Software Design and Class Diagrams

Massimo Felici

School of **informatics**

Software Design

- The SEOC course is concerned with software design in terms of objects and components, in particular, object-oriented design

- Object-oriented design is part of object-oriented development where an object-oriented strategy is used throughout the development process

- The main activities are: Object-oriented analysis, Object-oriented design, Object-oriented programming

Slide 1: Software Design

There are various definitions about Software Design. They refer to (the result of) the process of defining a software system design consisting in the definition of the architecture, components (or modules), interfaces and other characteristics (e.g., design constraints) of a system or component.

Software design provides a (traceability) link between requirements and an implementable specification. It is a pervasive activity for which often there is no definitive solution. Design solutions are highly context dependent.

Key Design techniques and issues involve the identification of a overall structure or architecture, the identification of the main elements of software that need to be managed.

The design activities involve decomposing system (components) into smaller more manageable (definitions of) components that are easily implementable. Usually, design is a two stage process: architectural design and detailed design. Architectural design (or High-level Design) involves (the identification and specification of) the components forming the system and how they relate one another. Moreover, it is concerned with those issues related to the system architecture. Detailed design deals with the function and characteristics of components and how they relate to the overall architecture.

Slide 1: Software Design

Suggested Readings

- Chapter 14 on Object-oriented design, I. Sommerville. Software Engineering, Eighth Edition, Addison-Wesley 2007.

School of **informatics**

Key Issues in Software Design

- Concurrency

- Workflow and event handling

- Distribution

- Error handling and recovery

- Persistence of data

- Can you think through some of these issues for the SEOC project?

Slide 2: Key Issues in Software Design

- Concurrency – Often there is significant interaction that needs management What are the main concurrent activities? How do we manage their interaction? For instance, in the VolBank example matching and specifying skills and needs goes on concurrently.
- Workflow and event handling What are the activities inside a workflow? How do we handle events?
- Distribution – How is the system distributed over physical (and virtual) systems?
- Error handling and recovery - What are suitable actions when a physical component fails (e.g., the database server)? How to handle exceptional circumstances in the world? For instance, in the VolBank example, a volunteer fails to appear.
- Persistence of data - Does data need to persist across uses of the system, how complex? How much of the state of the process?
- Can you think through some of these issues for VolBank?

School of **informatics**

Key Design Techniques

- Abstraction – ignoring detail to get the high level structure right

- Decomposition and Modularization – big systems are composed from small components

- Encapsulation/information hiding – the ability to hide detail (linked to abstraction)

- Defined interfaces – separable from implementation

- Evaluation of structure – Coupling: How interlinked a component is; Cohesion: How coherent a component is

Architecture and Structure

- Architectural structures and viewpoints
- Architectural styles
- Design patterns
  small-scale patterns to guide the designer
- Families and frameworks
  component sets and ways of plugging them together
  software product lines
- Architectural design

Slide 4: Architecture and Structure

Architectural structures and viewpoints deal with system facets (e.g., physical view, functional or logical view, security view, etc.) separately. Depending on the architectural emphasis, there are different styles, for example, Three-tier architecture for a distributed system (interface, middleware, back-end database), Blackboard, Layered architectures, Model-View-Controller, Time-triggered and so forth.

Architectural Design supports stakeholder communication, system analysis and large-scale reuse. It is possible to distinguish diverse design strategies: function oriented (sees the design of the functions as primary), data oriented (sees the data as the primary structured element and drives design from there), object oriented (sees objects as the primary element of design). There is no clear distinction between Sub-systems and modules. Intuitively, sub-systems are independent and composed of modules, have defined interfaces for communication with other sub-systems. Modules are system components and provide/make use of service(s) to/provided by other modules.

Slide 4: Architecture and Structure

The system architecture affects the quality attributes (e.g., performance, security, availability, modifiability, portability, reusability, testability, maintainability, etc.) of a system. It supports quality analysis (e.g., reviewing techniques, static analysis, simulation, performance analysis, prototyping, etc.). It allows to define (predictive) measures (i.e., metrics) on the design, but they are usually very dependent on the process in use.

The software architecture is the fundamental framework for structuring the system. Different architectural models (e.g., system organizational models, modular decomposition models and control models) may be developed. Design decisions enhance system attributes like, for instance, performance (e.g., localize operations to minimize sub-system communication), security (e.g., use a layered architecture with critical assets in inner layers), safety (e.g., isolate safety-critical components), availability (e.g., include redundant components in the architecture) and maintainability (e.g., use fine-grain self-contained components).

Slide 4: Architecture and Structure

Required Readings

- P. Kruchten. The 4+1 View Model of architecture. IEEE Software, 12(6): 42-50, November, 1995.

Suggested Readings

- P. Kruchten, H. Obbink, J. Stafford. The Past, Present and Future of Software Architecture. IEEE Software, 23(2):22-30, March/April, 2006.

Architecture Models

- A **static structural model** that shows the sub-systems or components that are to be developed as separate units.

- A **dynamic process model** that shows how the system is organized into processes at run-time. This may be different from the static model.

- An **interface model** that defines the services offered by each sub-system through their public interface.

- A **relationship model** that shows relationships such as data flow between the sub-systems.

Slide 5: Comparing Architecture Design Notations

- **Modelling Components:** Interface, Types, Semantics, Constraints, Evolution, Non-functional Properties

- **Modelling Connectors:** Interface, Types, Semantics, Constraints, Evolution, Non-functional Properties

- **Modelling Configurations:** Understandable Specifications, Compositionality (and Conposability), Refinement and Traceability, Heterogeneity, Scalability, Evolvability, Dynamism, Constraints, Non-functional Properties

Slide 5: UML Design Notations

- **Static Notations:** Class and object diagrams, Component diagrams, Deployment diagrams, CRC Cards

- **Dynamic Notations:** Activity diagrams, Communication diagrams, Statecharts, Sequence diagrams

Slide 5: What are the Architects Duties?

- Get it Defined, documented and communicated, Act as the emissary of the architecture, Maintain morale
- Make sure everyone is using it (correctly), management understands it, the software and system architectures are in synchronization, the right modeling is being done, to know that quality attributes are going to be met, the architecture is not only the right one for operations, but also for deployment and maintenance
- Identify architecture timely stages that support the overall organization progress, suitable tools and design environments, (and interact) with stakeholders
- Resolve disputes and make tradeoffs, technical problems
- Manage risk identification and risk mitigation strategies associated with the architecture, understand and plan for evolution

School of **informatics**

Class Diagrams

- Support **architectural design**

  Provide a structural view of systems

- Represent the basics of Object-Oriented systems

  Identify what classes there are, how they interrelate and how they interact

  Capture the static structure of Object-Oriented systems – how systems are structured rather than how they behave

- Constrain interactions and collaborations that support functional requirements

  Link to Requirements

School of **informatics**

Class Diagrams

- Desirable to build systems quickly and cheaply (and to meet requirements)

- Desirable to make the system easy to maintain and modify

- **Warnings**

  - The classes should be derived from the (user) domain - **avoid abstract objects**
  - Classes provide limited support to capture system behaviour – **avoid to capture non-functional requirements of the system as classes**

Class Diagrams in the Life Cycle

- Used throughout the development life cycle

- Carry different information depending on the phase of the development process and the level of detail being considered

  From the problem to implementation domain

Slide 8: Class Diagrams in the Life Cycle

Class diagrams can be used throughout the development life cycle. They carry different information depending on the phase of the development process and the level of detail being considered. The contents of a class diagram will reflect this change in emphasis during the development process. Initially, class diagrams reflect the problem domain, which is familiar to end-users. As development progresses, class diagrams move towards the implementation domain, which is familiar to software engineers.

School of
informatics

Class Diagrams

- Classes

  – Basic Class Components
  – Attributes and Operations

- Class Relationships

  – Associations
  – Generalizations
  – Aggregations and Compositions

School of **informatics**

Class Diagrams

Construction involves

1. Modelling classes

2. Modelling relationships between classes, and

3. Refining and elaborate as necessary

Classes and Objects

- Classes represent groups of objects all with similar roles in the system

  Structural features define what objects of the class know

  Behavioral features define what objects of the class can do

- Classes may

  inherit attributes and services from other classes

  be used to create objects

- Objects are instances of classes, real-world and system entities

Slide 11: Classes and Objects

Objects are entities in a software system which represent instances of real-world and system entities. Objects derive from things (e.g., tangible, real-world objects, etc.), roles (e.g., classes of actors in systems like students, managers, nurses, etc.), events (e.g., admission, registration, matriculation, etc.) and interactions (e.g., meetings, tutorials, etc.).

Objects are created according to some class definition. A class definition serves as a template for objects and includes declarations of all the attributes and operations which should be associated with an object of that class. Note that the level of detail known or displayed for attributes and operations depends on the phase of the development process. An object is an entity that has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects, which request these services when some functionality is required.

Basic Class Compartments

- **Name**

- **Attributes**

  – represent the state of an object of the class
  – are descriptions of the structural or static features of a class

- **Operations**

  – define the way in which objects may interact
  – are descriptions of behavioural or dynamic features of a class

Example 13

School of **informatics**

Basic Class Compartments

- - - - *Name*

Employee
+name : string
-address : string
+employeeNumber : int
-socialSecurityNumber : int
+department : string
-salary : int
-taxCode : int
+status : string
+join()
+leave()
+retire()
+changeInformation()

- - - *Attributes*

- - - *Operations*
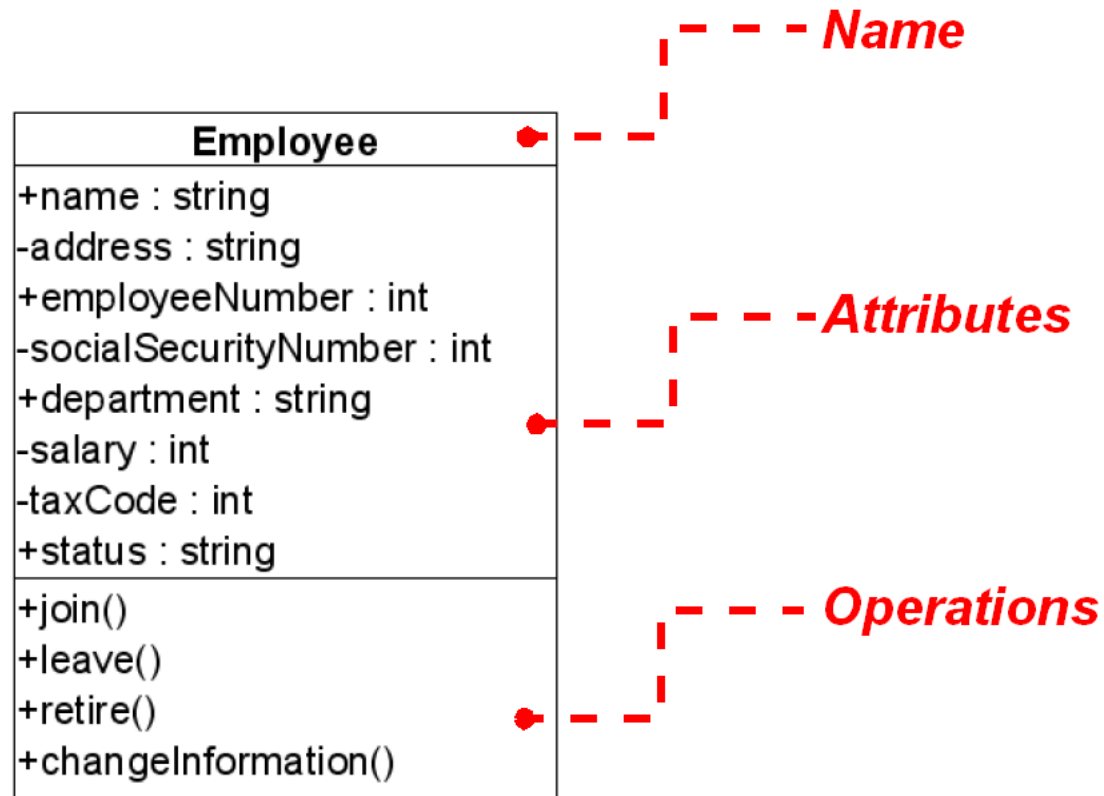
Slide 13: Basic Class Compartments

```
public class Employee {
    public string name;
    private string address;
    public int employeeNumber;
    private int socialSecurityNumber;
    private string department;
    private int salary;
    private string taxCode;
    public string status;

    public void join() {
    }

    public void leave() {
    }

    public void retire() {
    }

    public void changeInformation() {
    }
}
```

Attribute Definition

```
visibility / name :  type multiplicity = default {property strings and constraints}
```

- **visibility**
- **/ derived attribute** - Attributes by relationship allow the definition of complex attributes
- **name**
- **type** is the data type of the attribute or the data returned by the operation
- **multiplicity** specifies how many instances of the attributes type are referenced by this attribute
- **property strings:** readOnly, union, subset ¡attribute-name¿, redefines ¡attribute-name¿ composite, ordered, bag, sequence, composite
- **constraints**

Slide 14: Derived Attributes

Account
-balance : float - / overdrawn : boolean

overdrawn == true
if balance < 0

DataRange
start : Date end : Date / length : int

{length = end - start}

School of **informatics**
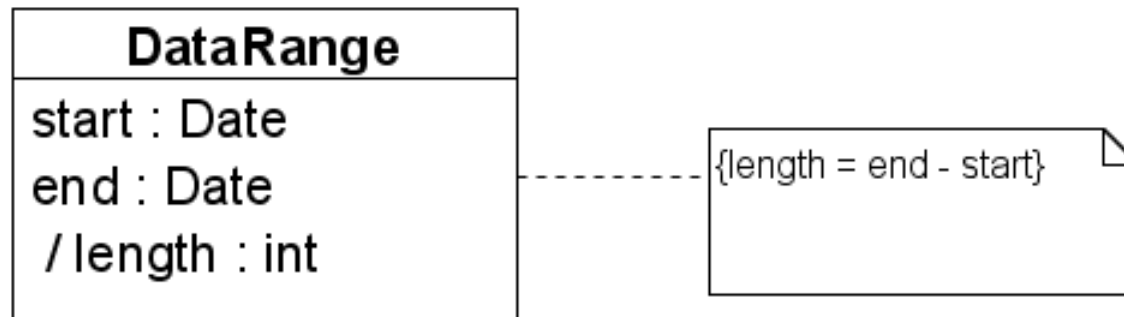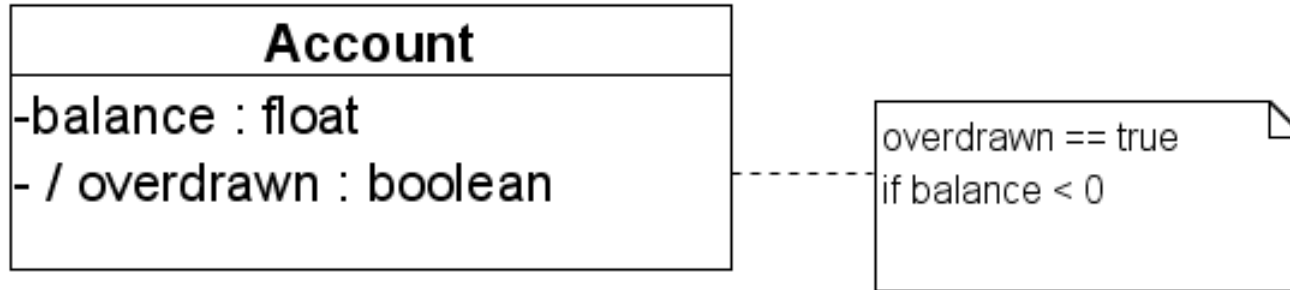
Visibility

- From More accessible to Less Accessible

  public (**+**), protected ($\sharp$), package($\sim$), private (-)

- *Warnings: Java allows access to protected parts of a class to any class in the same package*

- *Warnings: Although many languages use such terms as public, private, and protected, they mean different things in different languages. The meanings of visibility markers can change from language to language.*

Slide 15: Visibility

- **public** (+) visibility means that the feature is available to any class associated with the class that owns the class

- **protected** ($\sharp$) visibility means that the feature is available within the class that owns that feature and any subtype of that class

- **package** ($\sim$) visibility means that the feature is available only to other classes in the same package as the declaring class (and the declaring class itself)

- **private** (-) visibility means that the feature is available only within the class that owns that feature

Multiplicity

Multiplicity specifies how many instances of the attributes type are referenced by this attribute

- **[n..m]** – n to m instances

- **0..1** – zero or one instance

- **0..*** or ***** – no limit on the number of instances (including none)

- **1** – exactly one instance

- **1..*** – at least one instance

Operation Definition

`visibility name (parameters) :  return-type {properties}`
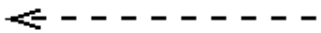
- **(Parameters)**

  `direction parameter_name :  type [multiplicity] = default_value properties`
- **direction:** in, inout, out or return
- **Operation constraints:** preconditions, postconditions, body conditions, query operations, exceptions
- **Static operations:** Specify behaviour for the class itself; Invoked directly on the class
- **Methods** are implementations of operations
  Abstract classes provide operation signatures, but no implementations

School of **informatics**

Class Relationships

Relationship	Description
	Dependency: objects of one class work briefly with objects of another class
	Association: objects of one class work with objects of another class for some prolonged amount of time
	Aggregation: one class owns but share a reference to objects of other class
	Composition: one class contains objects of another class
	Generalization (Inheritance): one class is a type of another class

Dependency

- A dependency exists between two elements if changes to the definition of one element (the supplier or target) may cause changes to the other (the client or source)

- A Dependency between two classes means that one class uses, or has knowledge of, another class (i.e., a transient relationship)

- Dependency relationships show that a model element requires another model element for some purpose

- Dependencies can also indicate relationships between model elements at different level of abstraction

Slide 19: Dependency

Dependencies between classes exist for various reasons:

- one class sends a message to the other

- one class has another has part of its data

- one class mentions another as a parameter to an operation

Slide 19: Selected Dependency Keywords

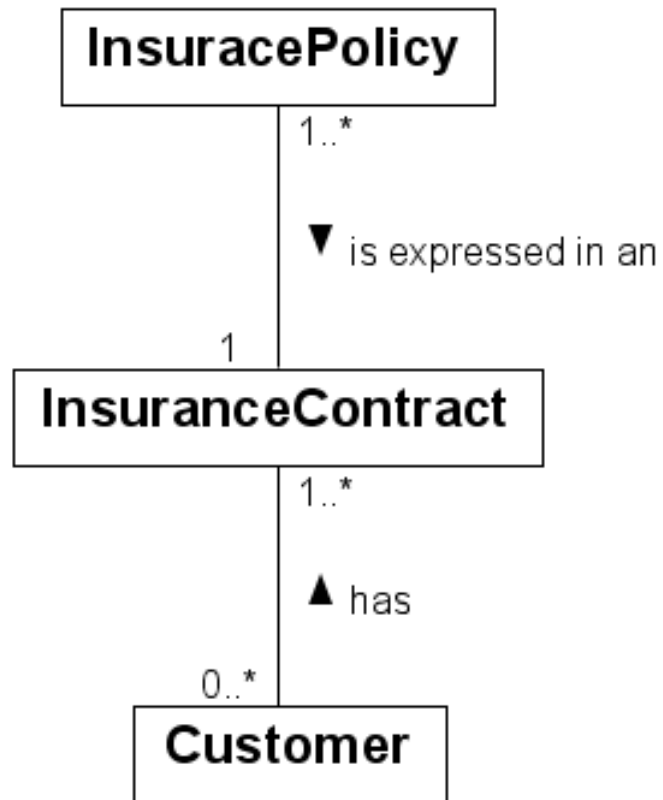| Keyword	Meaning
<<call>>	The source calls an operation in the target
<<create>>	The source creates instances of the target
<<derive>>	The source is derived from the target
<<instantiate>>	The source is an instance of the target
<<permit>>	The target allows the source to access the target's private feature
<<realize>>	The source is an implementation of a specification or interface defined by the target
<<refine>>	Refinement indicates a relationship between different semantic levels
<<substitute>>	The source is substitutable for the target
<<trace>>	Used to track such things as requirements to classes ot how changes in one model link to changes elsewhere
<<use>>	The source requires the target for its implementation

Dependency
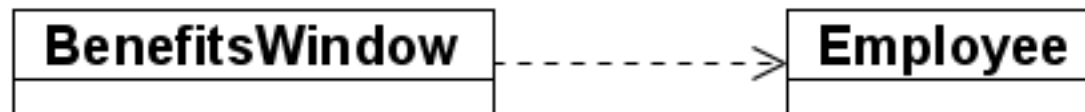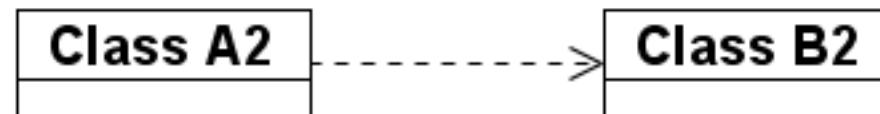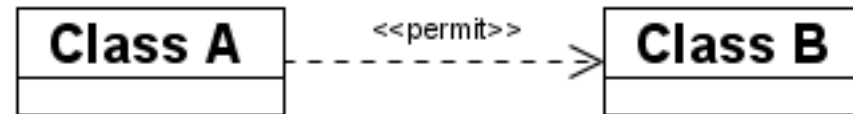
Association

- an attribute of an object is an associated object

- a method relies on an associated object

- an instance of one class must know about the other in order to perform its work

- Passing messages and receiving responses

Associations may be annotated with information: Name, Multiplicity, Role Name, Ends, Navigation
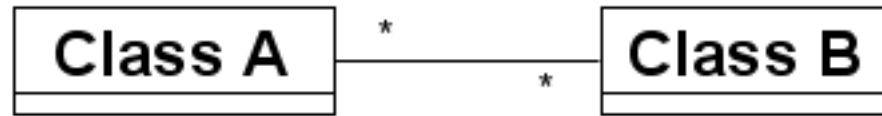
Association
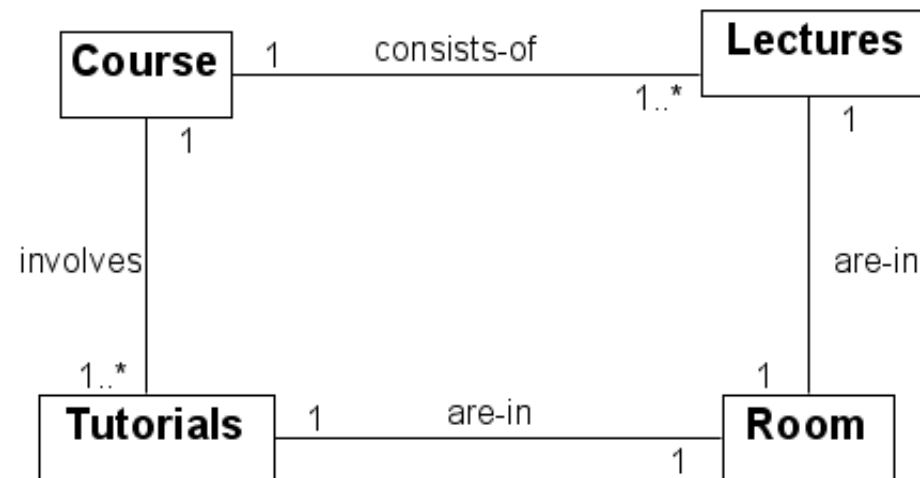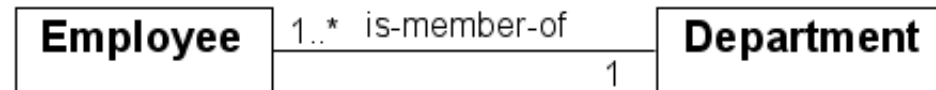
Example 23

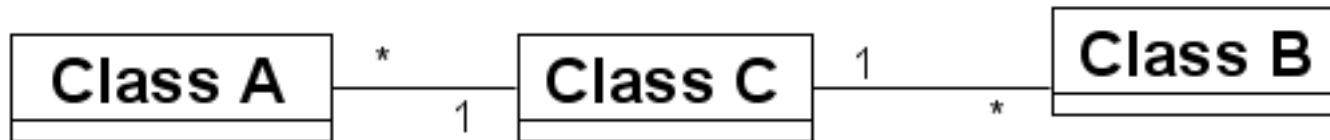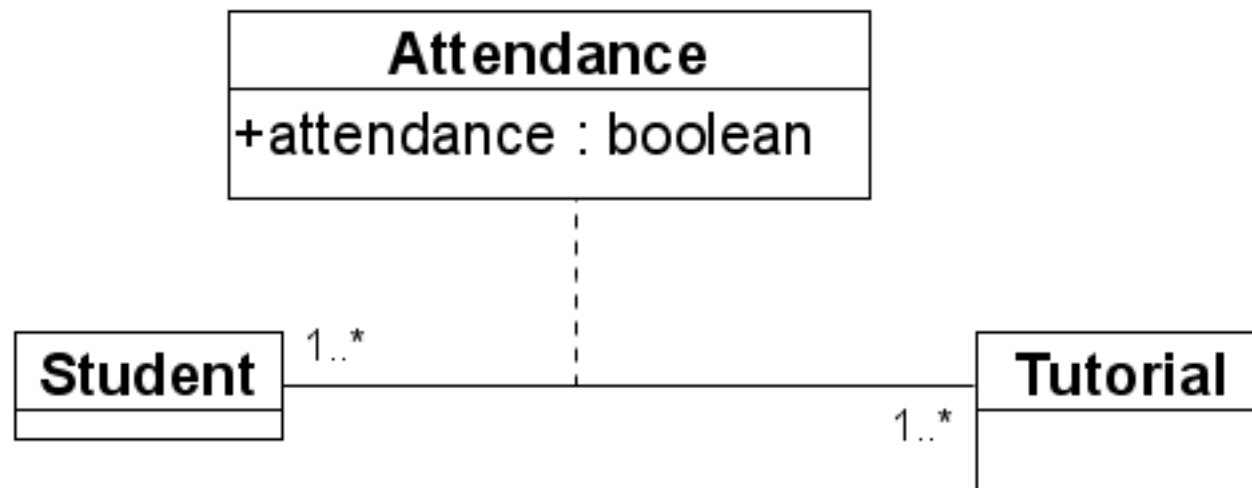# Association

Slide 23: Association Example

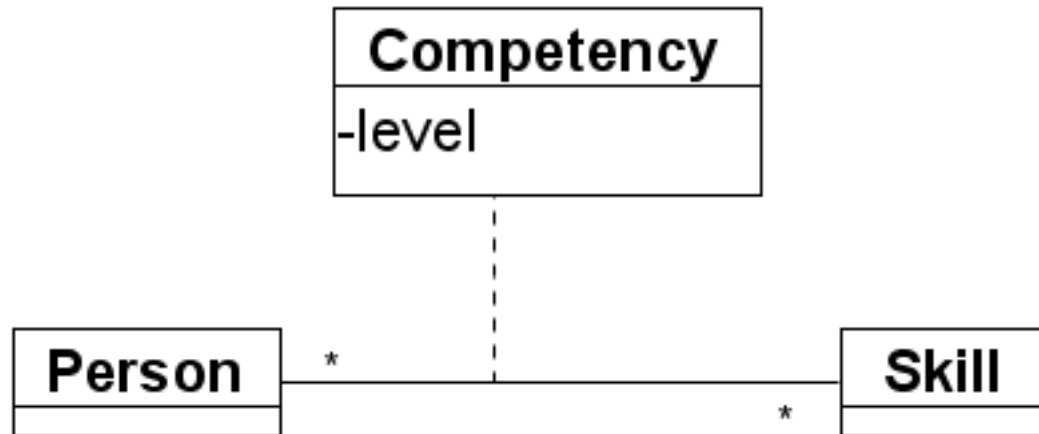# Association Class

- A class can be attached to an association, in which case it is called an association class

- The association class is not connected at any of the ends of the association, but is connected to the actual association

- Association classes allow you to add attributes, operations, and other features to associations

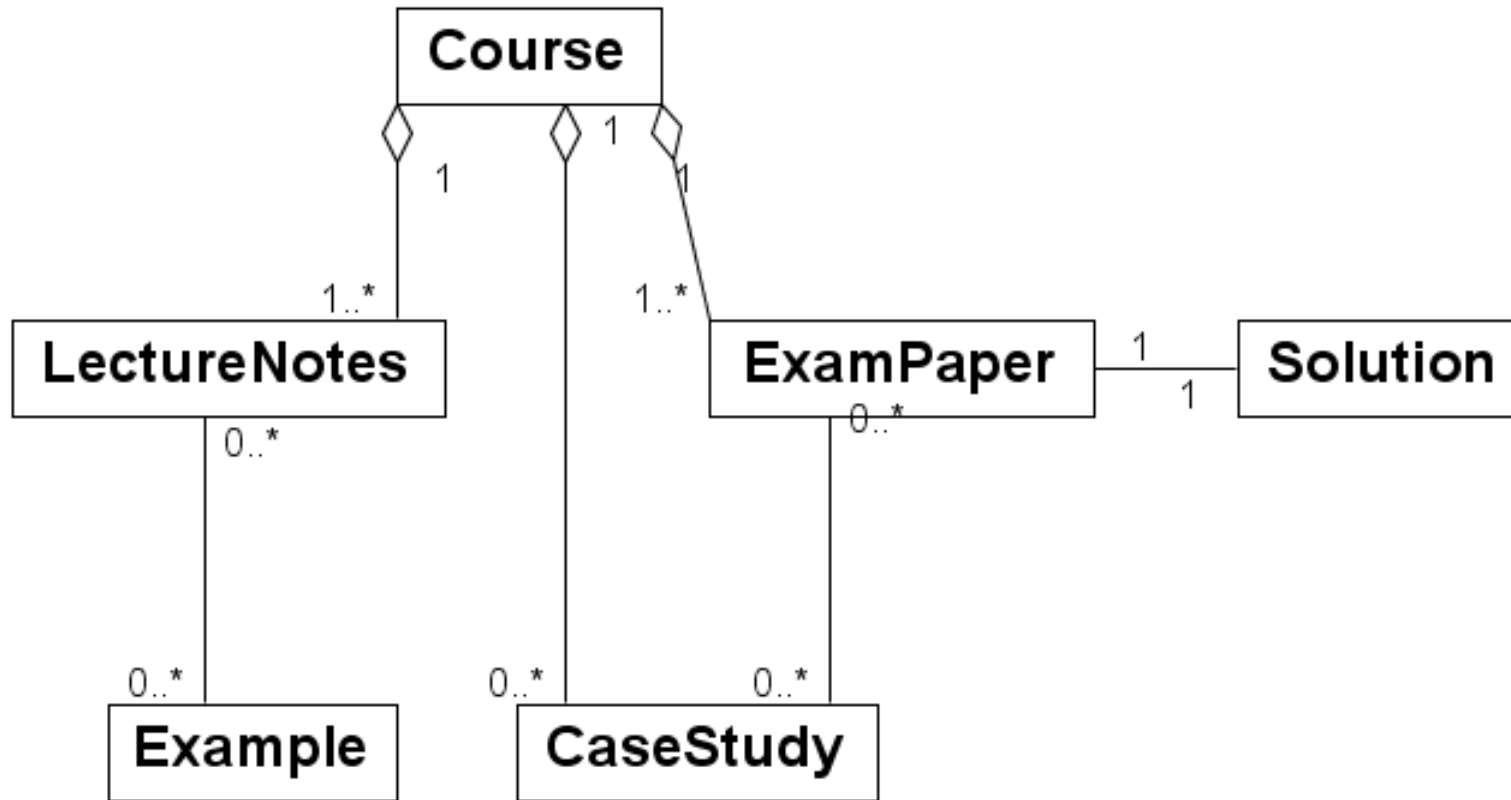Example   25

Association Class

Slide 25: Another Example of Association Class
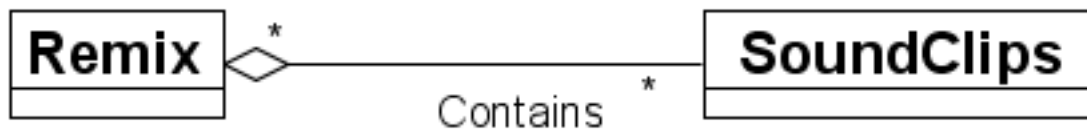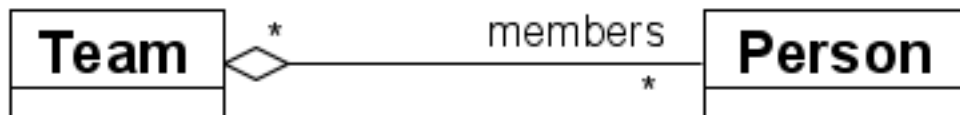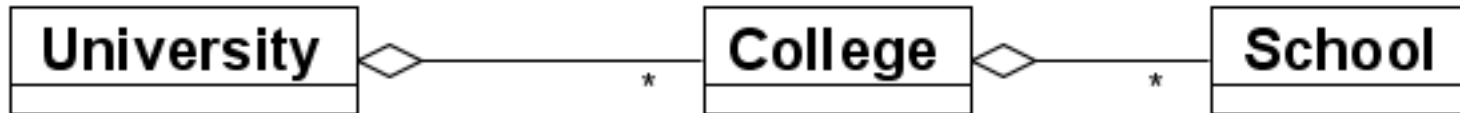
School of
informatics

Aggregation

- is a stronger version of association

- is used to indicate that, as well as having attributes of its own, an instance of one class may consist of, or include, instances of another class

- are associations in which one class belongs to a collection

Example   27

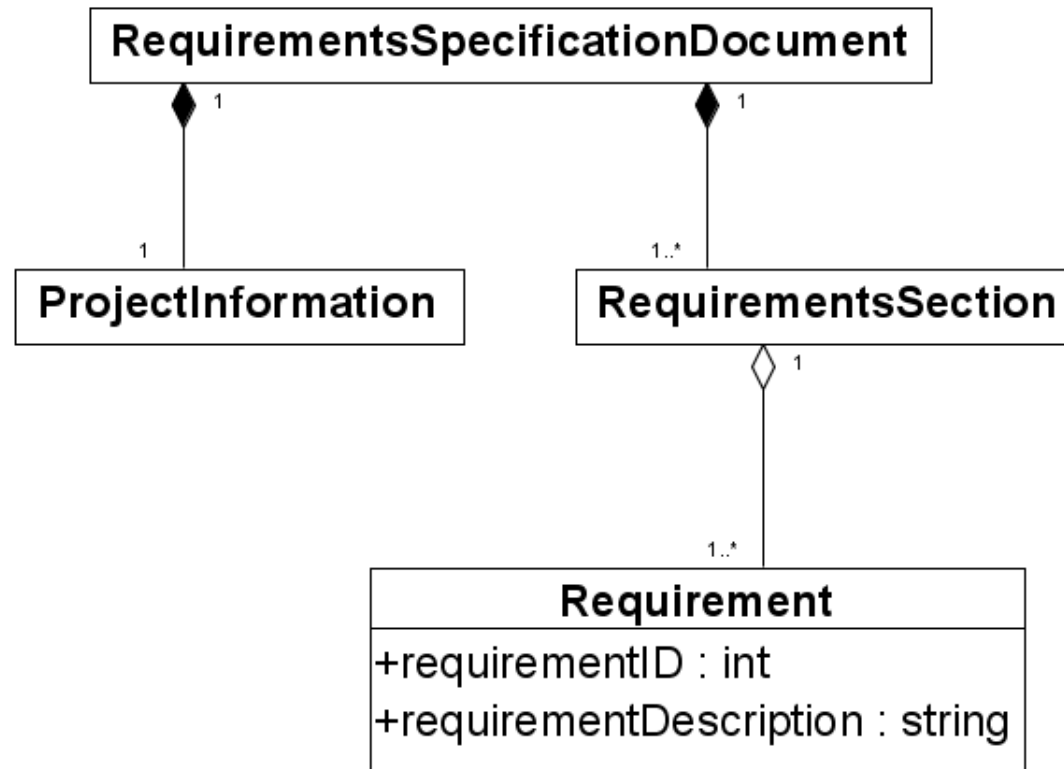# Aggregation

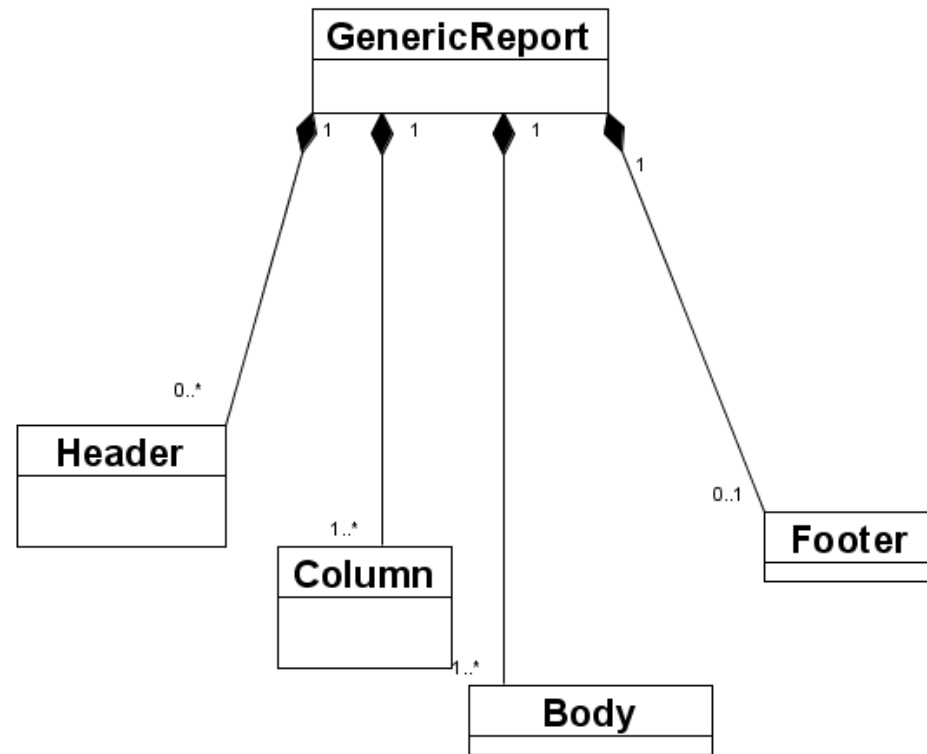# Slide 27: Other Examples of Aggregations

Composition

- Compositions imply coincident lifetime.

- A coincident lifetime means that when the whole end of the association is created (deleted), the part components are created (deleted).

Example 29

Composition

Slide 29: Another Examples of Composition

Note that the java code implementation for an aggregation (composition) relationship is exactly the same as the implementation for an association relationship. It results in the introduction of an attribute.

Aggregation versus Composition

Criteria	Decision Result
Part-of, contains, owns words are used to describe relationship between two classes	Aggregation or Composition
No symmetry	Aggregation or Composition
Transitivity among parts	Aggregation or Composition
Parts are not shared	Composition
Multiplicity of the whole is 1 or $0..1$	Composition
Parts may be shared	Aggregation
Multiplicity of the whole may be larger that 1	Aggegration
Relationship does not fit the other criteria	Association

School of **informatics**

Generalization (Inheritance)

- An inheritance link indicating one class is a superclass of the other, the subclass

  - An object of a subclass to be used as a member of the superclass
  - The behaviour of the two specific classes on receiving the same message should be similar

- Checking Generalizations: If class A is a generalization of a class B, then "Every B is an A"

Slide 31: Design by Contract

Design by Contract: A subclass must keep to the contract of the superclass by ensuring operations observe the pre and post conditions on the methods and that the class invariant is maintained.

Assertion is central to Design by Contract – An assertion is a boolean statement that should never be false and, therefore, will be false only because of a bug.

Design by Contract uses three particular kinds of assertions: **post-conditions**, **pre-conditions** and **invariants**.

A **post-condition** is a statement of what the world should look like after execution of an operation.
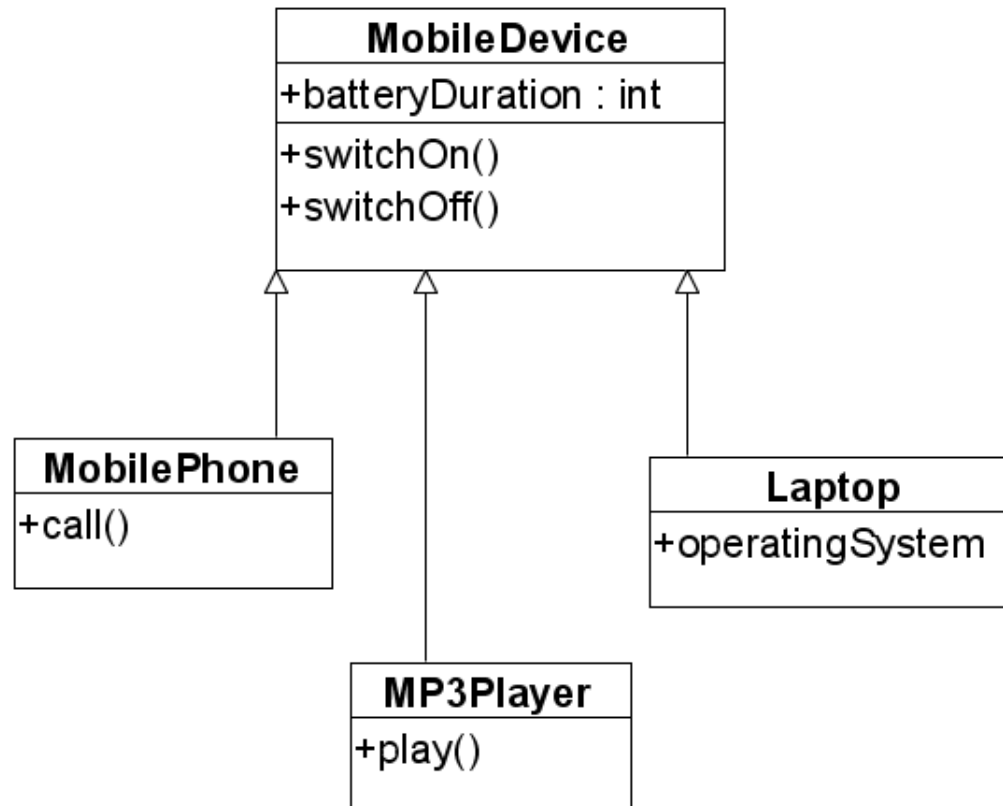
A **pre-condition** is a statement of how we expect the world to be before we execute an operation.

An **invariant** is an assertion about a class.

Suggested Readings

- B. Meyer. Applying "design by contract". IEEE Computer, 25(10):40-51, 1992.

Example1 32

School of **informatics**

Generalization (Inheritance)

Example2   33

Generalization (Inheritance)
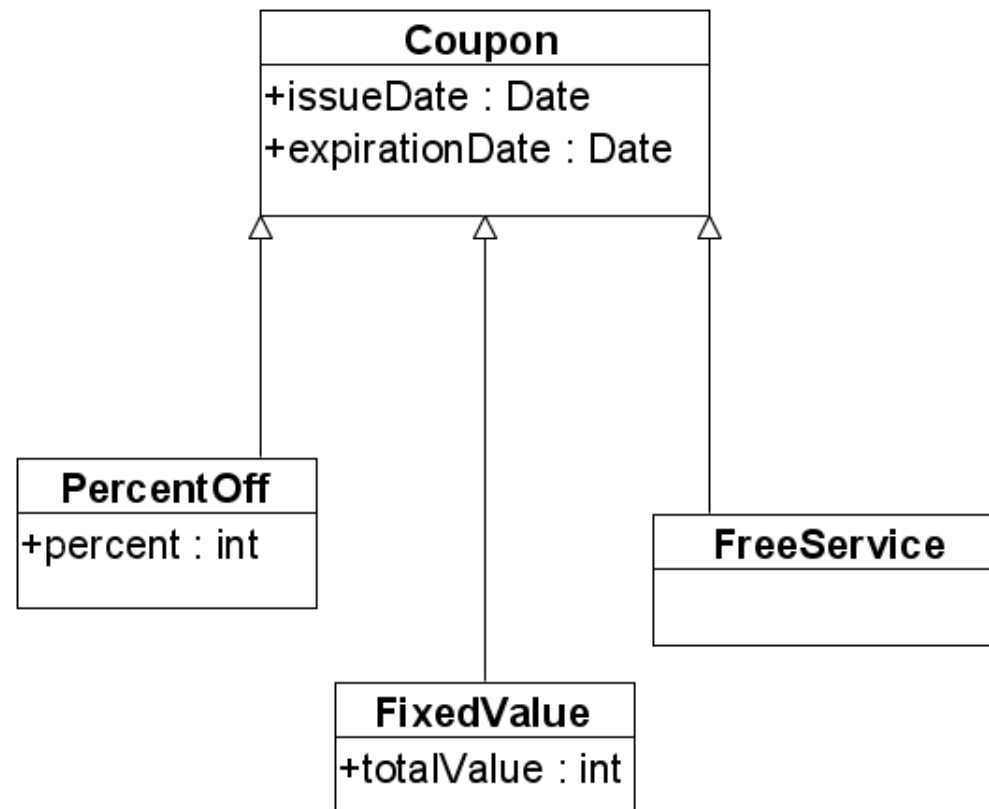
Slide 33: Generalization (Inheritance)

```
public class Coupon {
    public Date issueDate;
    public Date expirationDate;
}
```

```
public class PercentOff extends Coupon {
     public int percent;
}
```

```
public class FixedValue extends Coupon {
    public int totalValue;
}
```

```
public class FreeService extends Coupon {
}
```

Generalization (Inheritance)

- Java: creating the subclass by extending the superclass

- Inheritance increases system coupling

- Modifying the superclass methods may require changes in many subclasses

- Restrict inheritance to conceptual modelling

- Avoid using inheritance when some other association is more appropriate

More on Classes

- **Abstract Classes** provide the definition, but not the implementation

- **Interfaces** are collections of operations that have no corresponding method implementations

  – Safer than Abstract classes avoid many problems associated with multiple inheritance
  – Java allows a class to implement any number of interface, but a class inherit from only one regular or abstract class

- **Templates** or **parameterized classes** allow us to postpone the decision as to which classes a class will work with
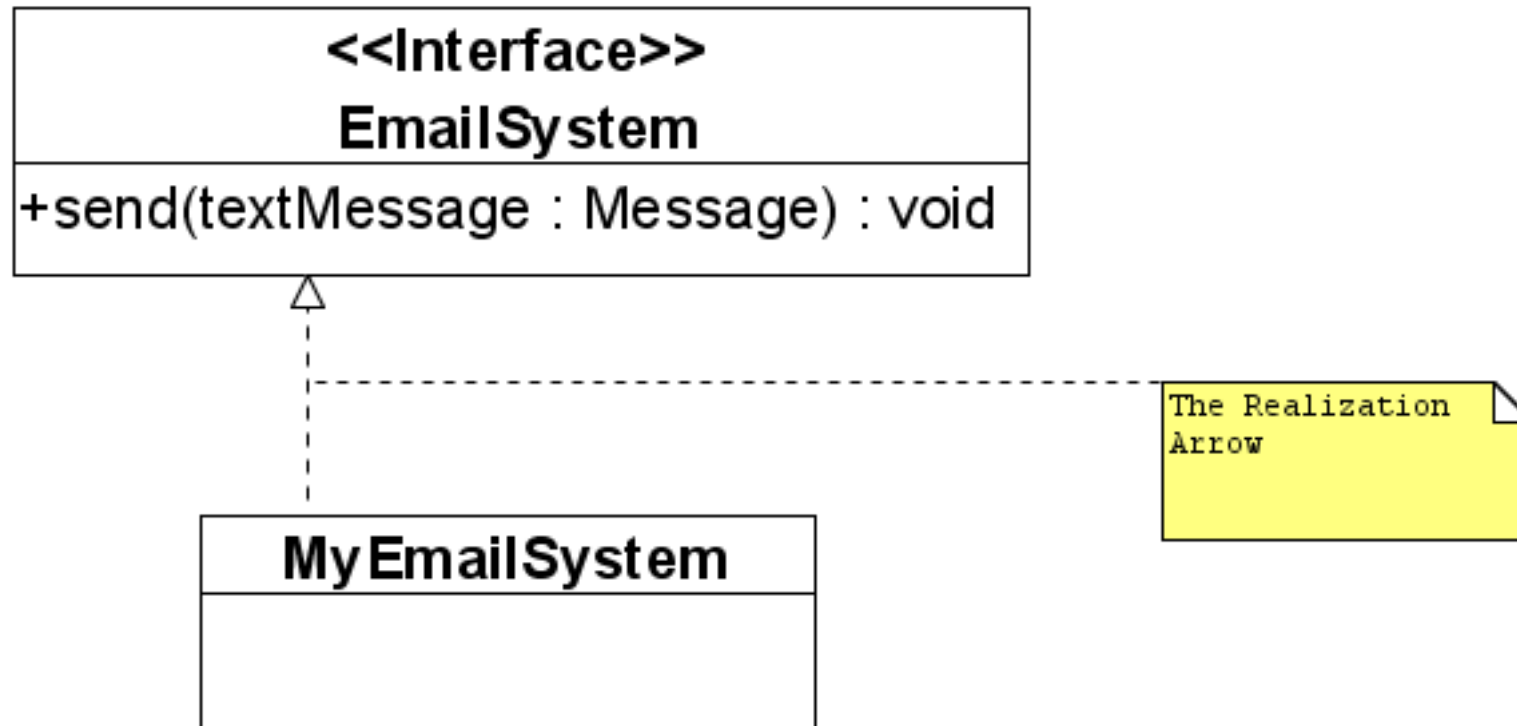
Example 36
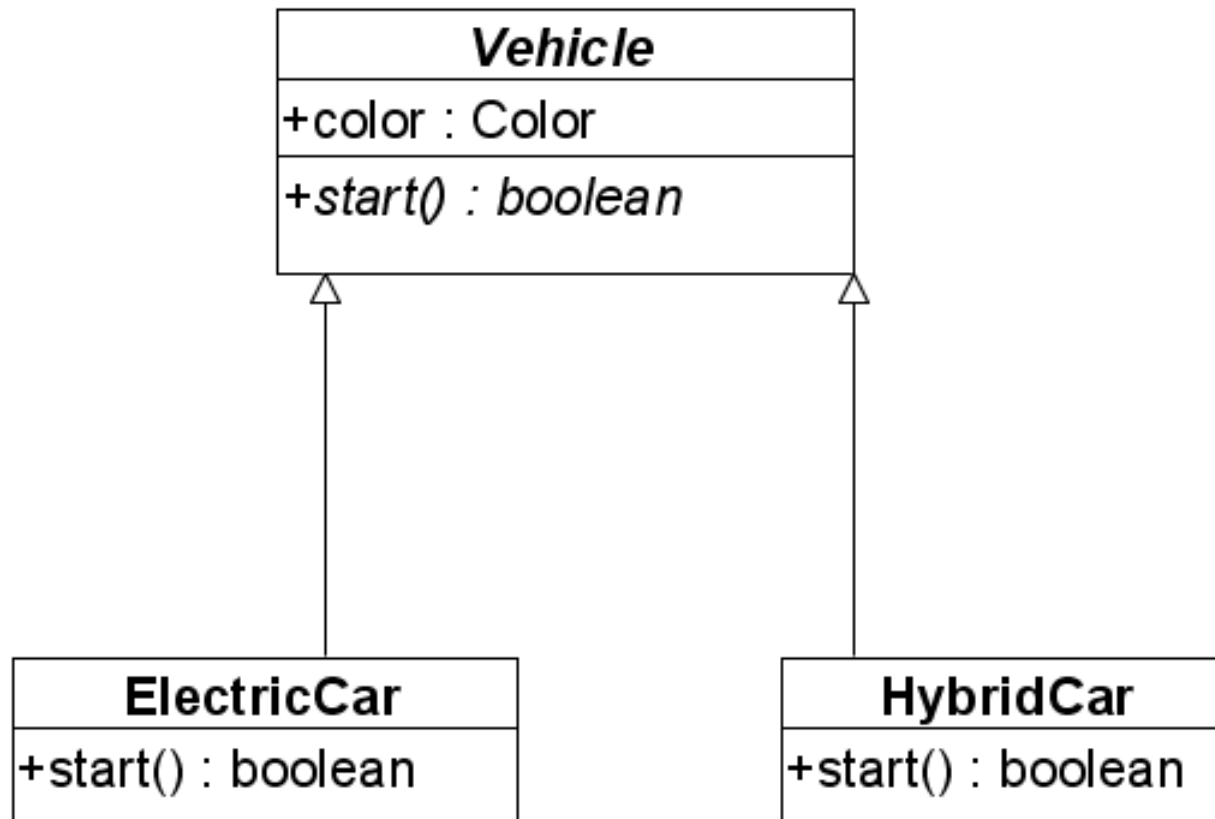
Abstract Classes

Slide 36: Abstract Classes

```
public abstract class Vehicle {

    public Color color;

    public abstract boolean start();

}
```

```
public class ElectricCar extends Vehicle {

    public boolean start() {
    }

}
```

```
public class HybridCar extends Vehicle {

    public boolean start() {
    }

}
```

Example   37

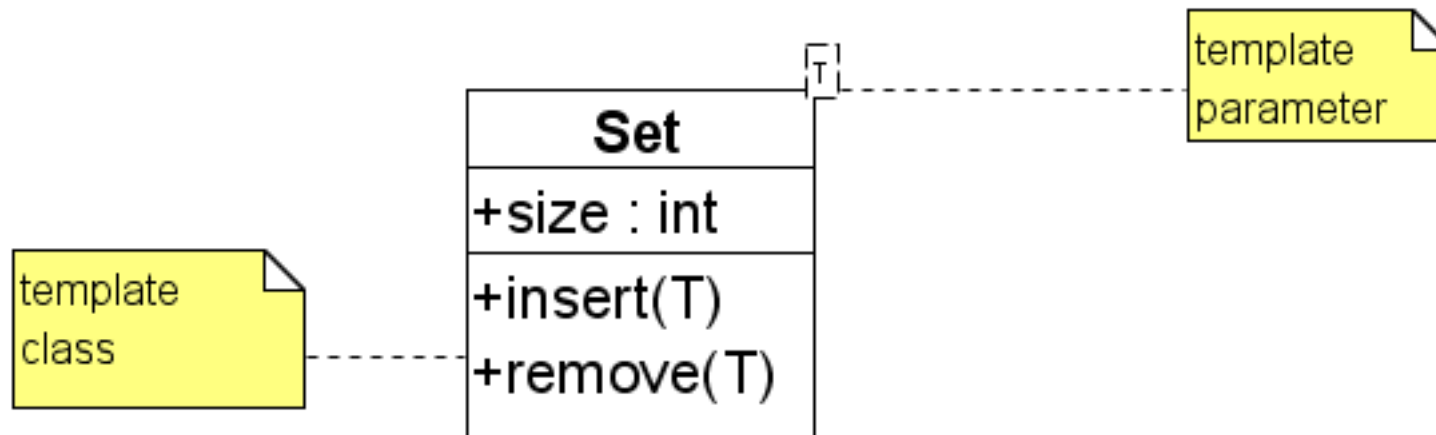School of **informatics**

Interface Classes

Slide 37: Interface Classes

```java
public interface EmailSystem {

    void send(Message textMessage);

}
```

```java
public class MyEmailSystem implements EmailSystem {

}
```

Template Classes

School of
informatics

Modelling by Class Diagrams

- **Class Diagrams** (models)

  – from a conceptual viewpoint, reflect the requirements of a problem domain
  – From a specification (or implementation) viewpoint, reflect the intended design or implementation, respectively, of a software system

- Producing class diagrams involve the following iterative activities:

  – Find classes and associations (directly from the use cases)
  – Identify attributes and operations and allocate to classes
  – Identify generalization structures

School of **informatics**

How to build a class diagram

- Design is driven by criterion of completeness either of data or responsibility
  - **Data Driven Design** identifies all the data and see it is covered by some collection of objects of the classes of the system
  - **Responsibility Driven Design** identifies all the responsibilities of the system and see they are covered by a collection of objects of the classes of the system
- **Noun identification**
  - **Identify noun phrases:** look at the use cases and identify a noun phrase. Do this systematically and do not eliminate possibilities
  - **Eliminate inappropriate candidates:** those which are redundant, vague, outside system scope, an attribute of the system, etc.
- Validate the model...

School of **informatics**

Common Domain Modelling Mistakes

- Overly specific noun-phrase analysis

- Counter-intuitive or incomprehensible class and association names

- Assigning multiplicities to associations too soon

- Addressing implementation issues too early: Presuming a specific implementation strategy, Committing to implementation constructs, Tackling implementation issues

- Optimising for reuse before checking use cases achieved

Class and Object Pitfalls

- Confusing basic class relationships (i.e., is-a, has-a, is-implemented-using)

- Poor use of inheritance

  - Violating encapsulation and/or increasing coupling
  - Base classes do too much or too little
  - Not preserving base class invariants
  - Confusing interface inheritance with implementation inheritance
  - Using multiple inheritance to invert is-a

School of **informatics**

Required Readings

- UML course textbook

  - Chapter 4 on Class Diagram: Classes and Associations
  - Chapter 5 on Class Diagram: Aggregation, Composition and Generalization
  - Chapter 6 on Class Diagram: More on Associations
  - Chapter 7 on Class Diagram: Other Notations

- P. Kruchten. The 4+1 View Model of architecture. IEEE Software, 12(6): 42-50, November, 1995.

School of **informatics**

Suggested Readings

- Chapter 14 on Object-oriented design, I. Sommerville. Software Engineering, Eighth Edition, Addison-Wesley 2007.

- P. Kruchten, H. Obbink, J. Stafford. The Past, Present and Future of Software Architecture. IEEE Software, 23(2):22-30, March/April, 2006.

- B. Meyer. Applying "design by contract". IEEE Computer, 25(10):40-51, 1992.

School of **informatics**

Summary

- Design is a complex matter
- Design links requirements to construction, essential to ensure traceability
- Class Diagram Rationale
- Classes
- Class Relationships
- Modelling by Class Diagrams
- How to build a class diagram
- Common domain modeling mistakes
- Class and Object Pitfalls