations between objects
- Label the associations
- Determine the multiplicity of the associations

Object Modeling in Practice: A Banking System



- Find new objects
- Define names, attributes and methods
- **Find associations between objects**
- **Label the associations**
- **Determine the multiplicity of the associations**

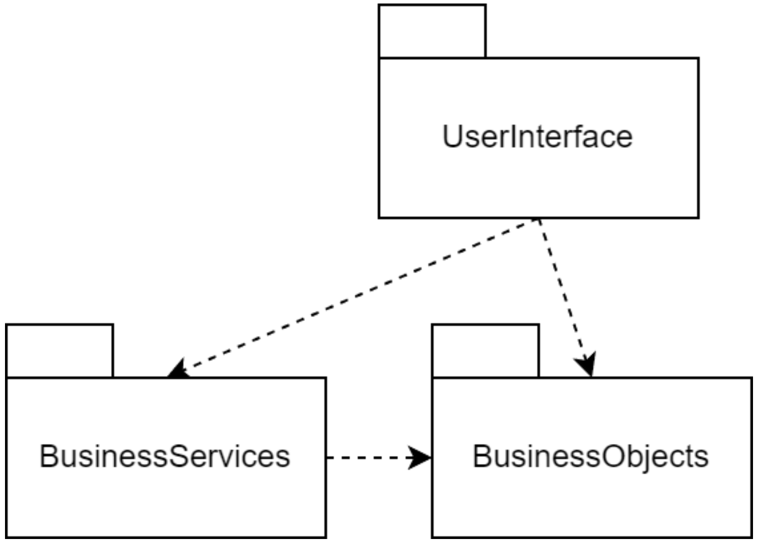
Object Modeling in Practice: A Banking System



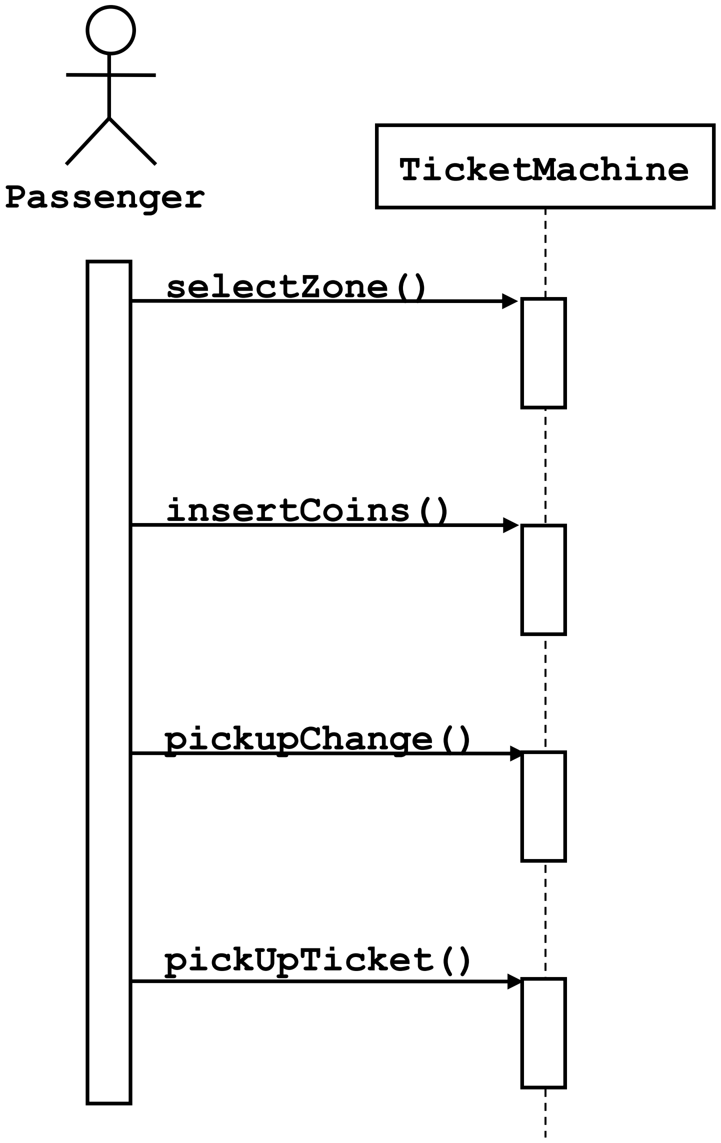
- **Categorize**

Packages

- A complex system can be decomposed into **subsystems**, where each subsystem is modeled as a package
- A package is a UML mechanism for organizing **elements into groups** (usually not an application domain concept)
- Packages are the basic grouping construct with which you may organize UML models to increase their readability.



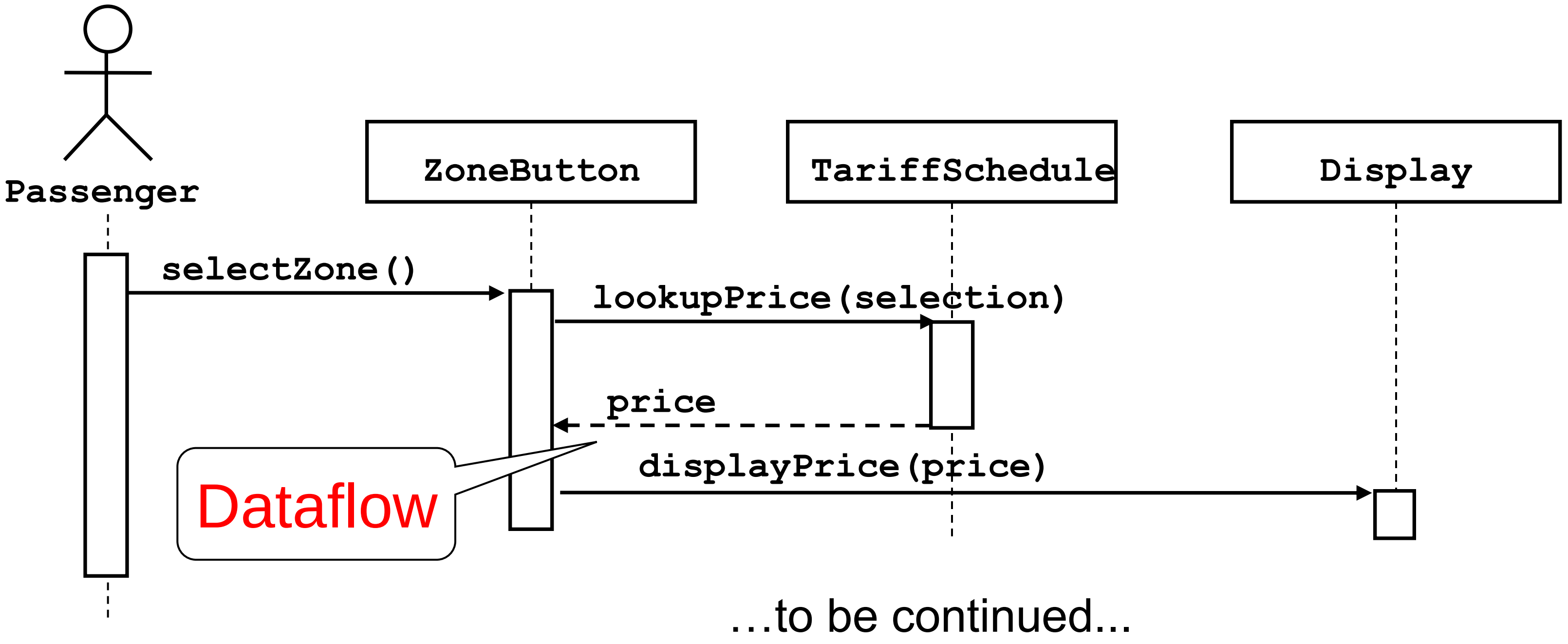
3. UML sequence diagrams



- Used during **system design**
  - to refine subsystem interfaces and interactions
- Also used during **requirements analysis**
  - To refine use case descriptions
  - to find additional objects (“participating objects”)

- ***Classes*** are represented by columns
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles
- ***Lifelines*** are represented by dashed lines

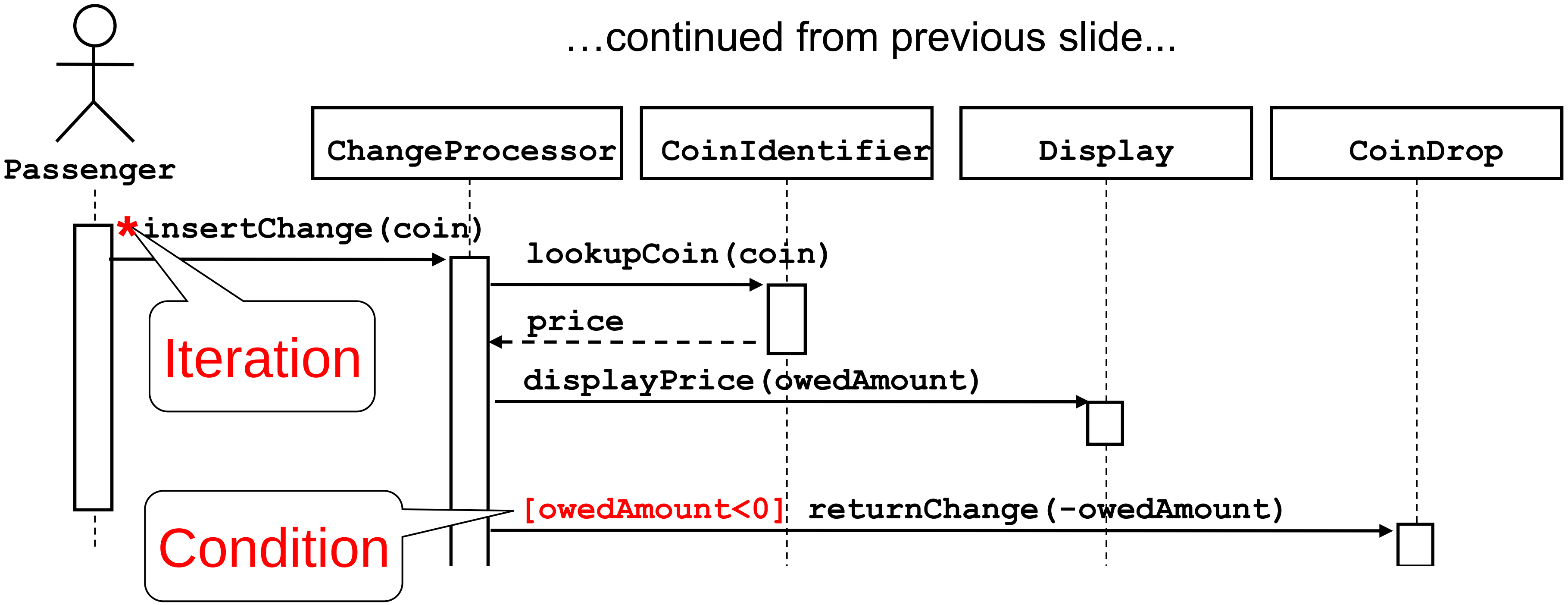
Nested messages



- The source of an arrow indicates the activation which sent the message
- An activation is as long as all **nested activations**
- Horizontal **dashed arrows** indicate data flow
- Vertical **dashed lines** indicate lifelines

Iteration & condition

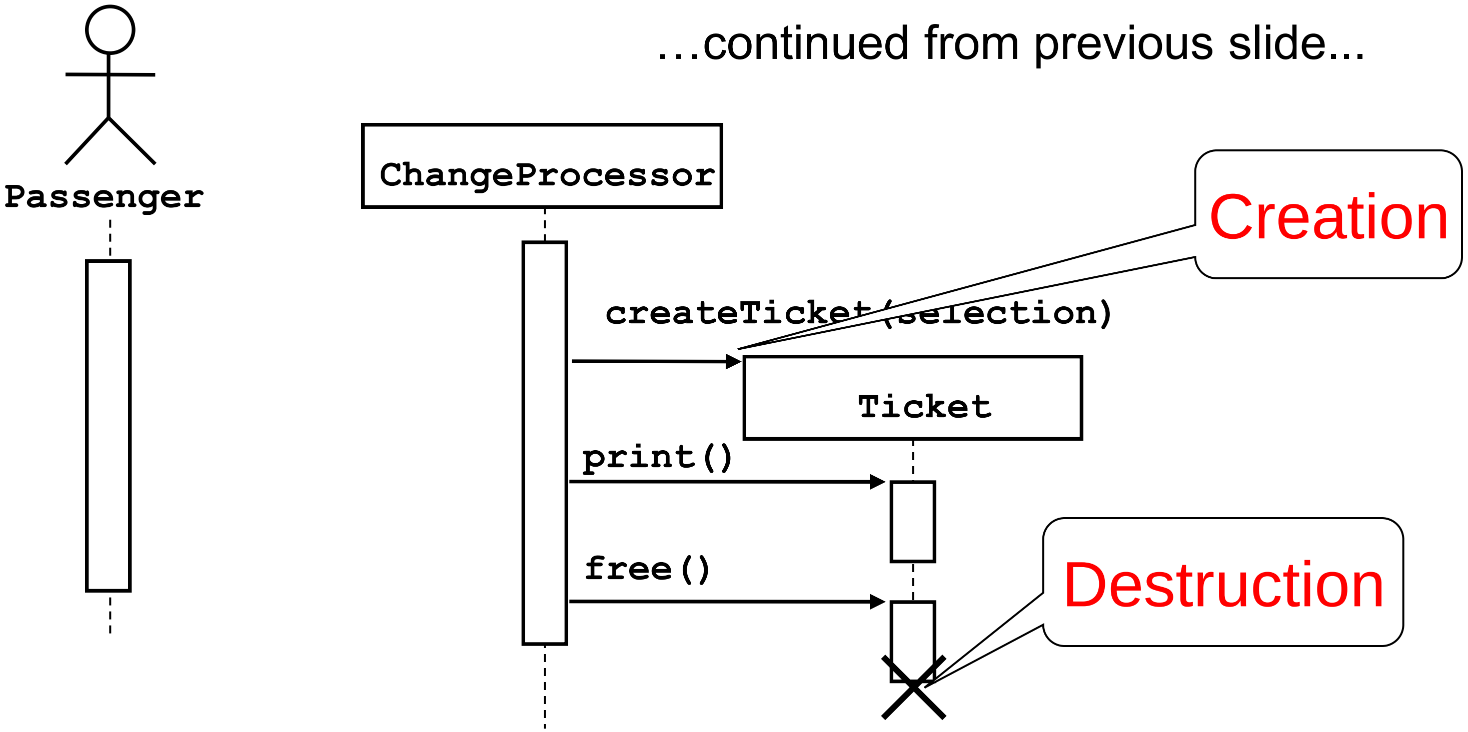
…continued from previous slide...



…to be continued...

- Iteration is denoted by a * preceding the message name
- Condition is denoted by boolean expression in [ ] before the message name

Creation and destruction

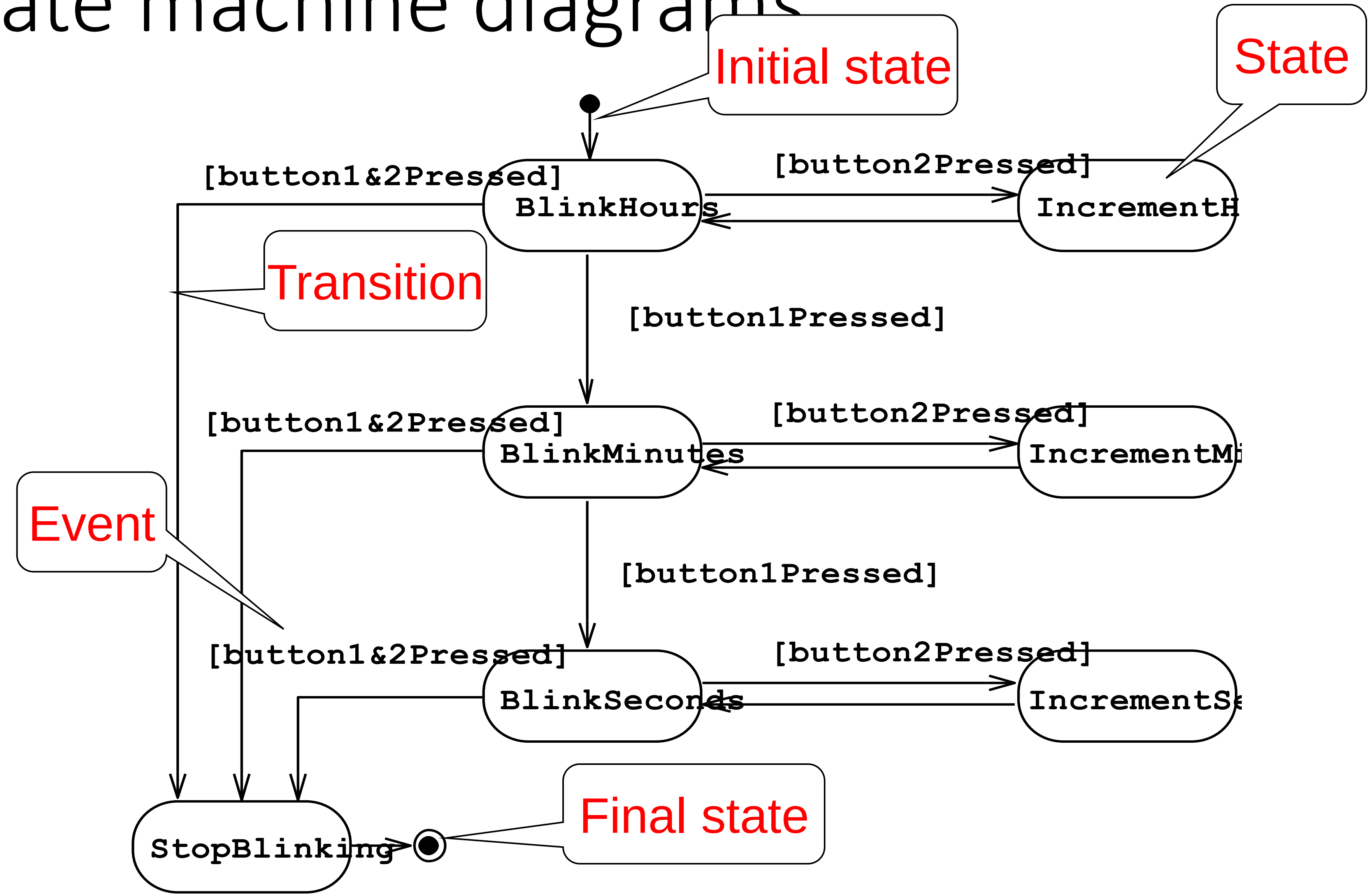


- Creation is denoted by a message arrow pointing to the object.
- Destruction is denoted by an X mark at the end of the destruction activation.
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

Sequence diagram summary

- UML sequence diagram represent **behavior in terms of interactions**
- Time consuming to build but can reveal **fine details** (interactions)
- Complement the class diagrams (which represent structure)

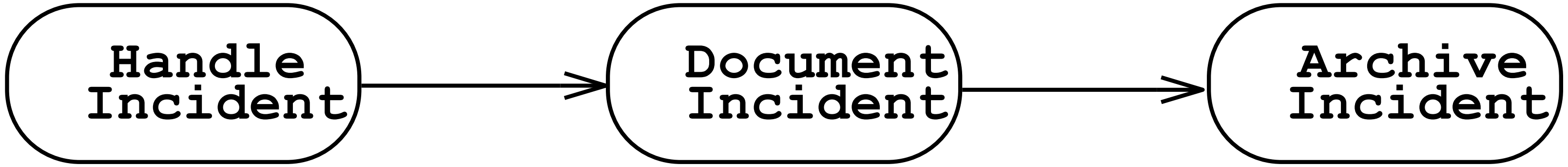
4. State machine diagrams



State Machine diagrams represent behavior as states and transitions

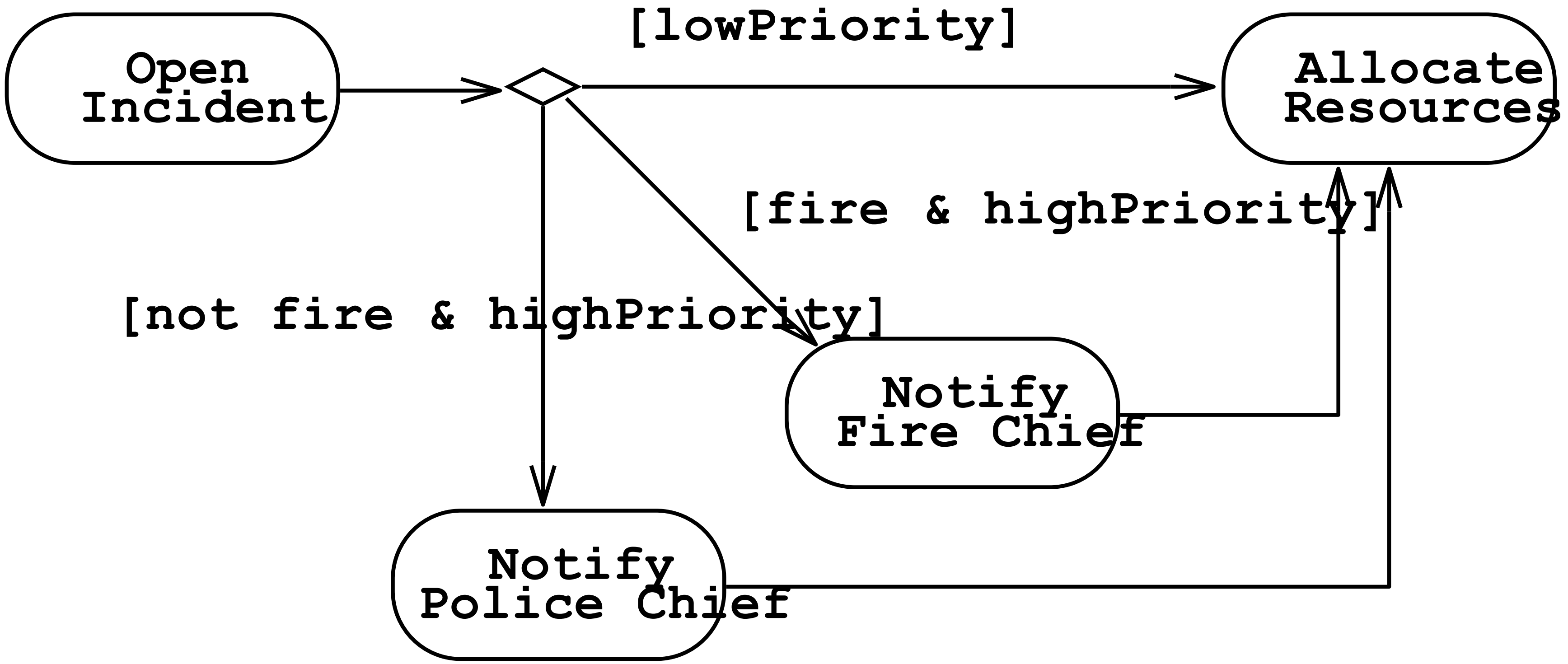
5. Activity diagrams

- An activity diagram shows flow control within a system



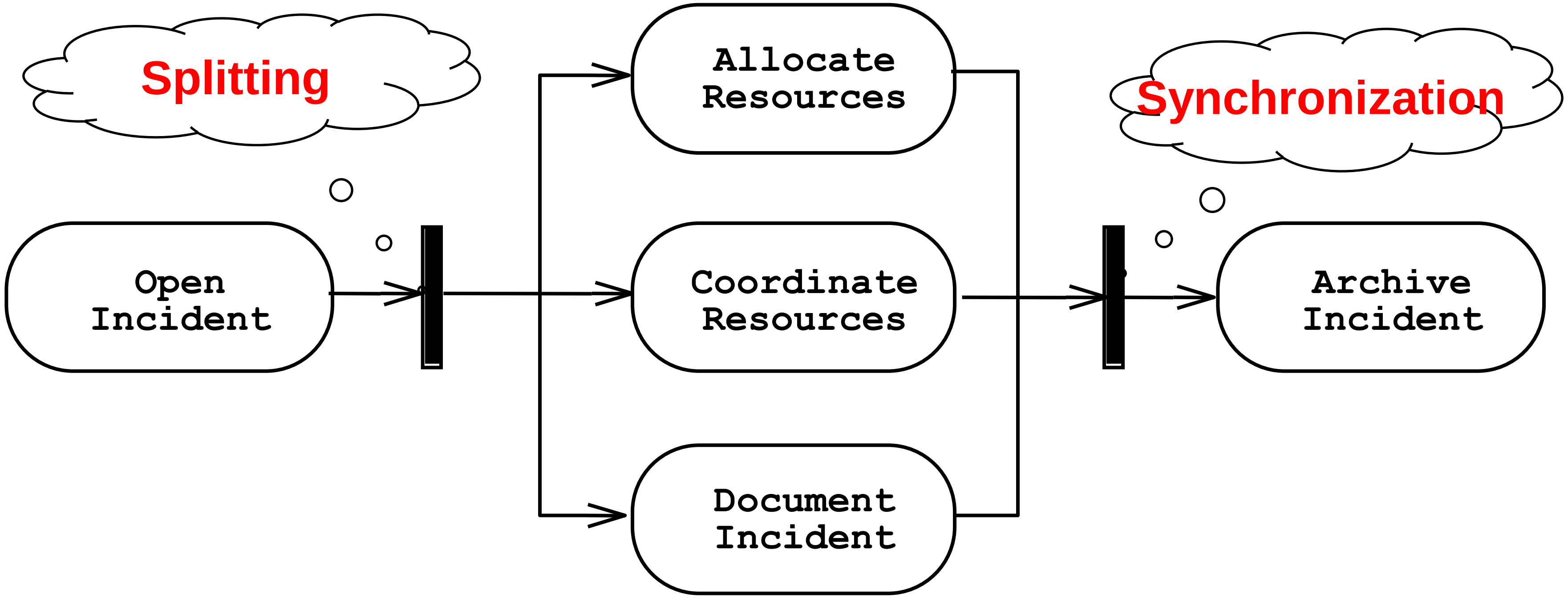
- An activity diagram is a special case of a statechart diagram in which states are activities ("functions") instead of states
- Activities can be further decomposed (modeled by another activity diagram)

Activity Diagrams: Modeling Decisions



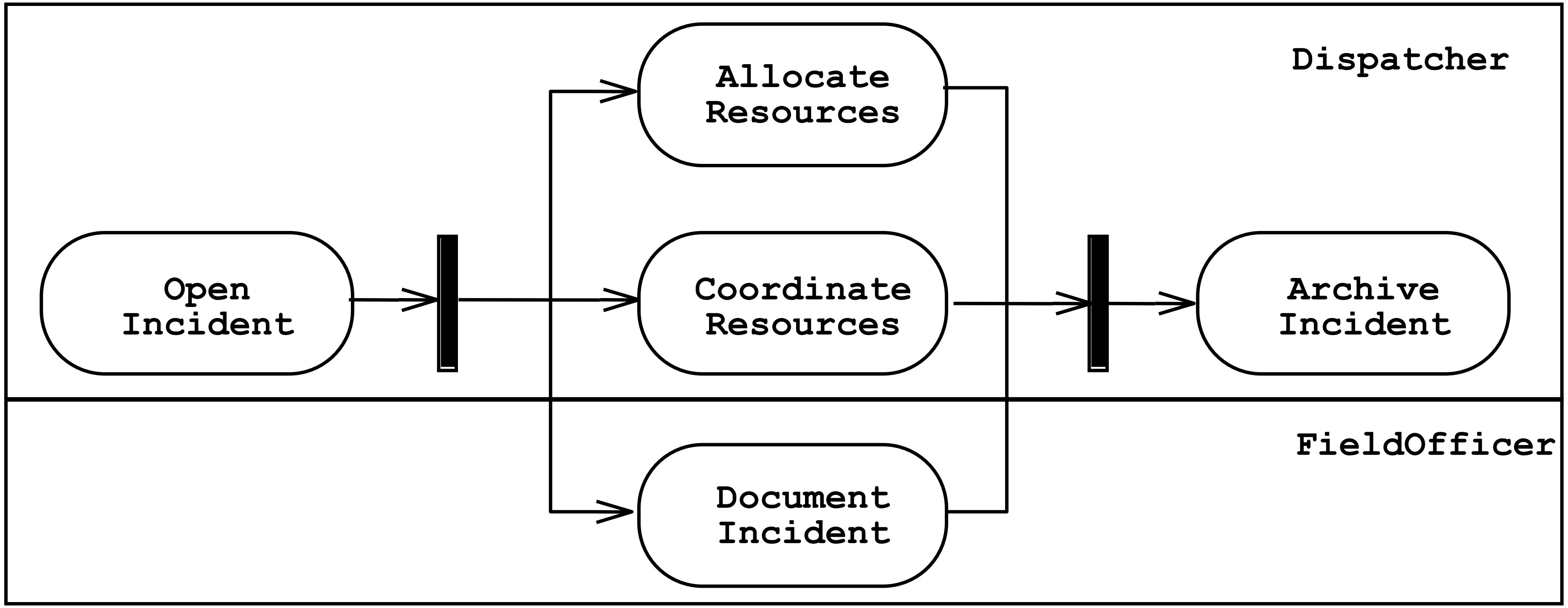
Activity Diagrams: Modeling Concurrency

- Synchronization of multiple activities
- Splitting the flow of control into multiple threads



Activity Diagrams: Swimlanes

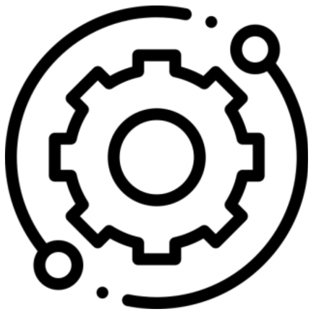
- Actions may be grouped into swimlanes to denote the object or subsystem that implements the actions.



What should be done first? Coding or Modeling?

- It all depends..

- Forward Engineering:
  - Creation of code from a model
  - Greenfield projects
- Reverse Engineering:
  - Creation of a model from code
  - Interface or reengineering projects
- Roundtrip Engineering:
  - Move constantly between forward and reverse engineering
  - Useful when requirements, technology and schedule are changing frequently



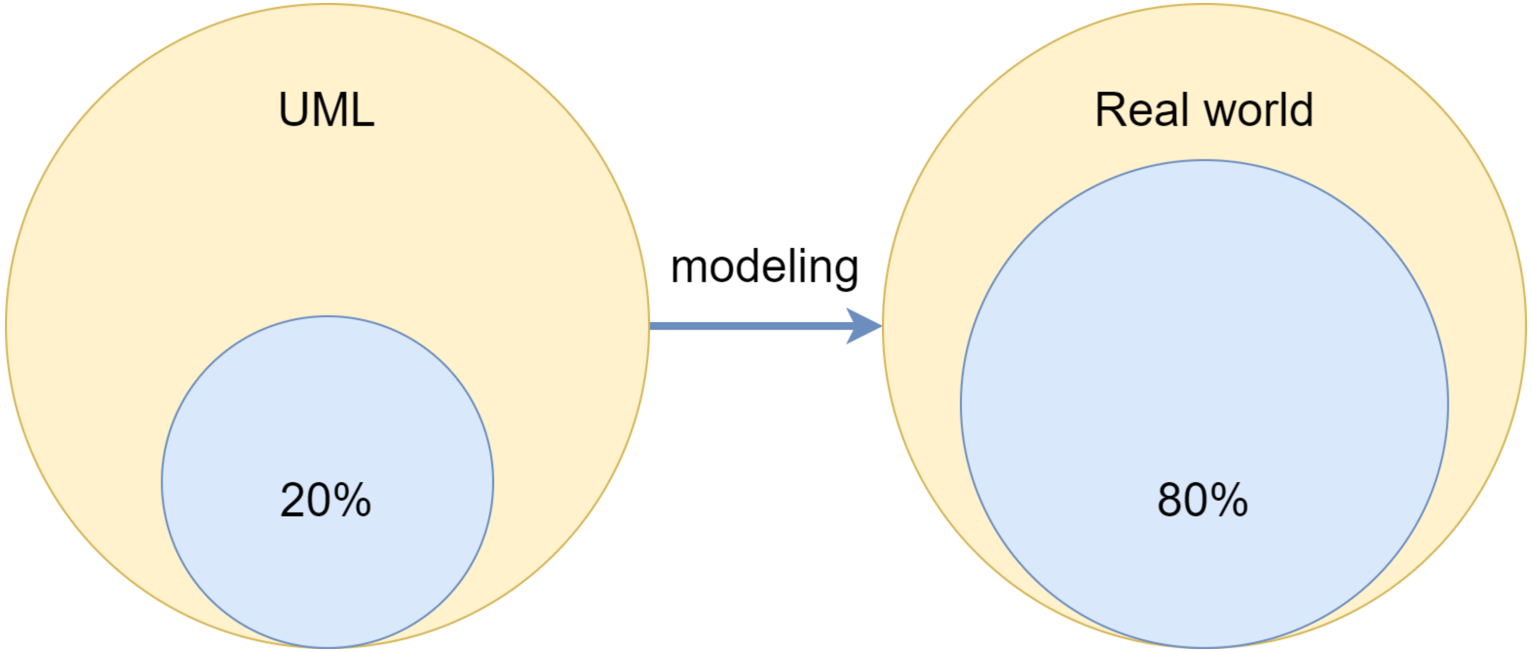
UML Summary

- UML provides a wide variety of notations for representing many aspects of software development
  - Powerful, but complex language
  - Can be misused to generate unreadable models
  - Can be misunderstood when using too many exotic features

- We can start by creating:
  - Functional models: **use case diagram**
  - Object models: **class diagram**
  - Dynamic models: **sequence diagrams**, **state machine** and **activity diagrams**

UML seems complicated?

- You can model 80% of most problems by using about 20% UML



Resources

- [StarUML Documentation]
- [Bernd Bruegge & Allen H. Dutoit, Object-Oriented Software Engineering - Using UML, Patterns, and Java]