Informatics II for engineering sciences (MSE)

Chapter II – Software Engineering, Principles, Tools, Lifecycles

Bernd Brügge Ph.D. , Jan Knobloch, Emitzá Guzmán

Homework #3

Solution / Discussion

Questions

In the last Lecture we talked about Interfaces and the Introduction into Software Engineering:

Answer the following questions:

1. What is the difference between an Interface implemented by a class and an abstract class as superclass while using Inheritance?

2. When we talk about Software Projects we have to deal with Complexity. Describe two different possibilities on how to deal with Complexity?

3. A typical Software Development Lifecycle consists of different activities, are all of these activities equally important? Could we leave some activities out, if yes which ones?

4. We talked about functional and object oriented decomposition, describe the trade-offs using one over the other, when would you use which type of decomposition?

5. What is the main difference between Drawing and Modeling?

6. We learned about 3 major types of models, briefly describe the differences.

Session Outline

Session #3 - #8

- Introduction in Software Engineering
- Software Development Lifecycle
- Model based Software Development
- Software Patterns

Session #9 - #13

- Databases
  - Introduction to databases
  - SQL
  - Mapping Code and Databases

Session Outline

Session #3 - #8

- Introduction in Software Engineering
- Software Development Lifecycle
- **Model based Software Development**
- Software Patterns

Session #9 - #13

- Databases
  - Introduction to databases
  - SQL
  - Mapping Code and Databases

Session #4 – Detailed Outline

Model based Software Development

- Software Lifecycle definition - redefined
- Software Lifecycle tailoring
- Software Lifecycle model definition
  - Examples
- Unified Modeling Language
  - Use case diagrams
  - Class diagrams
  - Communication diagrams
  - Component diagrams
  - Statechart diagrams
  - Deployment diagrams

Session #4 – Detailed Outline

Model based Software Development

- **Software Lifecycle definition - redefined**
- Software Lifecycle tailoring
- Software Lifecycle model definition
  - Examples
- Unified Modeling Language
  - Use case diagrams
  - Class diagrams
  - Communication diagrams
  - Component diagrams
  - Statechart diagrams
  - Deployment diagrams

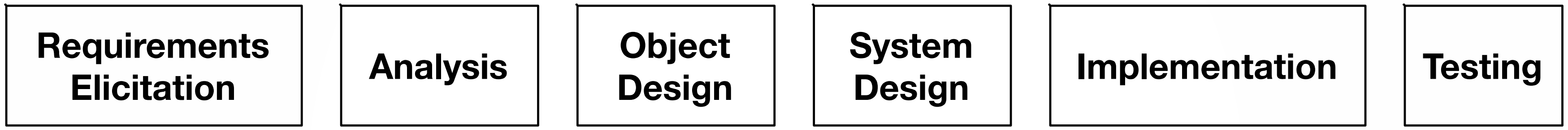
Software Life cycle: previous definition

Software life cycle (software process):

“All ***activities*** and ***work products*** necessary for the development of a software system.”

Software Life cycle activities – previous lecture

We covered several different activities in the last lecture:



Software Life cycle: refined definition

Software life cycle (software process):

"All ***activities*** and ***work products*** necessary for the development of a software system."

Refined:

Software life cycle (software process):

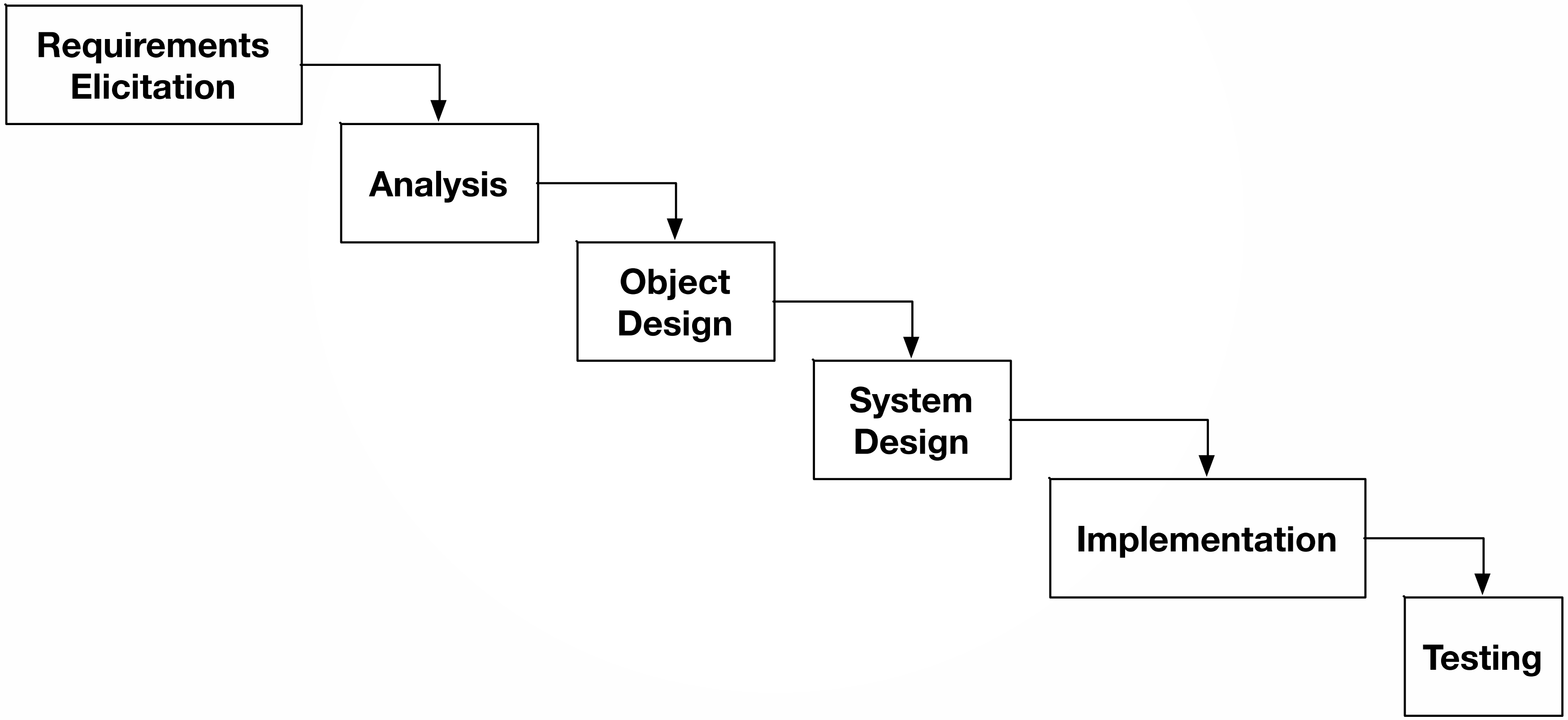
Set of **activities** and **work products** and their **relationships to each other to support** the development of a software system

What's new:

A software life cycle can contain activities and the **relationships between these activities** are important!

Software Life cycle activities - refined

Dependency arrows denote relationships between different activities



Software Life Cycle Questions

- ***Which activities*** should we select for the software project?
- What are the ***dependencies between activities***?
- How should we ***schedule the activities***?

Session #4 – Detailed Outline

Model based Software Development

- Software Lifecycle definition - redefined
- **Software Lifecycle tailoring**
- Software Lifecycle model definition
  - Examples
- Unified Modeling Language
  - Use case diagrams
  - Class diagrams
  - Communication diagrams
  - Component diagrams
  - Statechart diagrams
  - Deployment diagrams

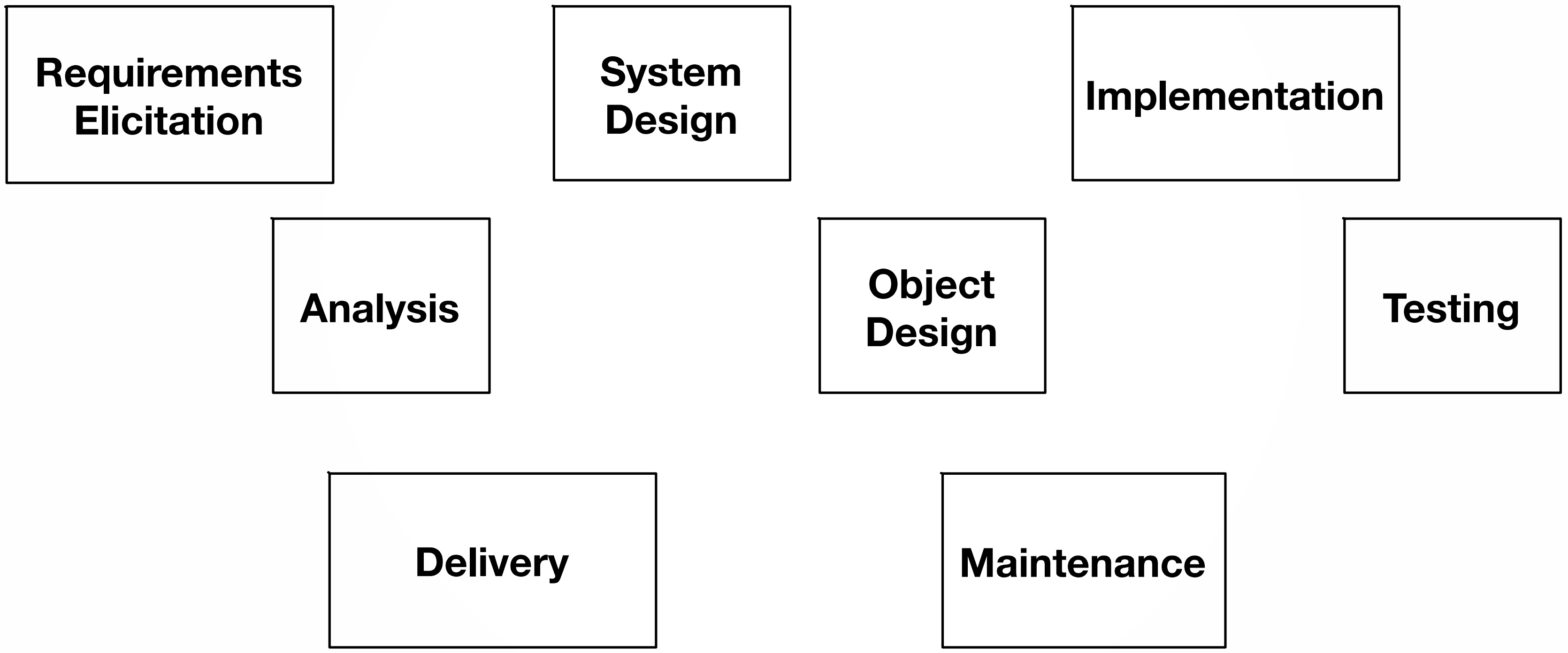
Software Life cycle activities – Tailoring

There is no “one size fits all” software lifecycle that works for all possible software engineering projects

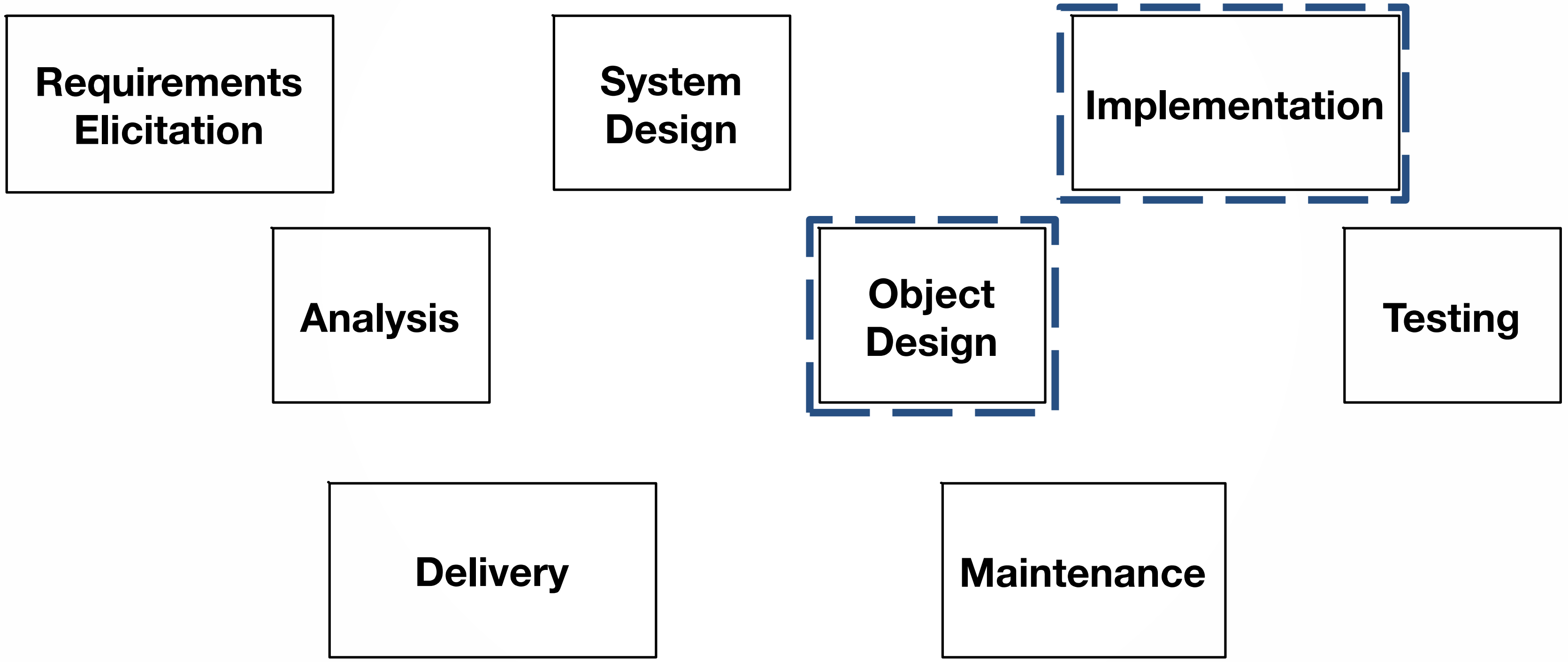
Tailoring: Adjusting lifecycle activities to fit a project

- Naming: Adjusting the naming of activities
- Adding/Cutting: Adding/removing activities to the project
- Ordering: Defining the order the activities take place in.

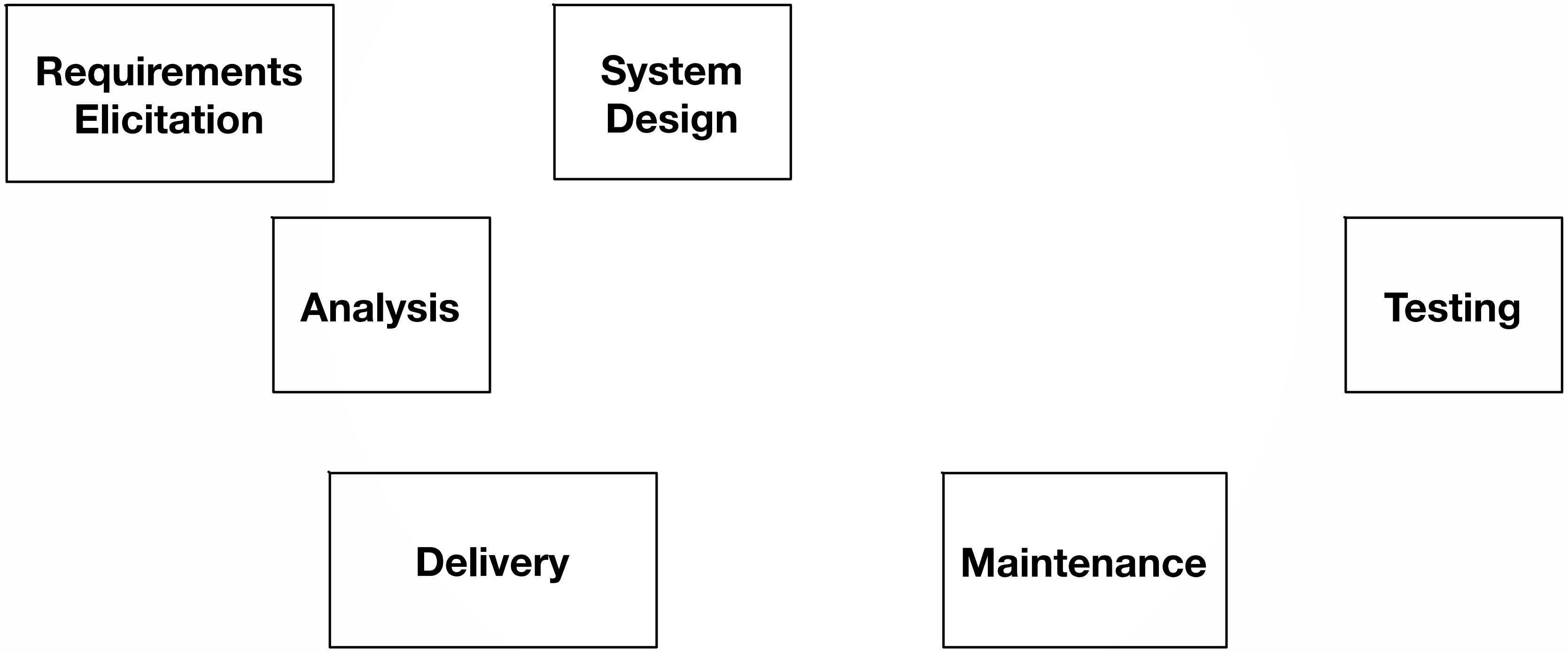
Software Life cycle activities – rename activities



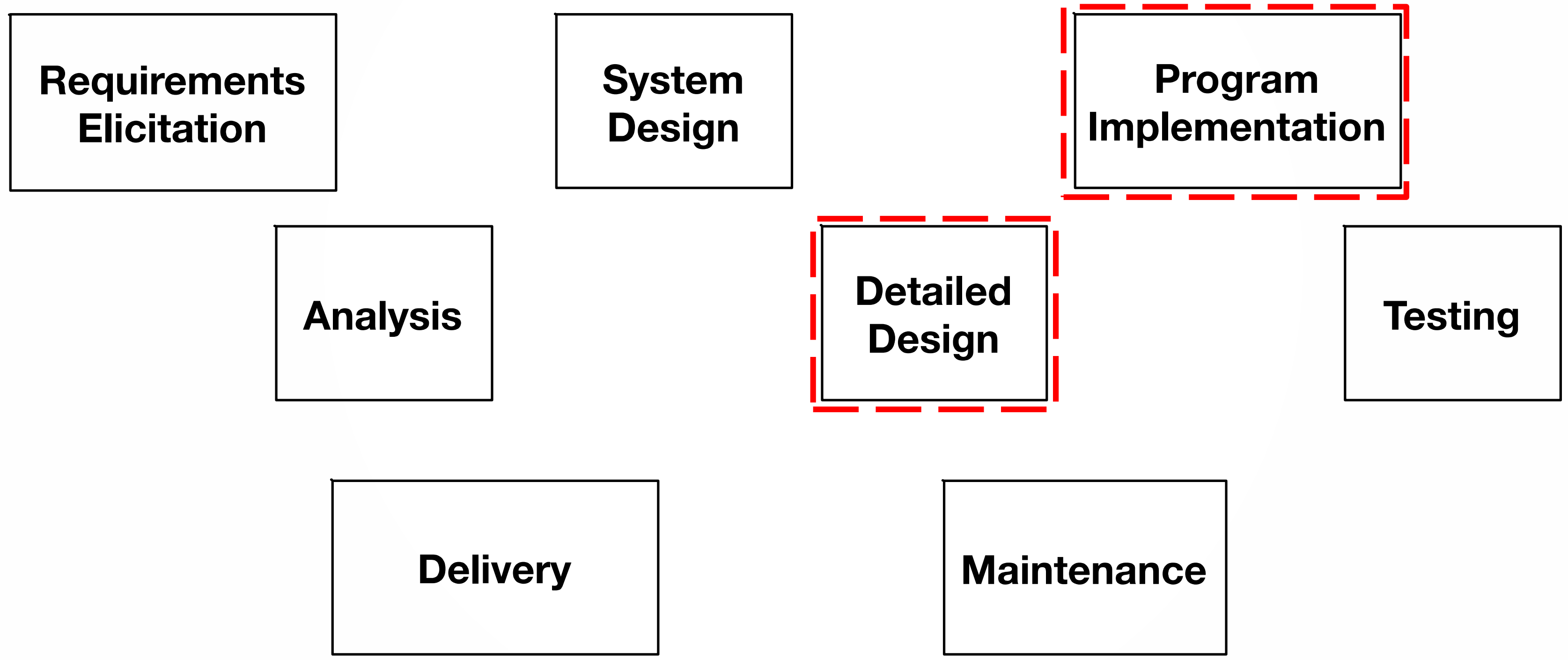
Software Life cycle activities – rename activities



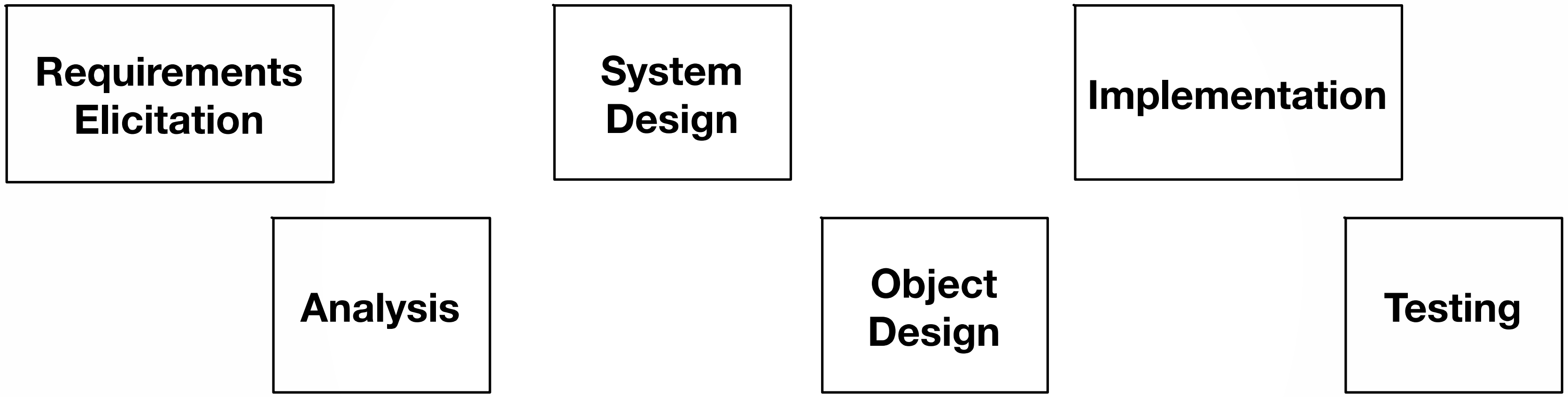
Software Life cycle activities – rename activities



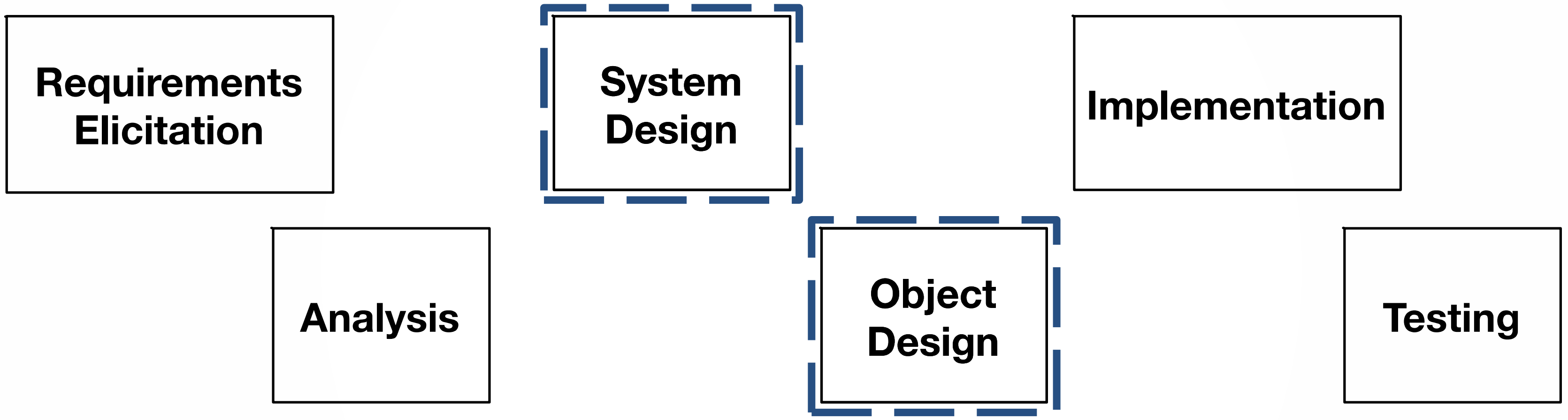
Software Life cycle activities – rename activities



Software Life cycle activities – add/cut activities



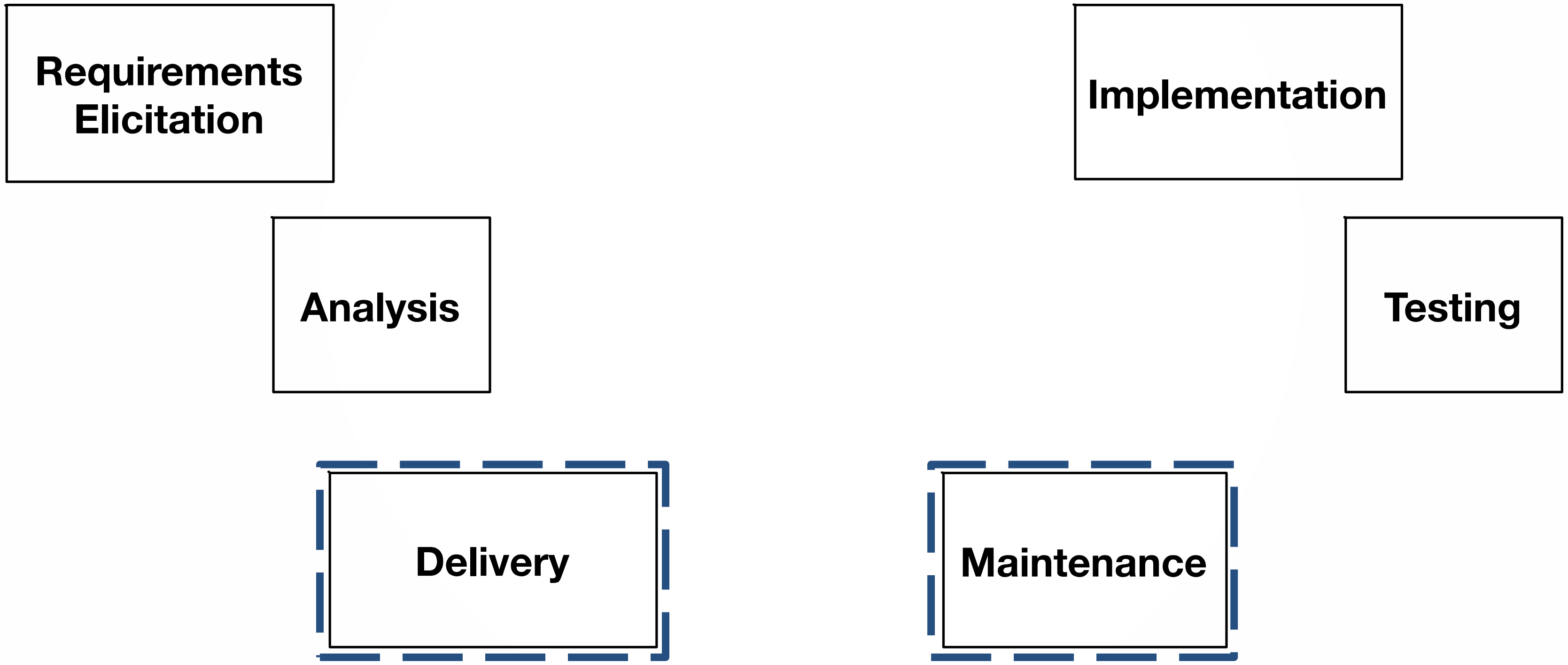
Software Life cycle activities – add/cut activities



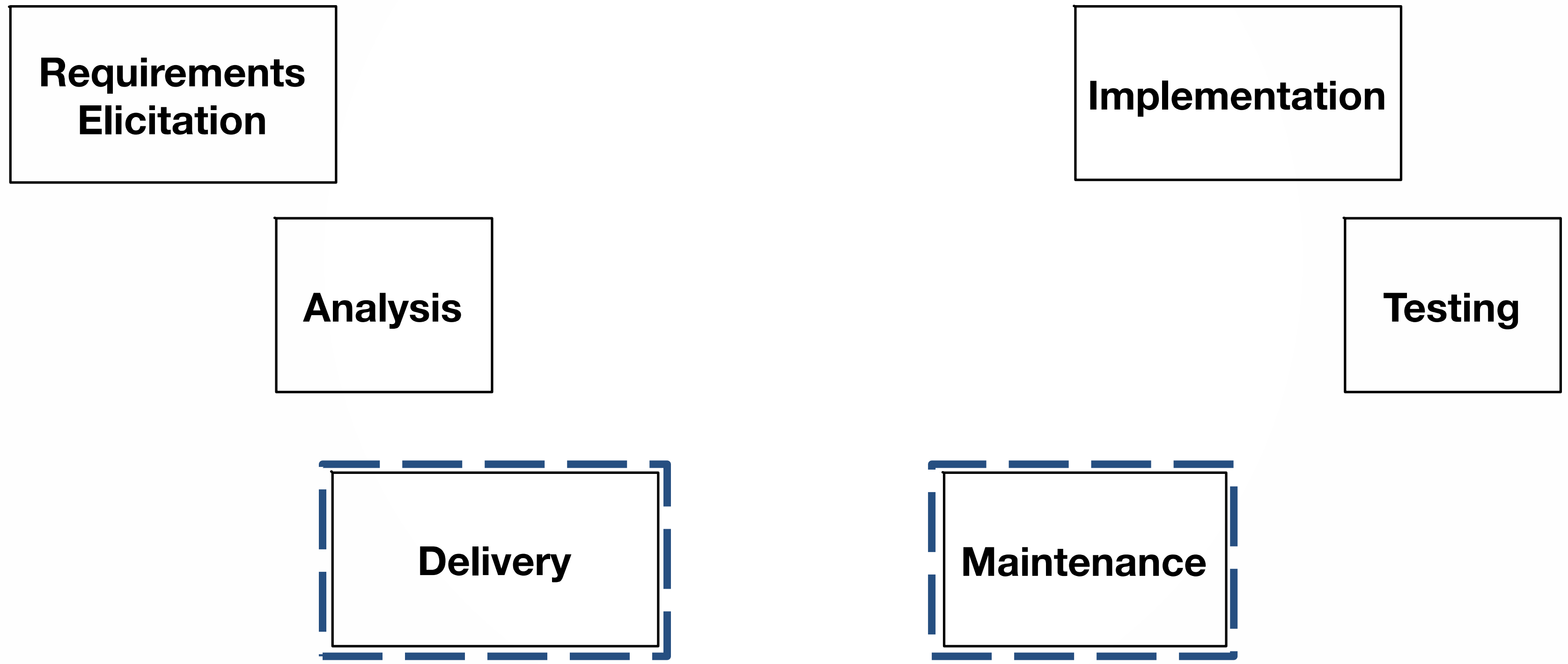
Software Life cycle activities – add/cut activities

Requirements Elicitation
Analysis
Implementation
Testing

Software Life cycle activities – add/cut activities



Software Life cycle activities – add/cut activities



Software Life cycle – Extreme tailoring

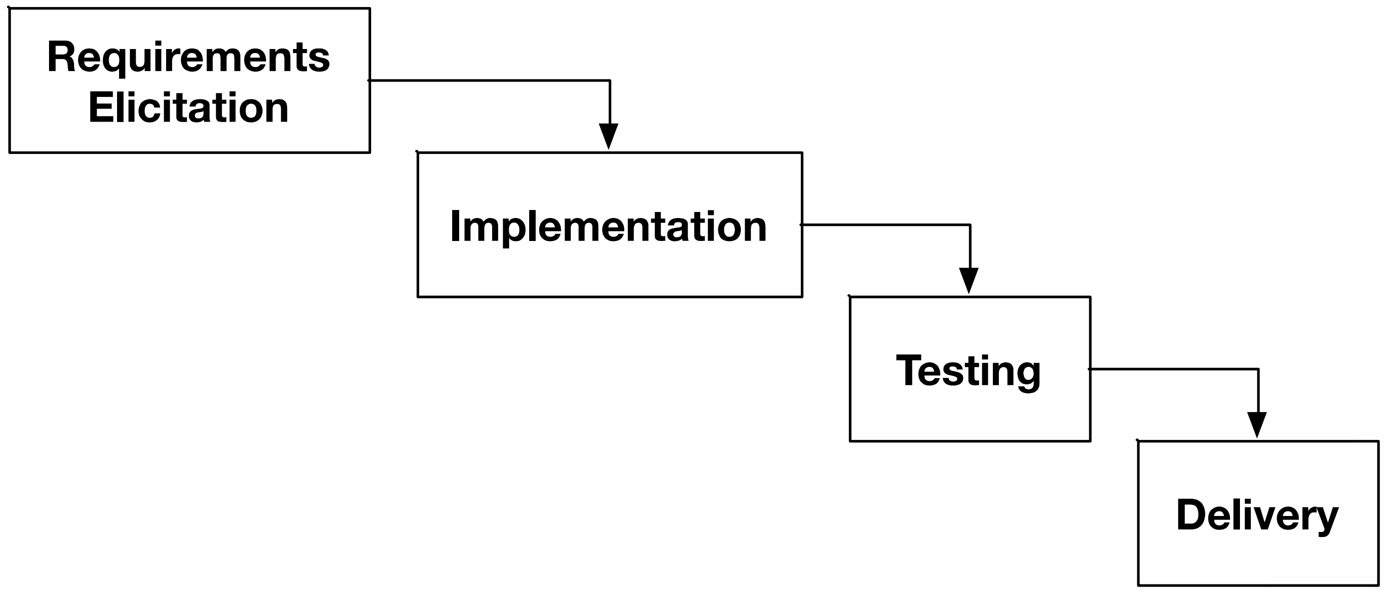
A Software Life cycle can contain only one single activity

Implementation

The hacker approach ☺.

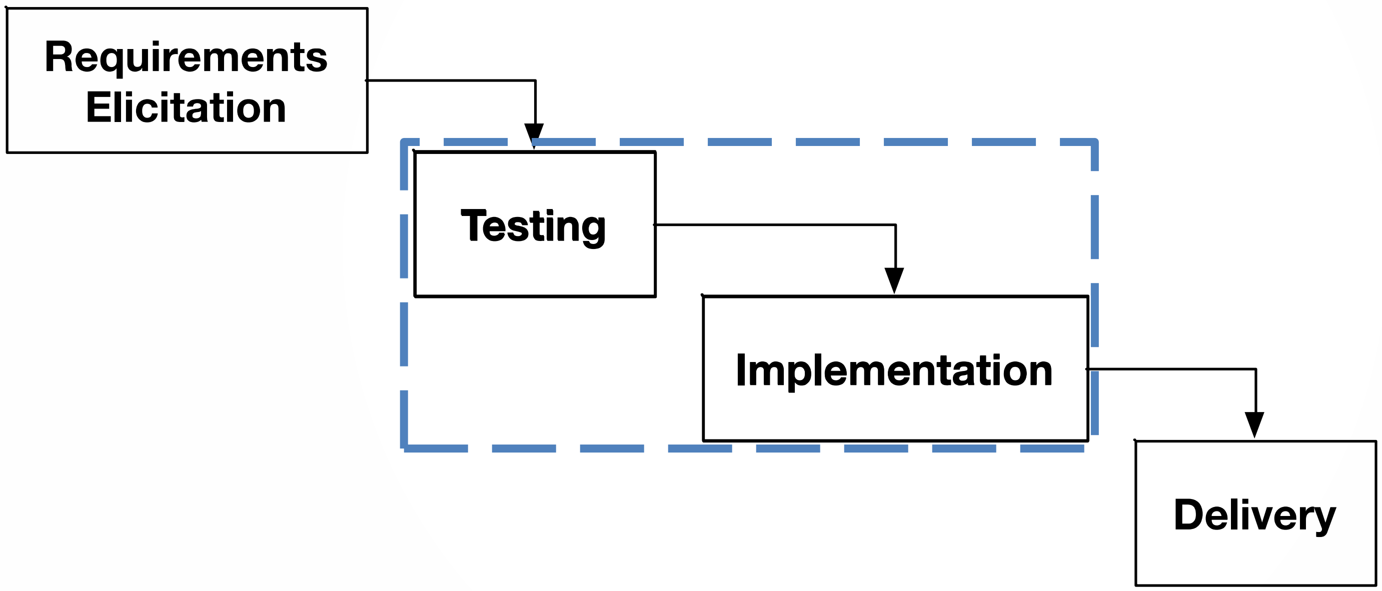
Software Life cycle activities - reorder

Usually we do Implementation before Testing:



Software Life cycle activities - reorder

Can we also do Testing before Implementation?!

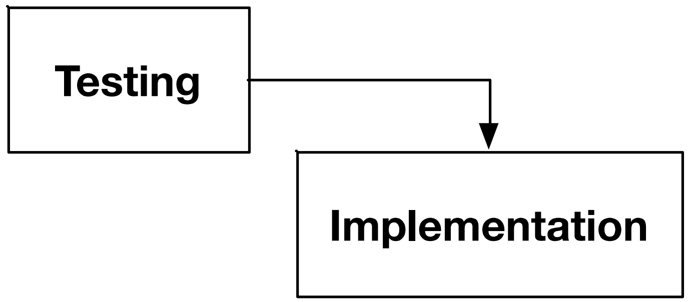


Software Life cycle – with two activities

Yes we can this process is called:

Test-driven Development

An Example for such a process is: Extreme Programming (XP)



Session #4 – Detailed Outline

Model based Software Development

- Software Lifecycle definition - redefined
- Software Lifecycle tailoring
- **Software Lifecycle model definition**
  - Examples
- Unified Modeling Language
  - Use case diagrams
  - Class diagrams
  - Communication diagrams
  - Component diagrams
  - Statechart diagrams
  - Deployment diagrams

Software Life cycle models

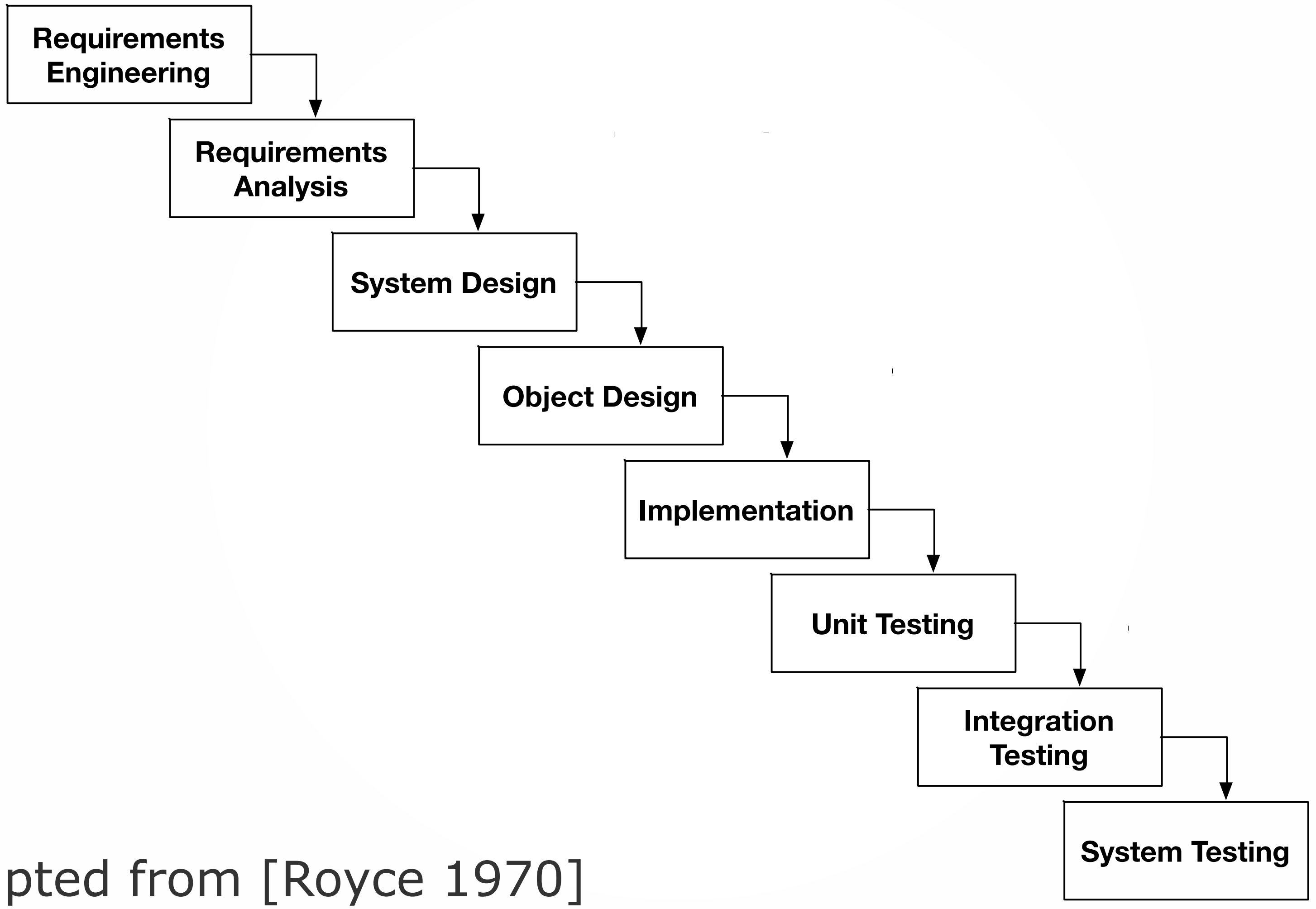
Software life cycle model:

- An abstraction that *represents* a software life cycle for the purpose of understanding, monitoring, or controlling the development of a software system.

- Examples of different model types:

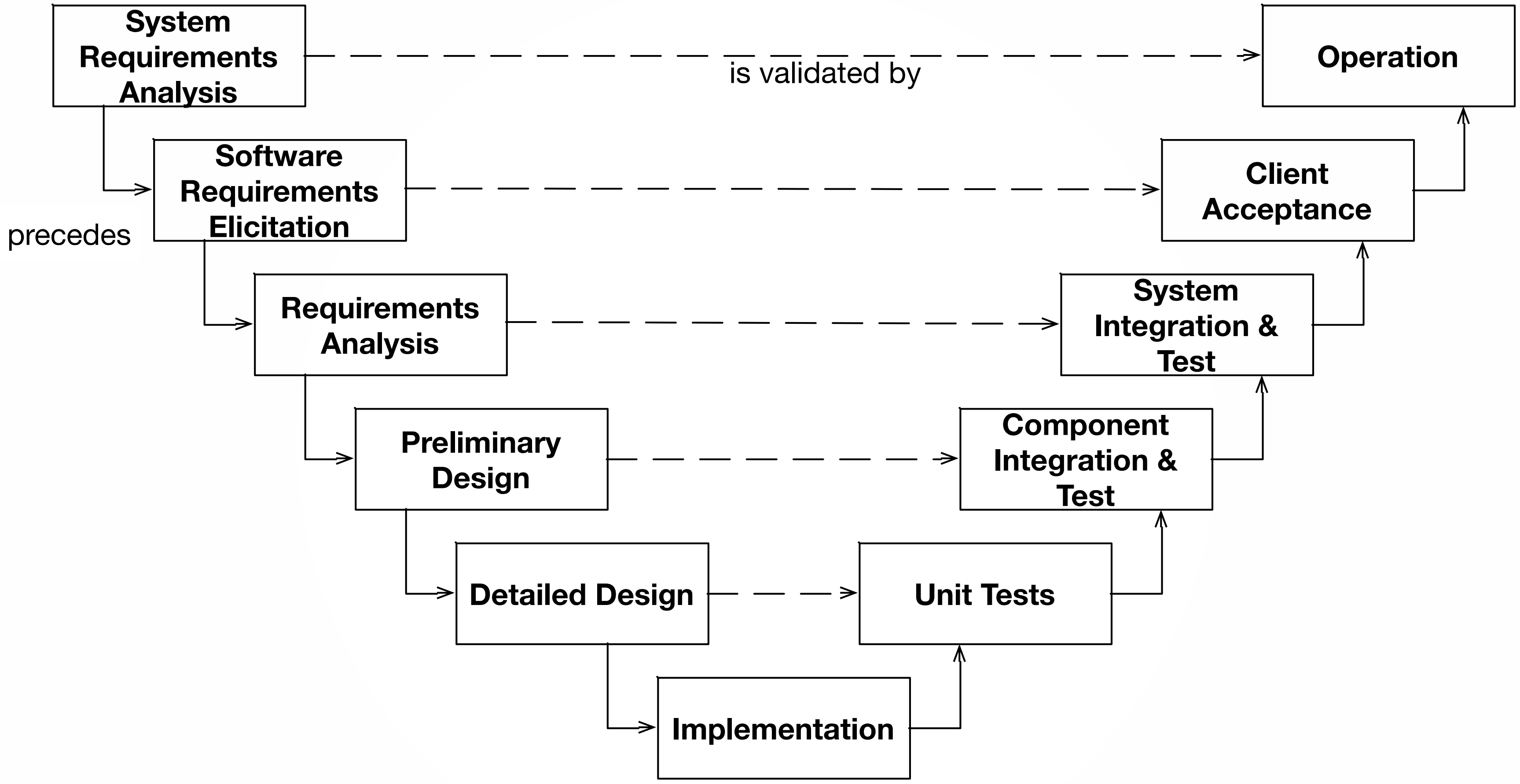
– **Linear models:**	Waterfall model , V model
– **Iterative models:**	Spiral model
– **Agile models:**	Extreme Programming (XP), Scrum

Waterfall Model with dependencies



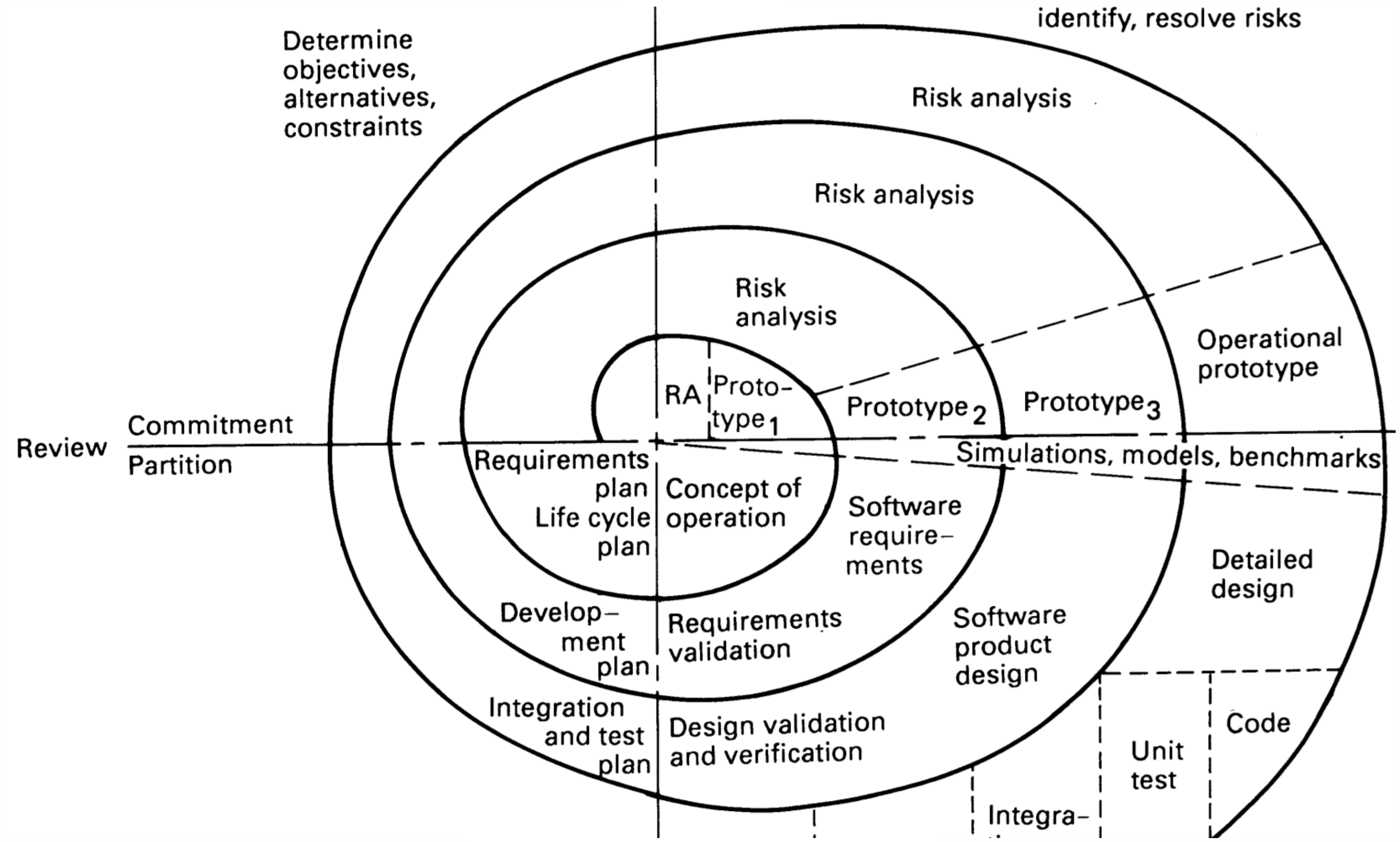
adapted from [Royce 1970]

V Model with dependencies



adapted from [Jensen & Tories, 1979]

Spiral Model



Software Life cycle / model summary:

Software Life cycle

- Activities, work products and their relations

Software life cycle model

- Abstraction of a Software Lifecycle for the purpose of:
- Understanding, Modeling and Controlling a Software development

What did we learn?:

We can use models to describe the Software Life cycle and its activities and dependencies.

We can also use models to describe a system to be built!

Session #4 – Detailed Outline

Model based Software Development

- Software Lifecycle definition - redefined
- Software Lifecycle tailoring
- Software Lifecycle model definition
  - Examples
- **Unified Modeling Language**
  - Use case diagrams
  - Class diagrams
  - Communication diagrams
  - Component diagrams
  - Statechart diagrams
  - Deployment diagrams

Unified Modeling Language (UML)

Using models to describe a software system to be built:

UML is a modeling language

- **Graphical notation**
- For documenting specification in **analysis** and **design**
- Model Driven Development (MDD),
  Code generation from the UML-Model

Importance

- Recommended OMG (Object Management Group) standard notation
- **De facto standard** in industrial software developments.

http://www.uml.org/

Creating a System model using UML

A System model is composed of:

Object model: What is the structure of the system?

- Class diagrams, Deployment diagrams, Component diagrams

Functional model: What are the functions of the system

- Scenarios, Use case diagrams

Dynamic model: How does the system react to external events?

- Communication diagrams, State chart diagrams, Sequence diagrams, activity diagrams

We use UML to describe the System model

Creating a System model using UML

A System model is composed of:

Object model: What is the structure of the system?

- **Class diagrams, Deployment diagrams, Component diagrams**

Functional model: What are the functions of the system

- Scenarios, **Use case diagrams**

Dynamic model: How does the system react to external events?

- **Communication diagrams, State chart diagrams**, Sequence diagrams, activity diagram

We use a <u>subset</u> of UML to describe the System model

UML diagrams

Use case diagrams

Requirements Elicitation	Analysis	System Design	Object Design	Implementation	Testing
🟩	🟩				

– Describe the functional behavior of a system as seen by the user

Class diagrams

Requirements Elicitation	Analysis	System Design	Object Design	Implementation	Testing
	🟩		🟩		

– Describe the static structure of the system: objects with attributes, operations and their associations

Communication diagrams

Requirements Elicitation	Analysis	System Design	Object Design	Implementation	Testing
	🟩		🟩		

– Describe the interaction between different objects by using method invocation

Component diagrams

Requirements Elicitation	Analysis	System Design	Object Design	Implementation	Testing
		🟩	🟩	🟩	

– Describe dependencies between components at design time, compilation time and runtime

Deployment diagrams

Requirements Elicitation	Analysis	System Design	Object Design	Implementation	Testing
		🟩			

– Describe the distribution of components on nodes

State chart diagrams

Requirements Elicitation	Analysis	System Design	Object Design	Implementation	Testing
			🟩		

– Describe the dynamic behavior of a single, individual object

Use case diagrams

Requirements Elicitation	Analysis	System Design	Object Design	Implementation	Testing

Use case diagrams

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

Class diagrams

- Describe the static structure of the system: objects with attributes, operations and their associations

Communication diagrams

- Describe the interaction between different objects by using method invocation

Component diagrams

- Describe dependencies between components at design time, compilation time and runtime

Deployment diagrams

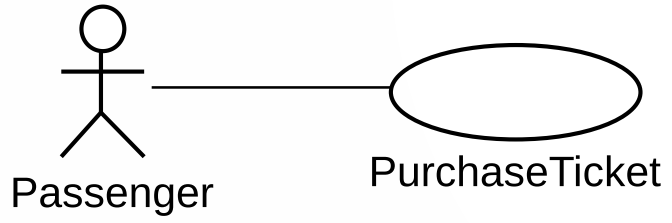
- Describe the distribution of components on nodes

State chart diagrams

- Describe the dynamic behavior of a single, individual object

Use case diagrams

- Used during requirements elicitation and analysis to represent the behavior that is visible from the outside of the system



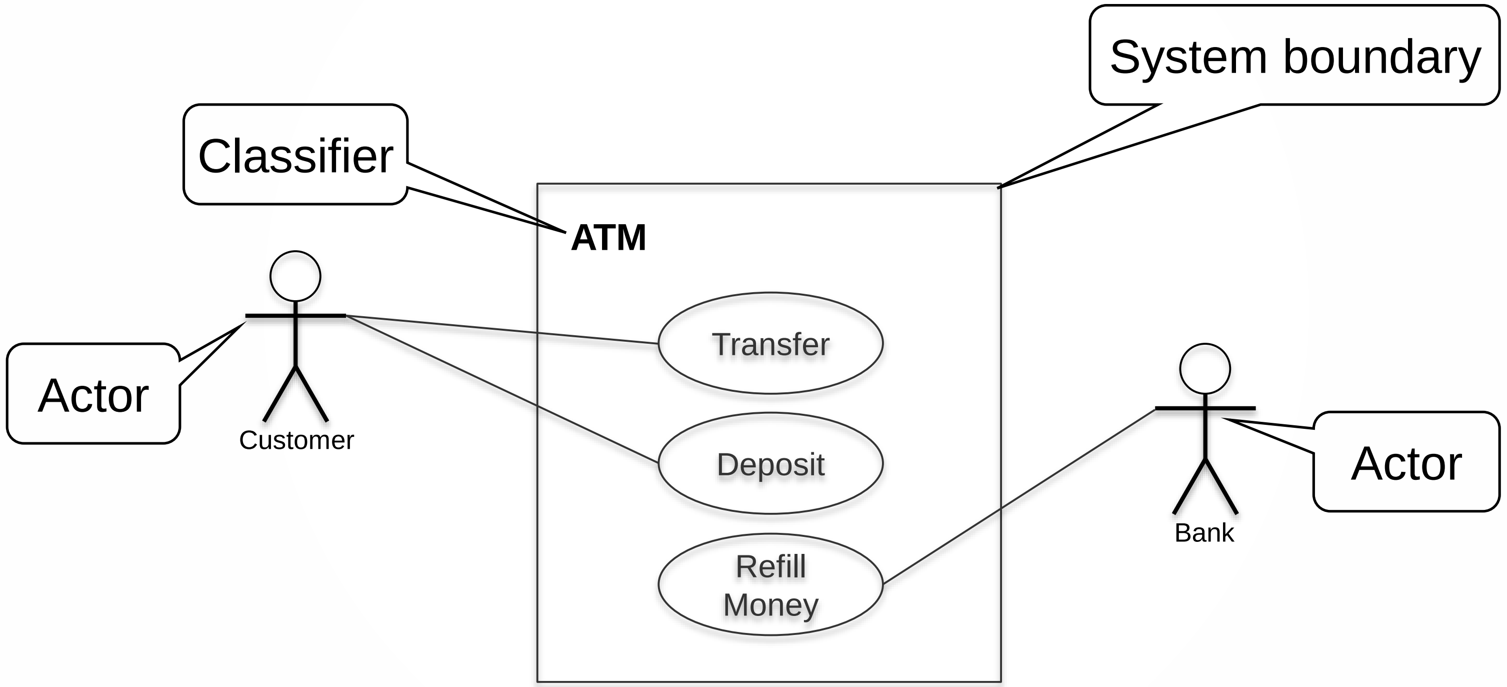
A *use case* represents a functionality provided by the system

Use case are associated with actors

An *actor* represents a specific type of user of the system (sometimes also called a role)

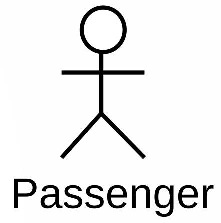
Use case model:
A graph of use cases and actors that describes the functionality of the system.

Use case diagrams - example



Actors

- An actor is a model for an external entity which interacts or communicates with the system:
  - User
  - External system (Another system)
  - Physical environment (e.g. Weather)



- An actor has a unique name and an optional description
- Examples:
  - Passenger: a person in the train
  - GPS satellite: An external system that provides a navigation system with GPS information.

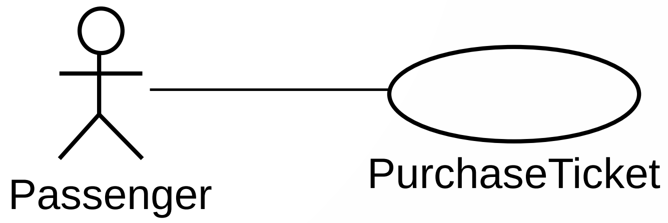
Name

Optional Description

Use Case

- A use case represents a class of functionality provided by the system
- Use cases can be described textually, with a focus on the event flow between the actor and the system
- A textual use case description consists of 6 parts:

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



Textual Use Case description: Example

- Name: purchase ticket
- Participating actors: Passenger
- Entry conditions:
  - The passenger stands in front of the ticket distributor
  - The passenger has sufficient money to purchase ticket
- Exit conditions:
  - The passenger has the ticket
- Flow of events:
  - The passenger selects the number of zones to be traveled
  - The ticket distributor displays the amount due
  - The passenger inserts at least the amount due
  - The ticket distributor returns change
  - The ticket distributor issues the ticket
- Special requirements:
  - The ticket distributor is connected to a power source

Uses Cases can be related with other Use Cases

- We distinguish two types of relationships:

- Extends Relationship
  - To model rarely invoked use cases or exceptional functionality

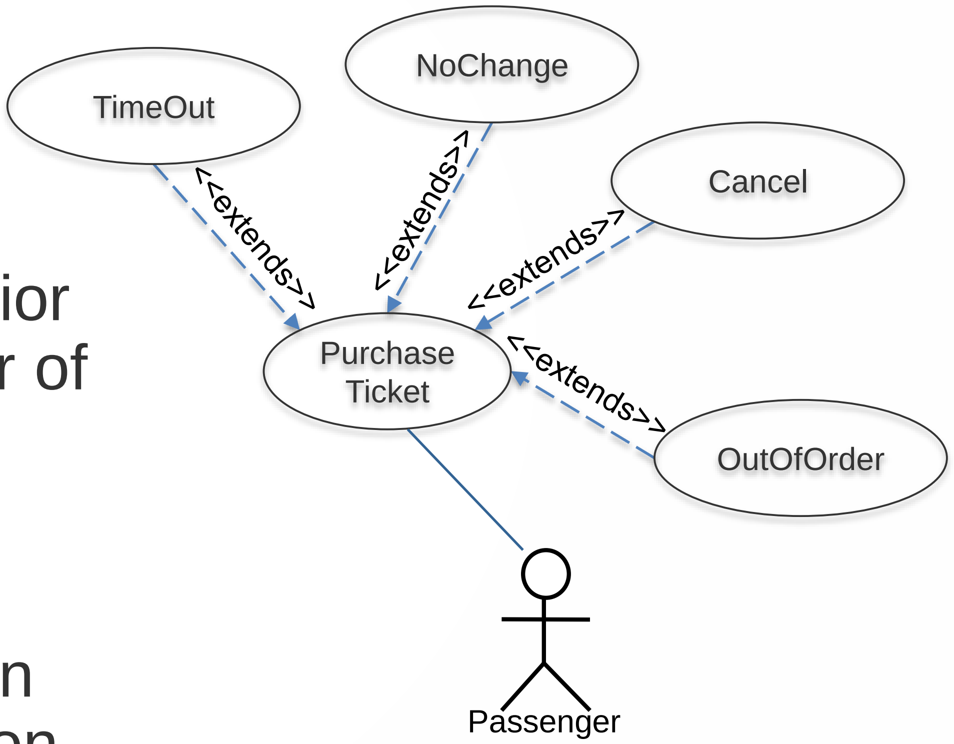
- Includes Relationship
  - To model functional behavior that is common to more than one use case.

The Extends Relationship

- The base use case describes basic functionality, it is meaningful by itself

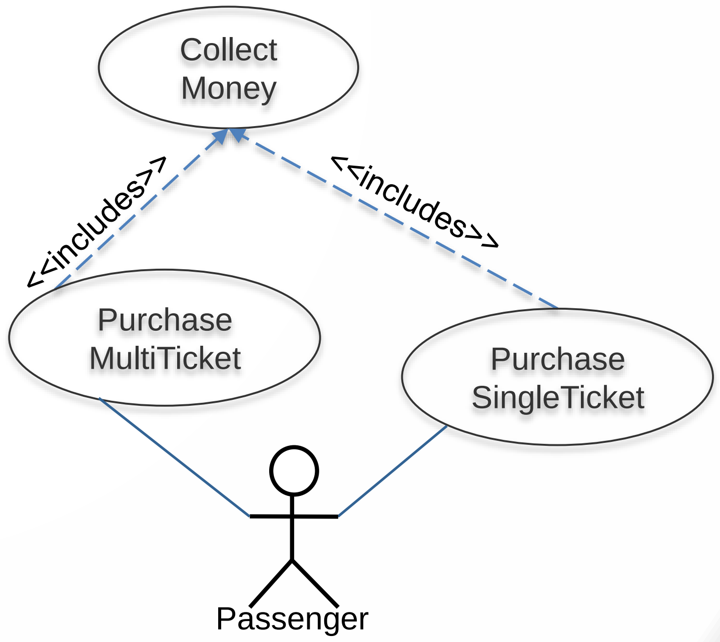
- The extension use case describes additional behavior that augments the behavior of the base case. It is not meaningful on its own.

- The direction of the arrow in the <<extends>> association points to the base use case

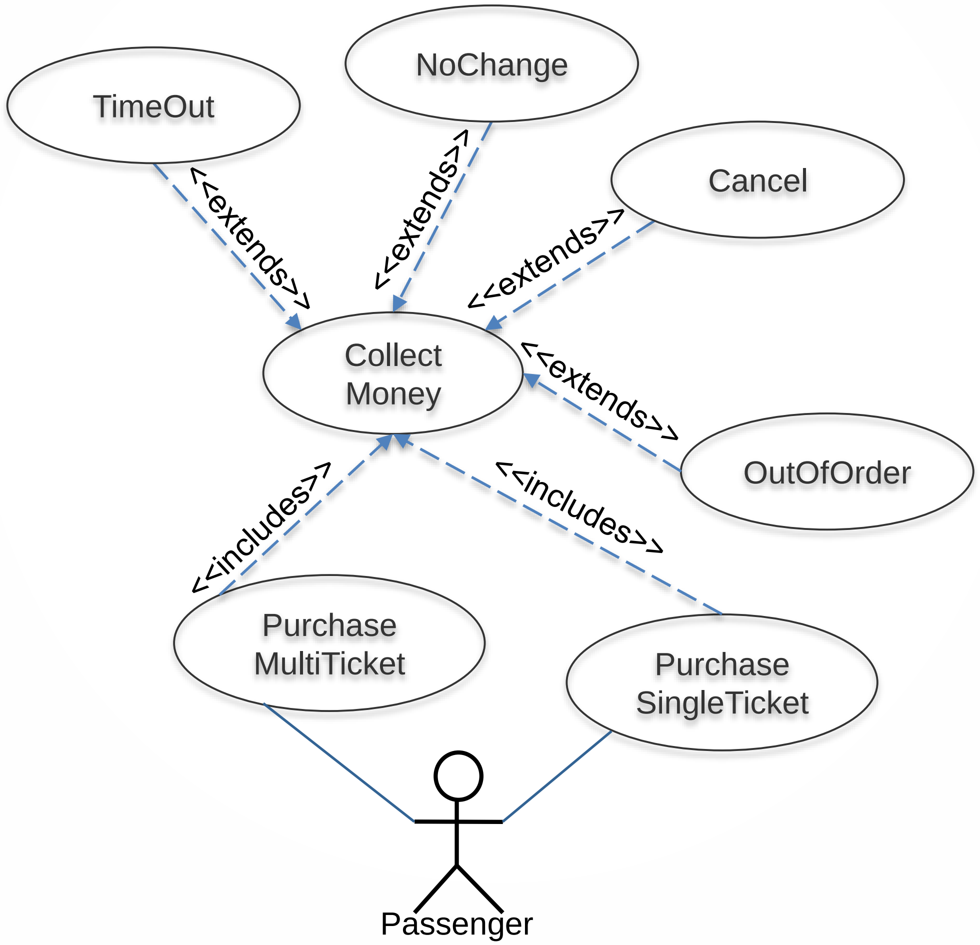


The Includes Relationship

- Assume we have several use cases that share some functionality
- An inclusion use case represents functionality needed in more than one base use case
- Arrows in <<includes>> associations point from the base use case to the inclusion use case.



Typical Use case contains both relations



Exercise #4.1

Draw a use case diagram for a ticket distributor for a train system.

- The system includes two actors:
  - a traveler, who purchases different types of tickets, and
  - a central computer system, which maintains a reference database for the tariff.

- Use cases should include: BuyOneWayTicket, BuyWeeklyCard, BuyMonthlyCard, UpdateTariff.
- Also include the following exceptional cases:
  - Time-Out (i.e., traveler took too long to insert the right amount)
  - TransactionAborted (i.e., traveler selected the cancel button)
  - DistributorOutOfChange
  - DistributorOutOfPaper

Class diagrams

Use case diagrams

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

Requirements Elicitation	Analysis	System Design	Object Design	Implementation	Testing

Class diagrams

- Describe the static structure of the system: objects with attributes, operations and their associations

Communication diagrams

- Describe the interaction between different objects by using method invocation

Component diagrams

- Describe dependencies between components at design time, compilation time and runtime

Deployment diagrams

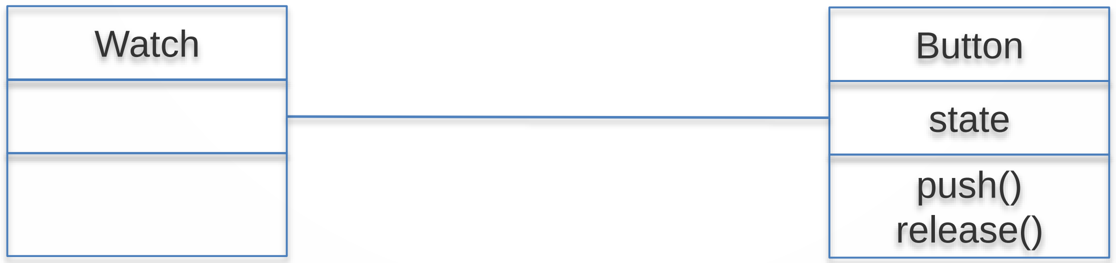
- Describe the distribution of components on nodes

State chart diagrams

- Describe the dynamic behavior of a single, individual object

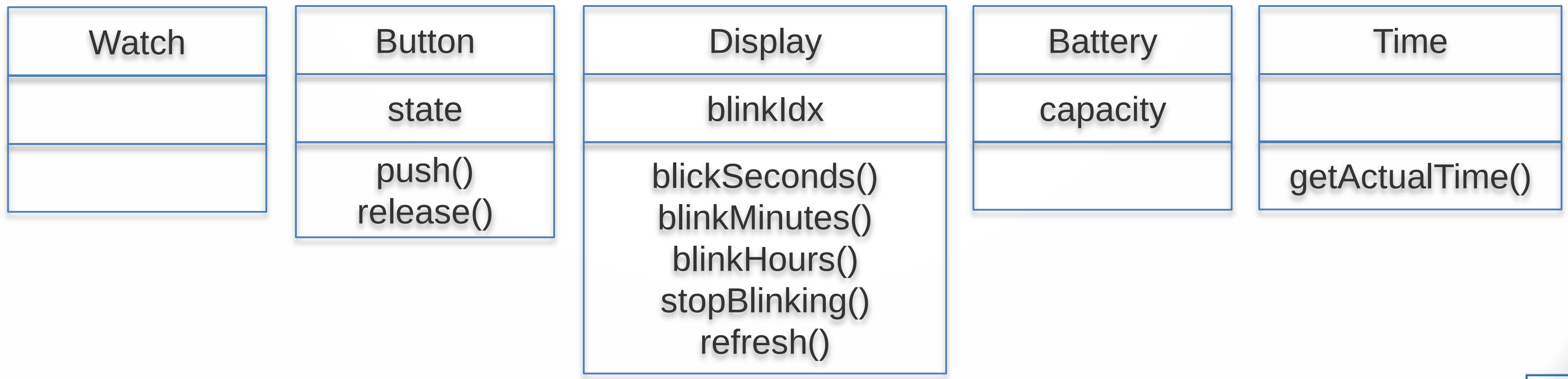
Class Diagrams

- Class diagrams represent the structure of the system
- We use them in many modeling situations:
  - During requirements elicitation to model system internal objects
  - During object design to specify the detailed behavior and the attributes of classes.



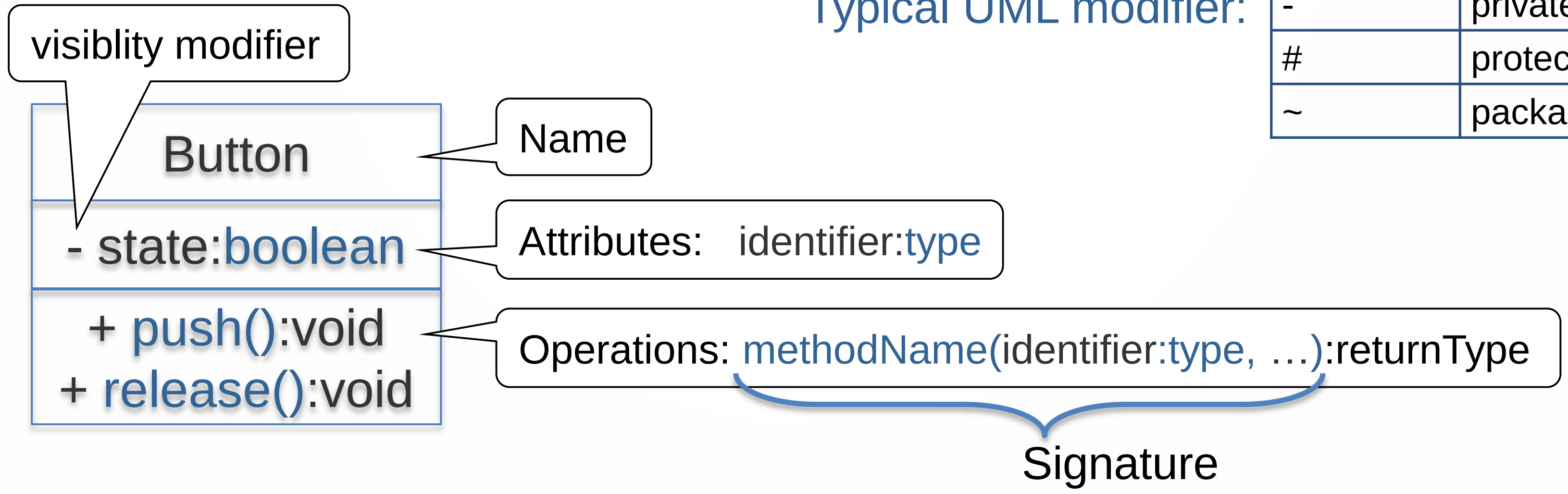
Classes in Analysis

- Each class has Name
- Each class has attributes (State)
- Each class has operations (Behavior)
- The class name is the only mandatory information



Classes in Object Design

- Each class has a name
- Each attribute has a **type**
- Each operation has a **signature**
- Each attribute and operation has a **visibility** modifier



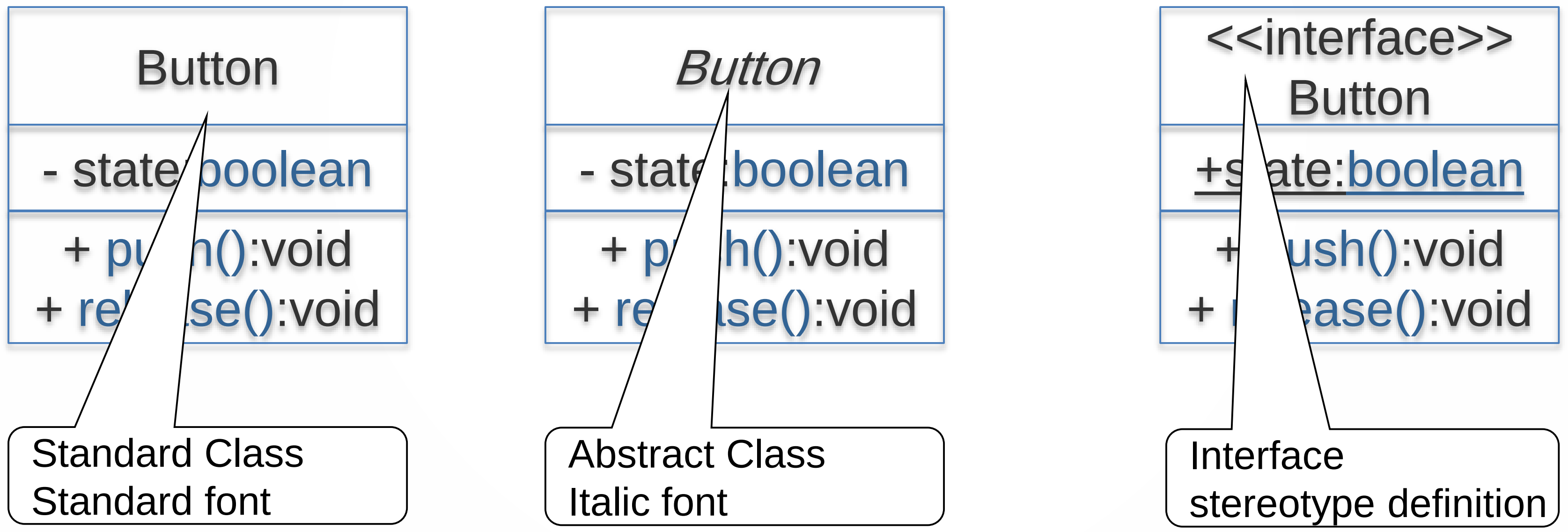
Typical UML modifier:

+	public
-	private
#	protected
~	package

Classes in Object Design

Classes vs. Interfaces vs. Abstract classes

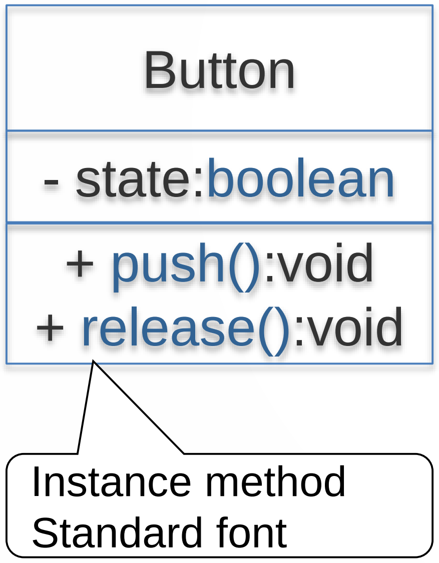
Class: **Abstract Class:** **Interface:**



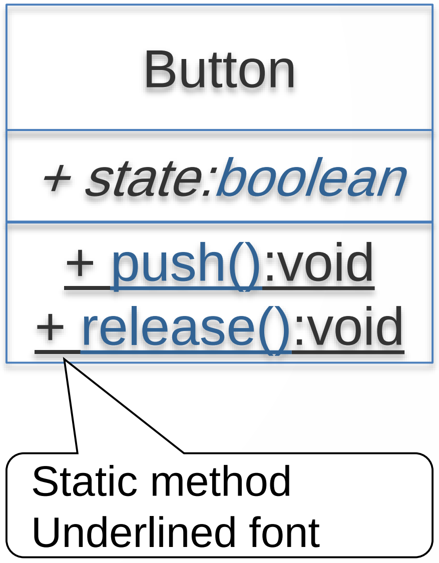
Classes in Object Design

Instance methods vs. Static methods

Class: Instance methods



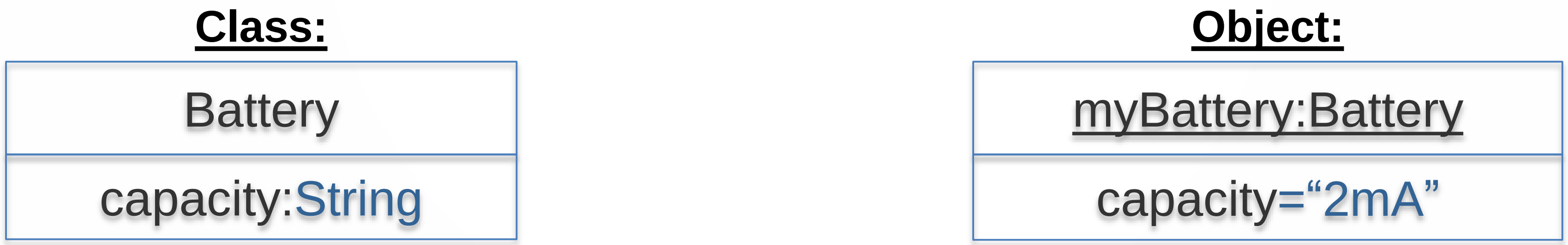
Class: Static methods



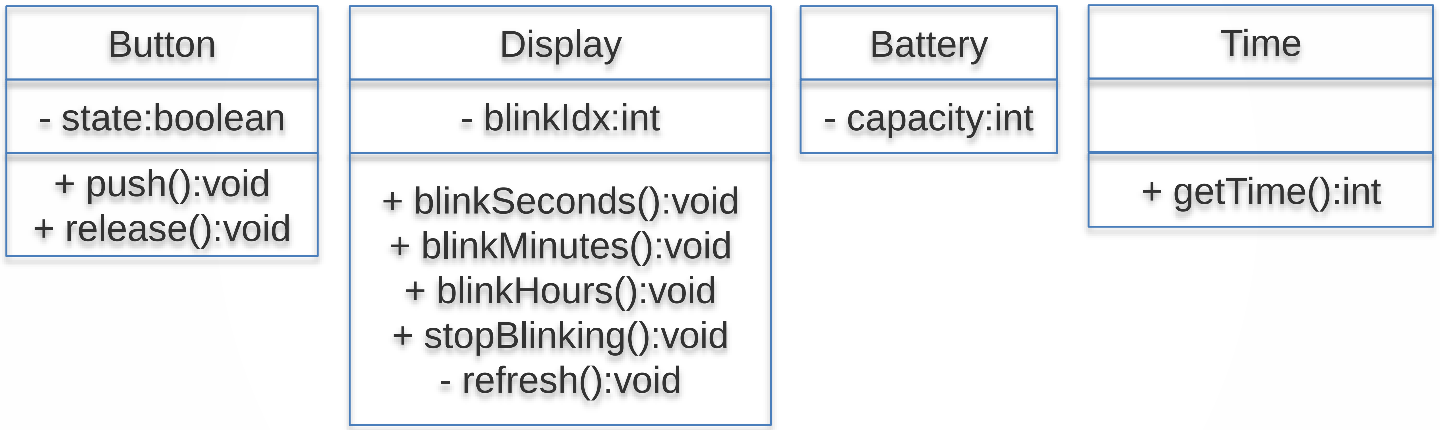
Instances in Object Design

Classes can be instantiated:

- They are called Instances and can also be modeled with Class diagrams
- The name of an instance is underlined
- The attributes are represented with their values



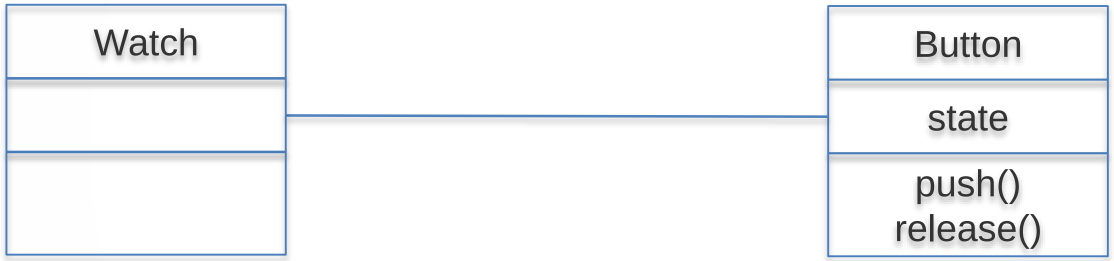
Example of Classes in Object Design



Associations

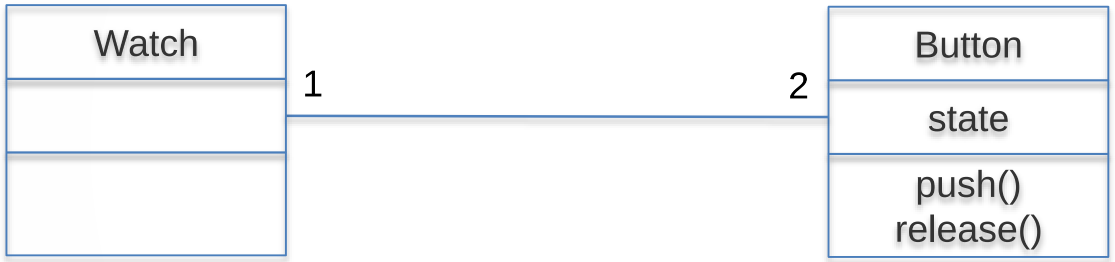
Associations denote relationships between classes

- Each association has two association ends



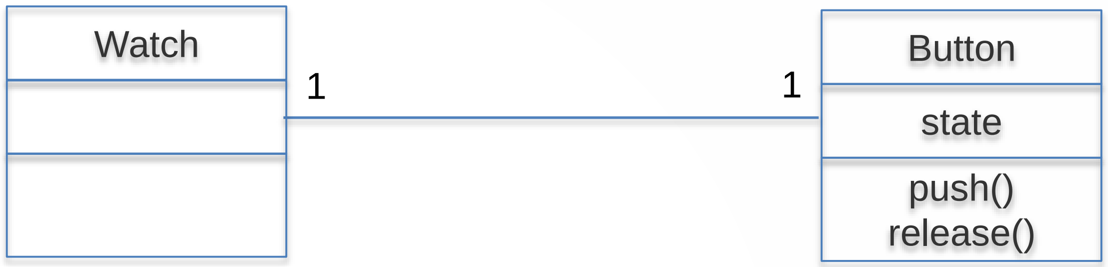
Multiplicity

Multiplicity of an association end denotes how many objects the instance of a class can reference.

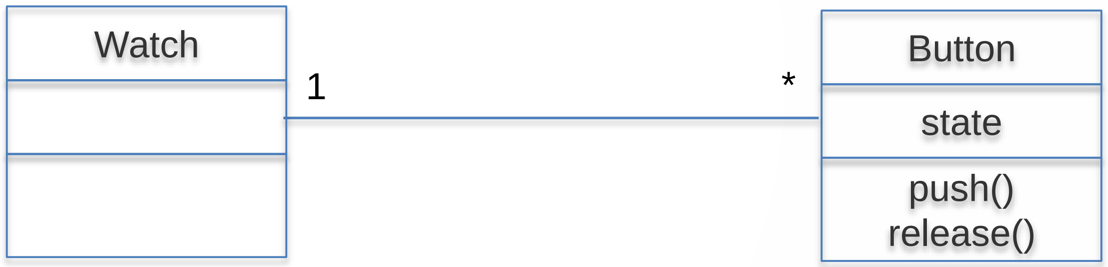


Associations

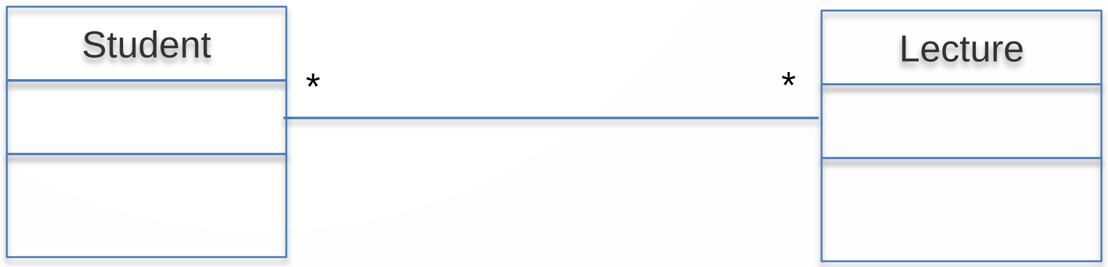
- 1-to-1:



- 1-to-many:

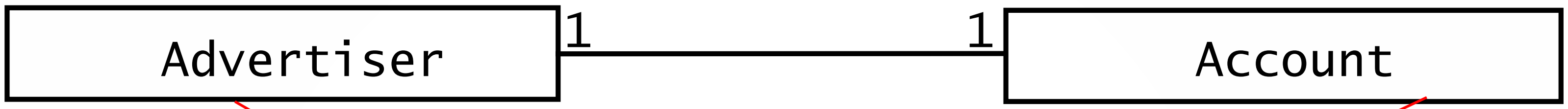


- many-to-many:



Mapping Unidirectional 1-to-1 Associations

Object design model before transformation:

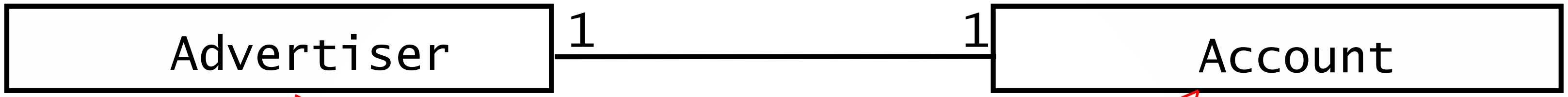


Source code after transformation:

```
public class Advertiser {
   private Account account;
   public Advertiser() {
      account = new Account();
   }
   public Account getAccount() {
      return account;
   }
}
```

Mapping Bidirectional 1-to-1 Associations

Object design model before transformation:



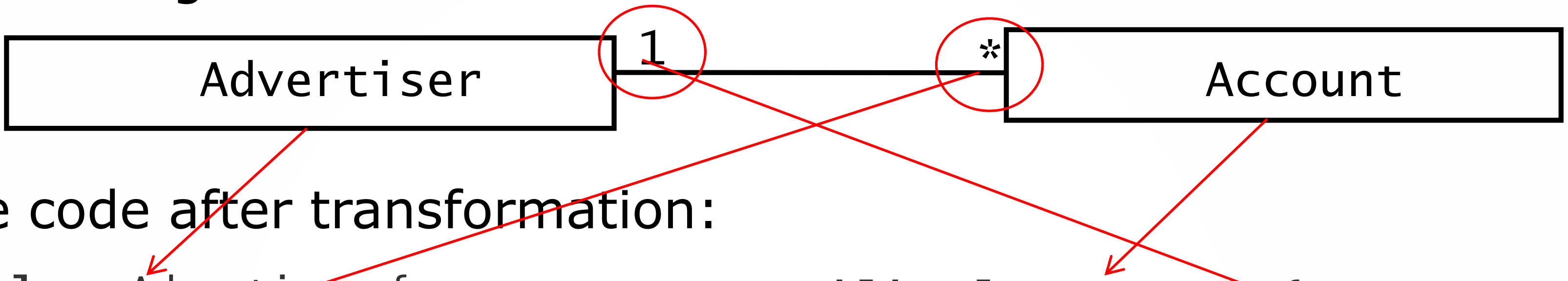
Source code after transformation:

```
public class Advertiser {
/* account is initialized
 * in the constructor and never
 * modified. */
   private Account account;
   public Advertiser() {
    account = new Account(this);
   }
   public Account getAccount() {
    return account;
   }
}
```

```
public class Account {
   /* owner is initialized
    * in the constructor and
    * never modified. */
   private Advertiser owner;
   public Account(Advertiser owner) {
    this.owner = owner;
   }
   public Advertiser getOwner() {
    return owner;
   }
}
```

Mapping Bidirectional 1-to-Many Associations

Object design model before transformation:



Source code after transformation:

```
public class Advertiser {
   private List accounts;
   public Advertiser() {
    accounts = new LinkedList();
   }
   public void addAccount(Account a) {
    accounts.add(a);
    a.setOwner(this);
   }
   public void removeAccount(Account a) {
    accounts.remove(a);
    a.setOwner(null);
   }
}
```

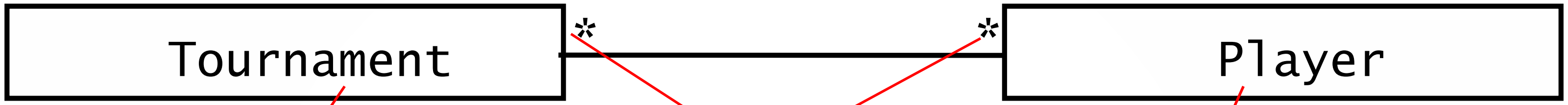
```
public class Account {
   private Advertiser owner;
   public void setOwner(Advertiser
   newOwner) {
     if (owner != newOwner) {
          Advertiser old = owner;
          owner = newOwner;
          if (newOwner != null)

     newOwner.addAccount(this);
          if (oldOwner != null)

     old.removeAccount(this);
     }
   }
}
```

Mapping Bidirectional Many-to-Many Associations

Object design model before transformation



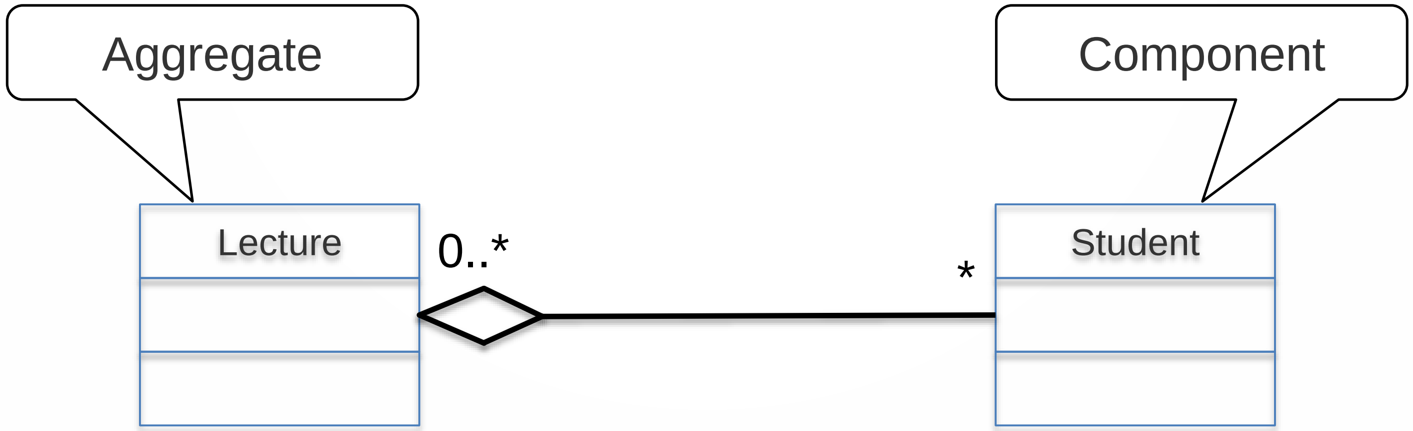
Source code after transformation

```
public class Tournament {
   private List players;
   public Tournament() {
       players = new ArrayList();
   }
   public void addPlayer(Player p)
   {
       if (!players.contains(p)) {
           players.add(p);
           p.addTournament(this);
       }
   }
}
```

```
public class Player {
   private List tournaments;
   public Player() {
       tournaments = new
   ArrayList();
   }
   public void
   addTournament(Tournament t) {
       if (!tournaments.contains(t))
   {
        tournaments.add(t);
        t.addPlayer(this);
       }
   }
}
```

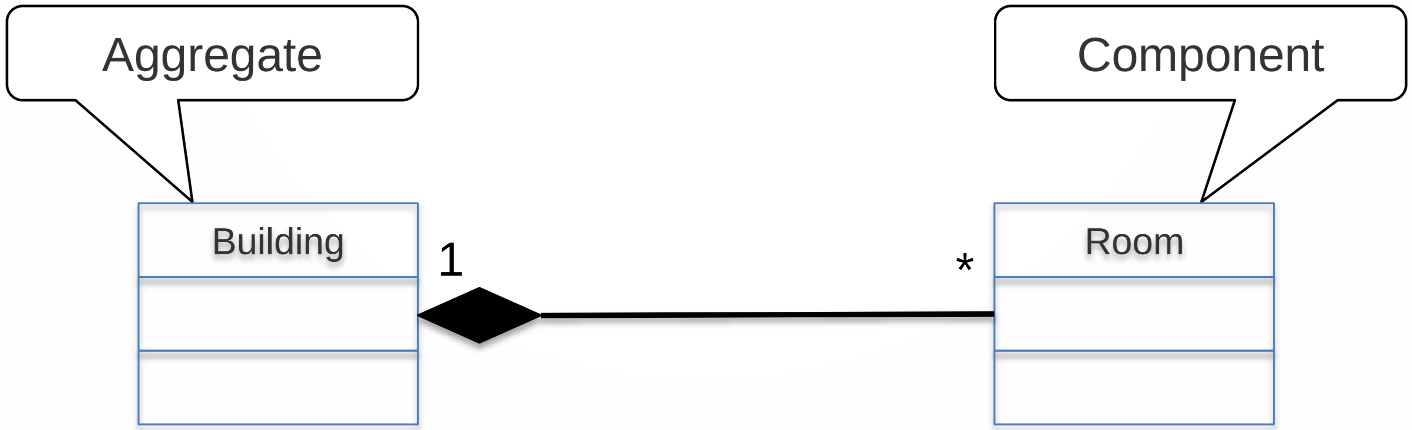
Aggregation

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



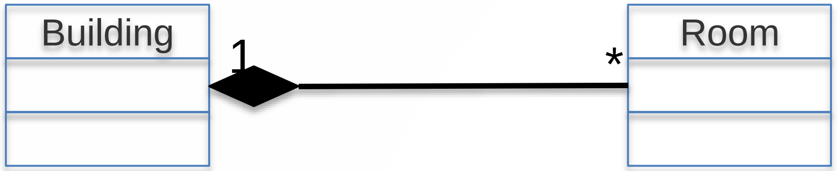
Composition

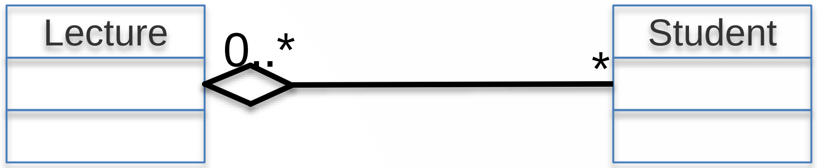
- Composition is a special form of aggregation
  - In UML diagrams it is drawn as a solid diamond
- It is used when the life time of the component instances are controlled by the aggregate
  - That is, the components (parts) do not exist on their own
  - “The aggregate controls/destroys the components”.



Aggregation vs. Composition

- Implementation difference e.g. Java:





Composition:

```
public class Building {

        private HashMap<String,Room> rooms;

        public Building()
        {
           rooms.put("01:07:60", new Room("01:07:60"));
           rooms.put("01:07:20", new Room("01:07:20"));

        }
        void cleanRoom(String roomName)
        {
           rooms.get(roomName).clean();
        }
}
```

Aggregation:

```
public class Lecture {

        private Collection<Student> students;

        void addStudent(Student s) {
                students.add(s);
        }

}
```

Exercise #4.2

Draw a class diagram representing a book defined by the following statement:

“A book is composed of a number of parts, which in turn are composed of a number of chapters. Chapters are composed of sections.”

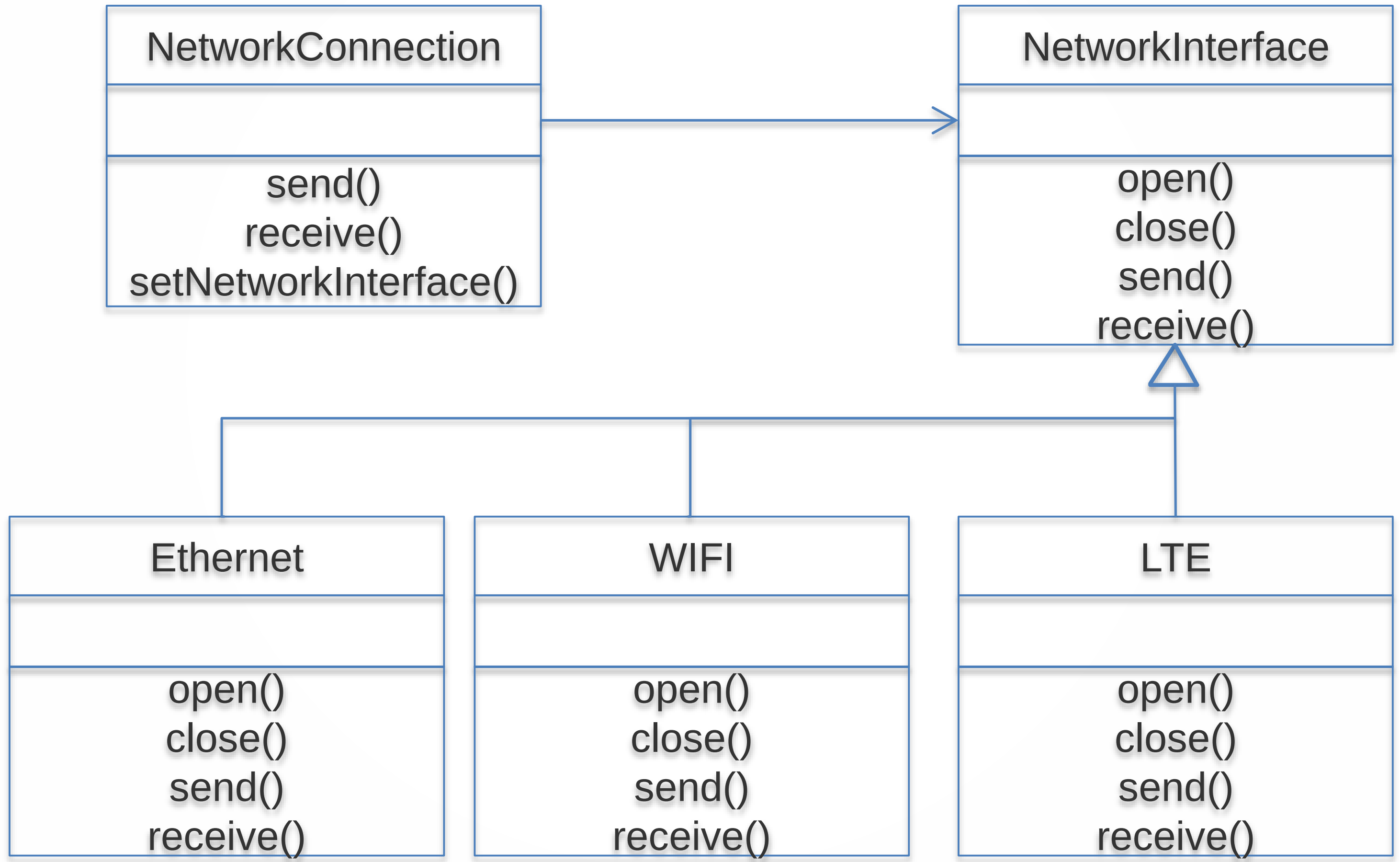
Focus only on classes and relationships.

Exercise #4.3

- Extend the class diagram to include the following attributes:

- a book includes a publisher, publication date, and an ISBN
- A part includes a title and a number
- a chapter includes a title, a number, and an abstract
- a section includes a title and a number

Inheritance in Analysis



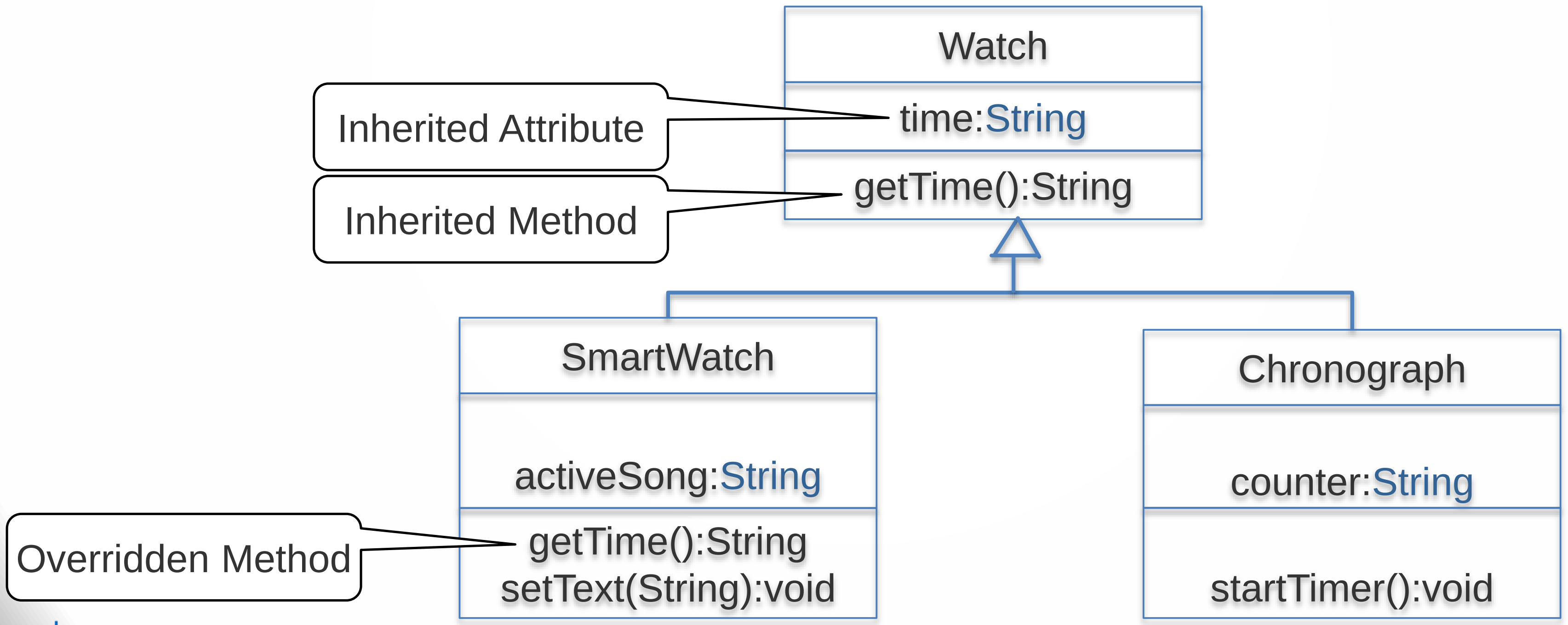
In my office: Local Wired Network

At home: Local Wireless Network

Traveling: Global Wireless Network

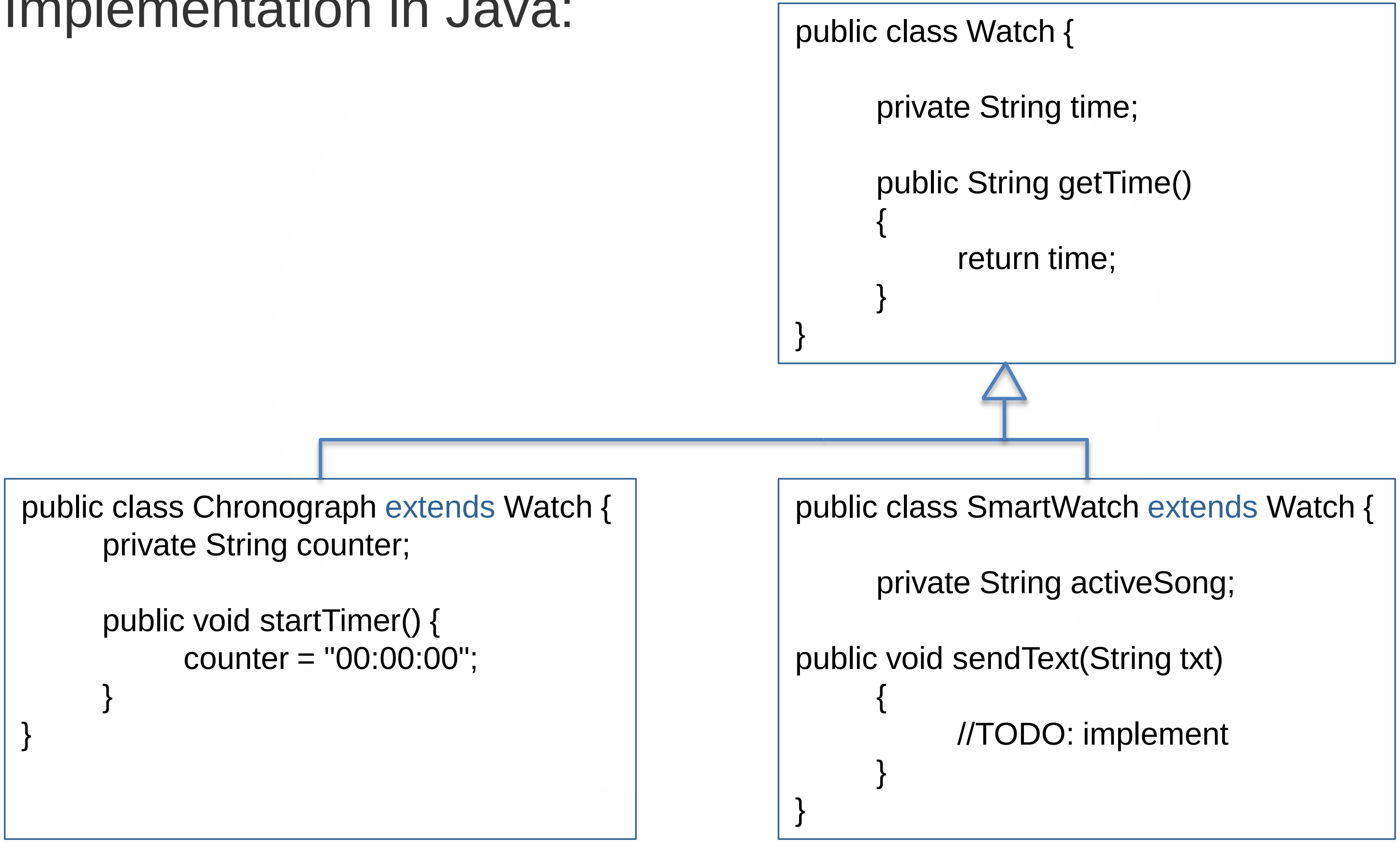
Inheritance in Object Design

- Inheritance allows the specialization and refinement of existing classes by using subclasses
  - In UML diagrams it is drawn as a triangle



Inheritance

Implementation in Java:



```
public class Watch {

        private String time;

        public String getTime()
        {
                return time;
        }
}
```

```
public class Chronograph extends Watch {
        private String counter;

        public void startTimer() {
                counter = "00:00:00";
        }
}
```

```
public class SmartWatch extends Watch {

        private String activeSong;

public void sendText(String txt)
        {
                //TODO: implement
        }
}
```

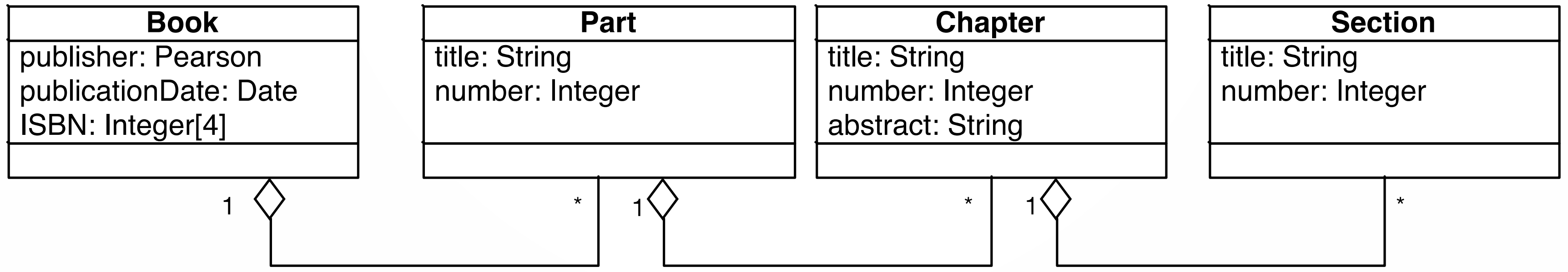
Actor vs. Class vs. Object

- Actor
  - Any entity outside the system, interacting with the system (“Passenger”)
- Class
  - An abstraction modeling an entity inside the system
  - Classes are part of the system model (“User”, “Ticket distributor”, “Server”)
- Object
  - A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).

Exercise #4.4

Consider the class diagram below:

- Note that the Part, Chapter, and Section classes all include a title and a number attribute. Add an abstract class and a inheritance relationship to factor out these two attributes into the abstract class.



Exercise #4.5

Draw a class diagram representing the following facts:

- A **project** involves a **number of participants**.
- **Participants** can take part in a **project** either as **project manager, team leader, or developer**.
- Within a project, each developer and team leader is **part of at least one team**.
- A **participant** can take part in **many projects**, possibly in different **roles**. For example, a participant can be a developer in project A, a team leader in project B, and a project manager in project C. However, the role of a participant within a project does not change.

Communication diagrams

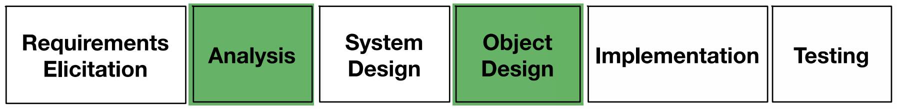
Use case diagrams

– Describe the functional behavior of a system as seen by the user

Class diagrams

– Describe the static structure of the system: objects with attributes, operations and their associations

Communication diagrams



– Describe the interaction between different objects by using method invocation

Component diagrams

– Describe dependencies between components at design time, compilation time and runtime

Deployment diagrams

– Describe the distribution of components on nodes

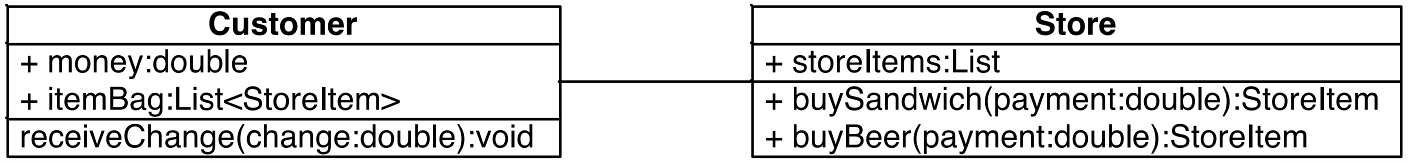
State chart diagrams

– Describe the dynamic behavior of a single, individual object

Communication diagrams

- Describe the interaction between different objects by using method invocation

Example class diagram:

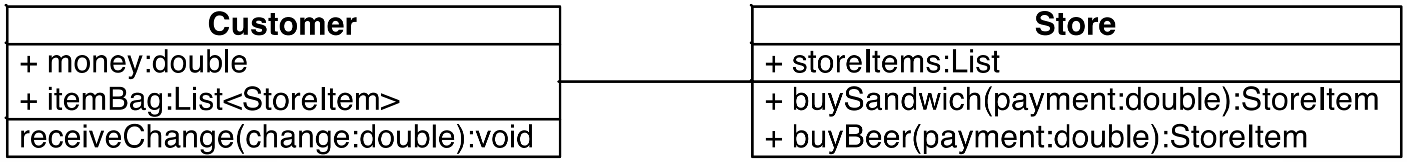

Customer
+ money:double
+ itemBag:List<StoreItem>
receiveChange(change:double):void
Store
+ storeItems:List
+ buySandwich(payment:double):StoreItem
+ buyBeer(payment:double):StoreItem

- **Recall**: Class diagrams show associations and structure but not the message flow represented by different method invocations from concrete instancess

Communication diagrams

Describe the interaction between different objects by using method invocations and sequence expressions

Example class diagram:


Customer
+ money:double
+ itemBag:List<StoreItem>
receiveChange(change:double):void
Store
+ storeItems:List
+ buySandwich(payment:double):StoreItem
+ buyBeer(payment:double):StoreItem

There are different types of messages:

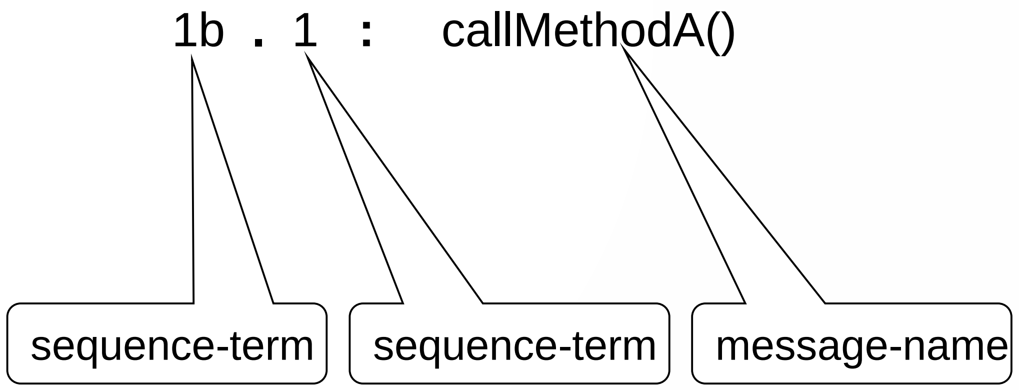
- Independent Messages
- Request-Response Messages
- Concurrent Messages

Communication diagrams

- Notation for Communication diagrams:

- sequence-term ::= [integer [name]] [recurrence]
- sequence-expression ::= sequence-term '.' . . . ':' message-name

- Examples:
- 1:callMethod1()
- 1.1:callMethod2()
- 1a:callMethod3()
- 1b.1:callMethodA()

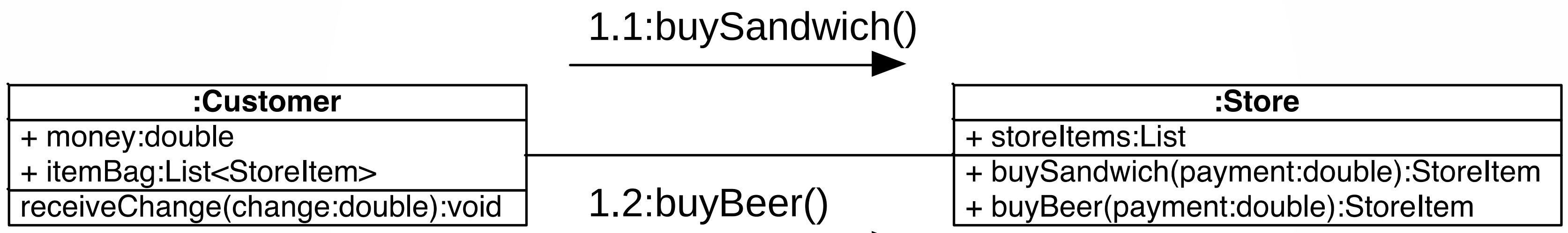


Communication diagrams

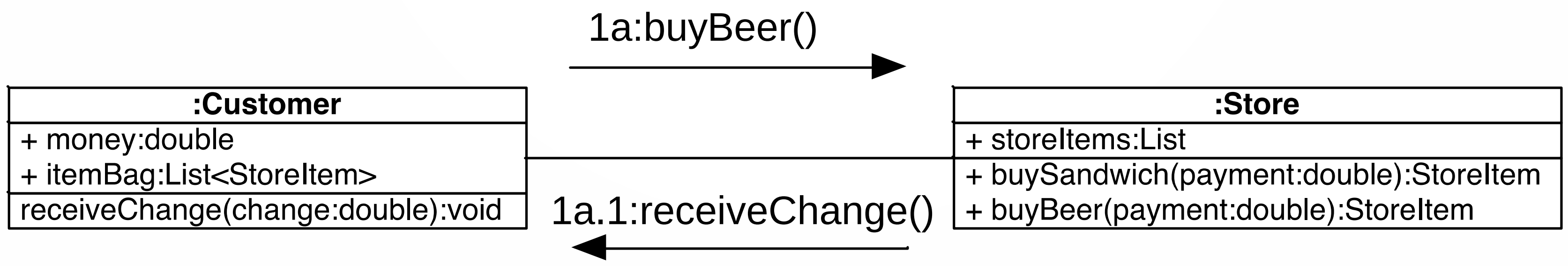
When to use Integer and when to use Name?

sequence-term ::= [integer [name]] [recurrence]

Independent following Messages:

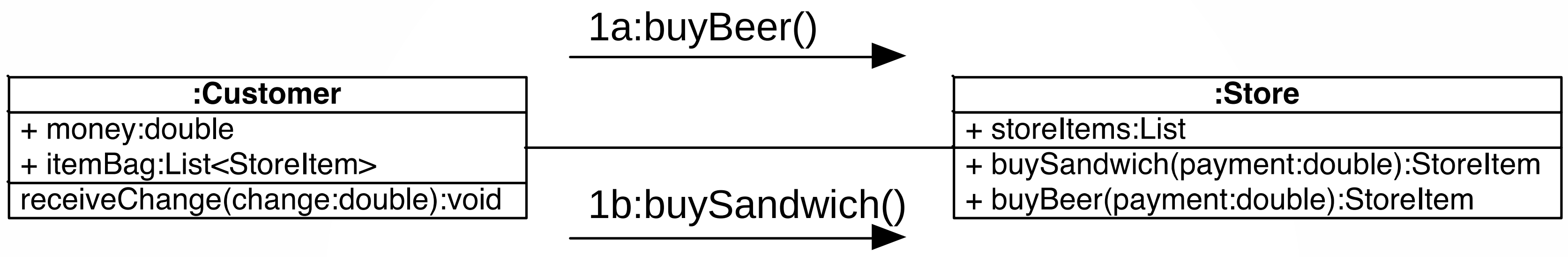

1.1:buySandwich()
:Customer
+ money:double
+ itemBag:List<StoreItem>
receiveChange(change:double):void
1.2:buyBeer()
:Store
+ storeItems:List
+ buySandwich(payment:double):StoreItem
+ buyBeer(payment:double):StoreItem

Request-Response Messages:


1a:buyBeer()
:Customer
+ money:double
+ itemBag:List<StoreItem>
receiveChange(change:double):void
1a.1:receiveChange()
:Store
+ storeItems:List
+ buySandwich(payment:double):StoreItem
+ buyBeer(payment:double):StoreItem

Communication diagram

Concurrent Messages


1a:buyBeer()
:Customer
+ money:double
+ itemBag:List<StoreItem>
receiveChange(change:double):void
1b:buySandwich()
:Store
+ storeItems:List
+ buySandwich(payment:double):StoreItem
+ buyBeer(payment:double):StoreItem

Exercise #4.6

- Given the following textual description first create a class diagram and then create a communication diagram describing the method invocations including their sequence expressions:

- A person has a name and age. Furthermore a person can be a child or an adult. A movie rental has a name and a list of rentable movies. A person can select movies by a given title he wants to rent at the movie rental. If a movie has an age rating the person has to verify his age before he/she can watch the movie.

Lecture Summary

- We covered Software Lifecycles
- We covered Software Lifecycle models
- We covered Software Lifecycle tailoring

We covered UML Part I

- Use case diagrams
- Class diagrams
- Communication diagrams

Next week: UML Part II

- Component diagrams
- Deployment diagrams
- State chart diagrams
- more Exercises