Software Design and Class Diagrams

Massimo Felici

JCMB-1402 0131 650 5899

1BP-G04 0131 650 4408

mfelici@inf.ed.ac.uk

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**

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?

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

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.

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**

VolBank: A Design Example

- Two possible **requirements**
  - That a request for a volunteer should produce a list of volunteers with appropriate skills.
  - The system shall ensure the safety of both volunteers and the people and organizations who host volunteers.
- **Traceability** from **requirements** to **components**
  - By allocating a particular requirement to a particular component as we decompose, e.g., in VolBank, we might require a log
  - By decomposing requirements into more refined requirements on particular components, e.g., a particular function in VolBank might be realized across several components
  - Some requirements (e.g., usability) are harder to decompose, e.g., it takes 30 minutes to become competent in using the system

Class Diagram Rationale

- 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 object**
  - 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**

Class Diagram Basics

- **Classes**
  - Basic Class Components
  - **Attributes** and **Operations**
- Class **Relationships**
  - Associations
  - Generalizations
  - Aggregations and Compositions

Construction involves

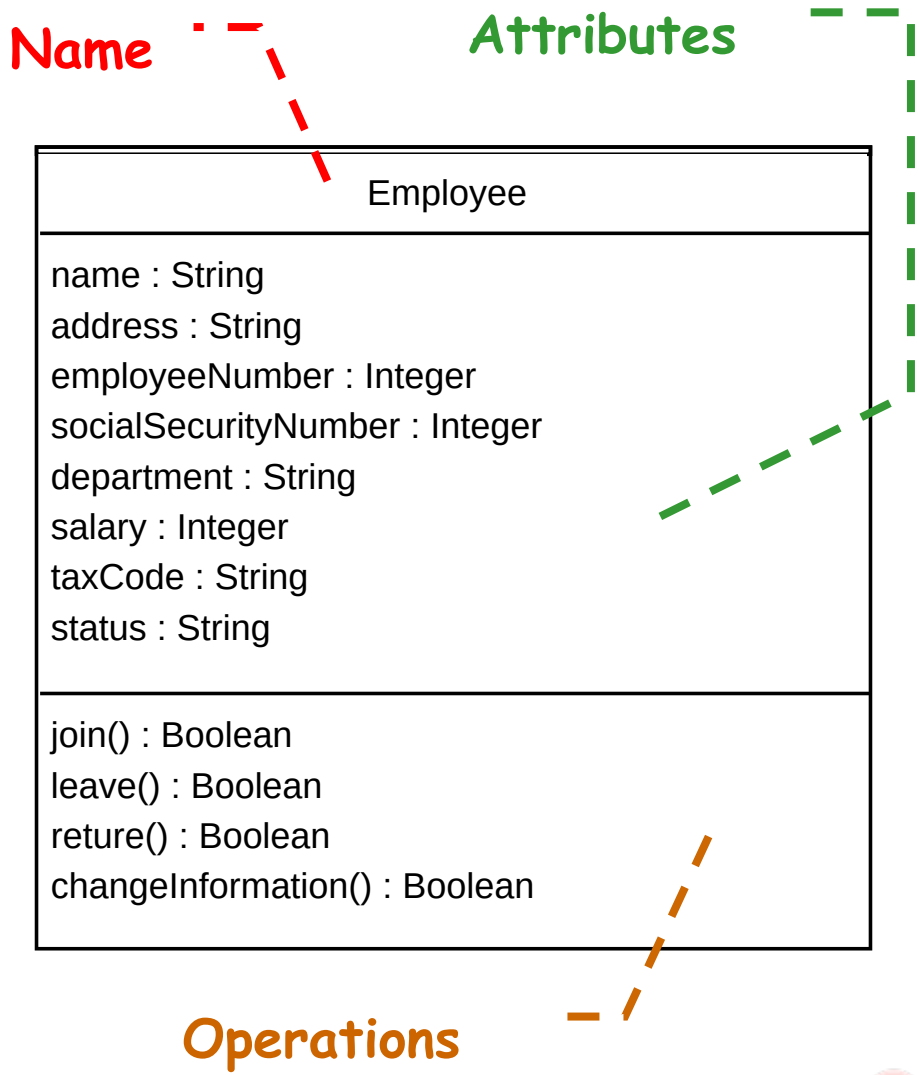
1. Modeling **classes**
2. Modeling **relationships** between classes and
3. Refining and elaborate as necessary

Classes and Objects

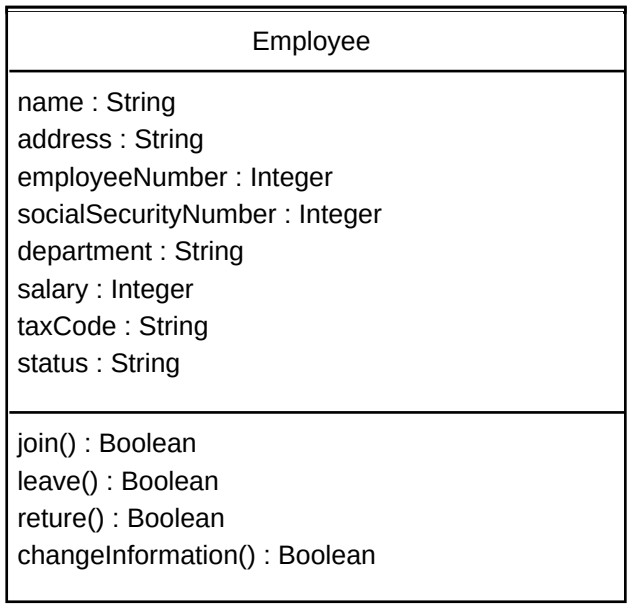
- **Classes** represent groups of objects all with similar roles in the system
  - **Structural features** define what objects of the class know
  - **Behavioral features** define what objects of the class can do
- **Classes** may
  - inherit attributes and services from other classes
  - be used to create objects
- **Objects** are instances of classes, real-world and system entities

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 behavioral or dynamic features of a class



Java Class Definition




```
class Employee {
  public String name;
  public String address;
  public Integer employeeNumber;
  public Integer socialSecurityNumber;
  public String department;
  public Integer salary;
  public String taxCode;
  /**
   *  current
   */
  public String status;
  public Boolean join() {
  return null;
  }
  public Boolean leave() {
  return null;
  }
  public Boolean reture() {
  return null;
  }
  public Boolean changeInformation() {
  return null;
  }
}
```

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 attribute's type are referenced by this attribute
- **property strings**: readOnly, union, subset <attribute-name>, redefines <attribute-name> composite, ordered, bag, sequence, coposite
- **constraints**

Visibility and Multiplicity

- **Visibility**
  - public (+), protected (#), package(~), private (-)

  From **More accessible** to **Less Accessible**
  - Warnings: Java allows access to protected parts of a class to any class in the same package
- **Multiplicity** specifies how many instances of the attribute's 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

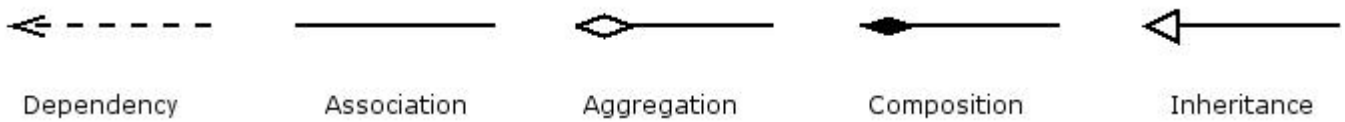
visibility name (parameters) : return-type {properties}

- **(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 an operations; Abstract classes provide operation signatures, but no implementations

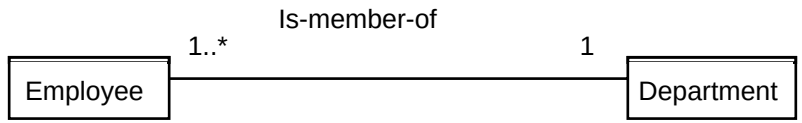
Class Relationships

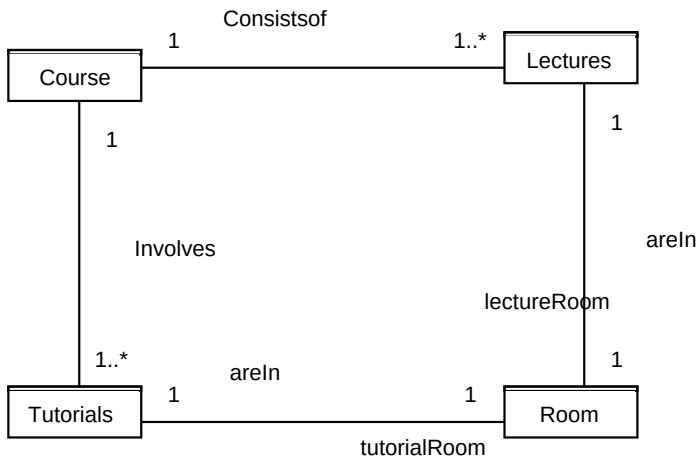


- **Dependency**: objects of one class work briefly with objects of another class
- **Association**: objects of one class work with objects of another class for some prolonged amount of time
- **Aggregation**: one class owns but share a reference to objects of other class
- **Composition**: one class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

Dependency and Association

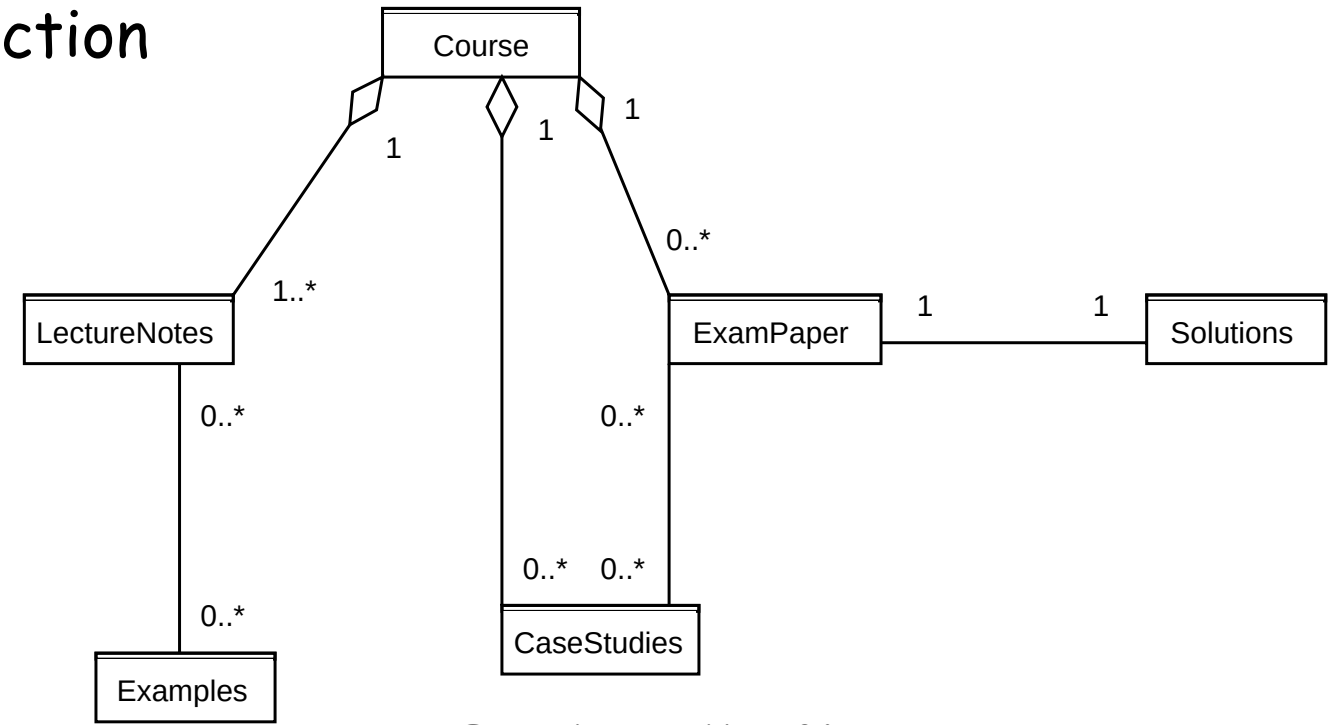
- **Dependency** between two classes means that one class uses, or has knowledge of, another class (i.e., a transient relationship)
- **Associations**
  - an attribute of an **object** is an associated **object**
  - a method relies on an associated object
  - an instance of one class must know about 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





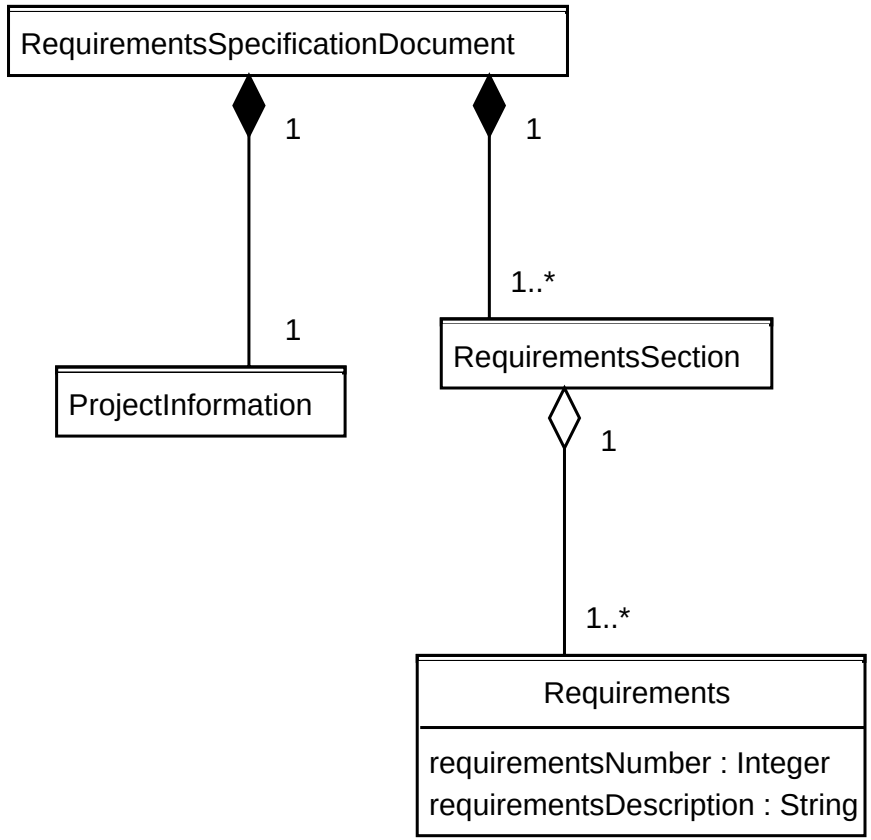
Aggregation

- 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



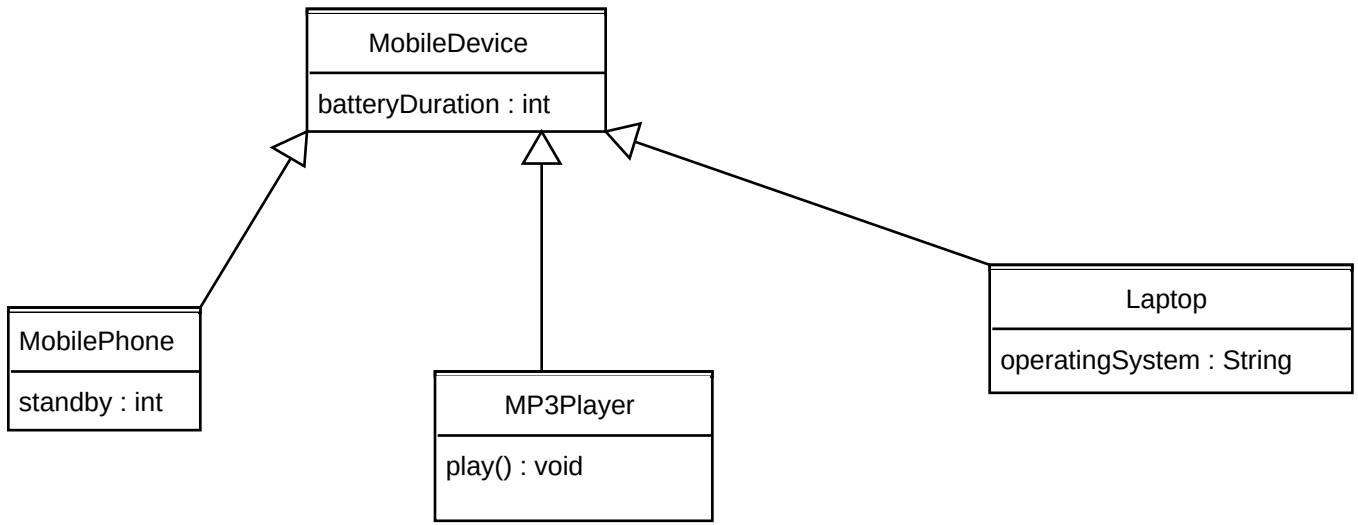
Composition

- **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).



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 behavior 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"



Implementing Generalizations

- 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 modeling
- 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

Modeling 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

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…

Common Domain Modeling 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
- Optimizing 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

(Suggested) Readings

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, H. Obbink, J. Stafford. The Past, Present and Future of Software Architecture. IEEE Software, March/April 2006.

Suggested Readings

- I. Sommerville. Software Engineering, Eighth Edition, Addison-Wesley 2007.
  - Chapter 14 on Object-oriented design
- B. Meyer. Applying `design by contract'. IEEE Compute, 25(10):40-51, 1992.

Summary

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