COMP70083 Principles and Practice of Programming

Part II: Object Oriented Design with UML Modelling and C++

William Knottenbelt

19 October 2024

Acknowledgements

I would like to thank:

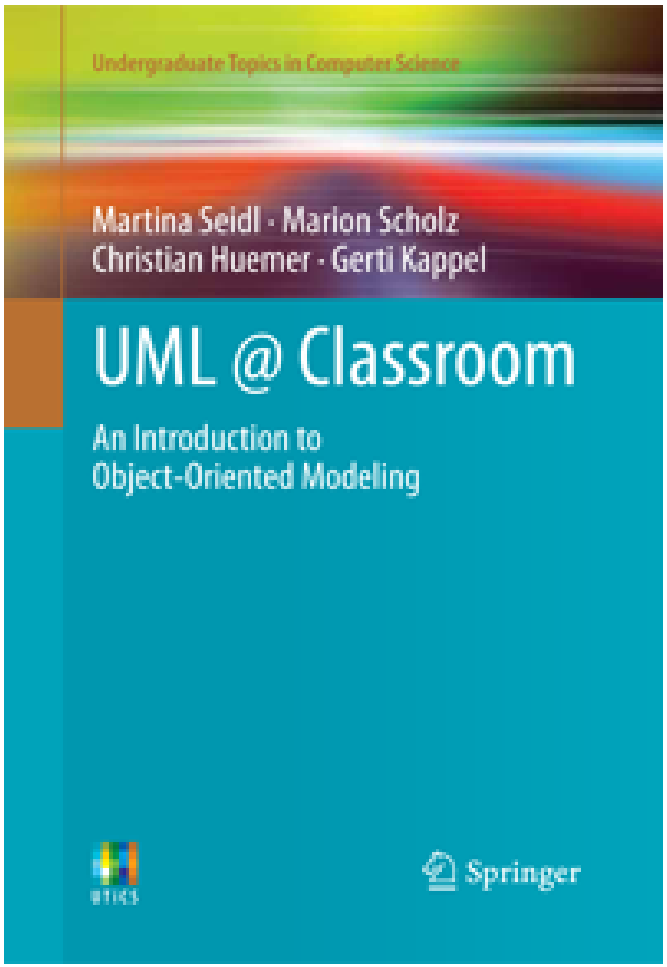
- Prof Sophia Drossopoulou for earlier versions of the notes
- PhD Teaching Scholar Titus Buckworth for help in preparing the slides

Objectives

This module part is concerned with the design and implementation of software. You will learn how to:

- Visualise the design of real world systems in an *object-oriented* manner using the *Unified Modelling Language* (UML)
- Map UML class diagrams onto C++ code using appropriate C++ language features

Recommended UML Textbook



- A concise UML primer
- Available from `https://link.springer.com/book/10.1007/978-3-319-12742-2` (use institutional login if necessary)
- Deals very satisfactorily with design aspects
- Implementation/code generation is not covered in as much detail in the book but we will cover it in lectures

COMP70083 Part II and III Structure

- **Lectures**: Monday 09h00–11h00 and Thursday at 16h00–17h00; lectured by William Knottenbelt and Tom Crossland.
- **Lab support:** Thursdays 09h00-10h00 by Tom Crossland.
- **Tutorials**: Thursdays 17h00–18h00
- **Assessments**: 1 group coursework, 2 lab exercises, Exam (90 mins)

Exam

- A typical exam question will present a description of a real-life scenario and hence require you to:
  - **Formulate an appropriate object-oriented design** (as a UML class diagram) by understanding the (a) key kinds of entities that make up the system and (b) the relationships between them
  - **Translate your object-oriented design into C++ code**, that is class declarations, class implementations and test functions.

What is Object Orientation

- In **procedural** programming we tackle programming problems by: (a) considering what tasks and subtasks we can break down the problem into and (b) working out how data needs to be passed from one (sub)task to another in order to solve the problem.
- In an **object-oriented** paradigm we solve programming problems by: (a) considering what kinds of **objects** are present in the system, (b) how they relate to each other, and (c) how they need to communicate in order to solve the problem.

Spot the Objects

Spot the Objects

Key Principles of Object Orientation

1. **Classes** – Blueprint for objects: e.g. Person
2. **Objects** – Concrete class instance: e.g. Will
3. **Encapsulation** – Protection of internal state: e.g. who should be allowed to change a Person's name?
4. **Messages** – receiving and sending requests for services e.g. Will asks you to do your coursework: you choose whether to do it
5. **Inheritance** – subclasses (derived classes) inherit properties from superclasses (base classes): e.g. a Professor inherits properties of a Person (name, birthday etc.)
6. **Polymorphism** – the ability to adopt different forms: e.g. an object could be a Person and an Employee at the same time

Key Principle of Object Orientation #1: Classes

- A class is a blueprint for a set of objects that share a **common structure**, and a **common behaviour**.
- A class comprises:
  - **data members** (aka attributes, fields, properties etc.): each object has its own copy of the local data members which jointly comprise the **state** of the object
  - **member functions** (aka operations, methods etc.): denote the services that objects offer to their clients. Applicable to all objects of the class.
- Both data members and operations have **types**.

Key Principle of Object Orientation #2: Objects

- Objects are instances of classes.
- Often correspond to real-world entities.
- An object has **state**, exhibits some well-defined **behaviour**, and has a **unique identity**.

Key Principle of Object Orientation #3: Encapsulation

- The idea of encapsulation is to **control access** to data members and some member functions so that the internal state is protected from arbitrary interference.
- Modification of state should take place through **authorized** member functions. Thus we guarantee that the internal state of an object is maintained in a consistent manner.
- Encapsulation is enforced through **visibility modifiers** (aka **access modifiers**), e.g.:

private	access for instances of the same class only
public	access for instances of any class
protected	access for instances of the same class and instances of derived classes

Key Principle of Object Orientation #4: Messages

- Objects communicate with each other via **messages**. A message is a request to execute a particular member function.
- Objects can **receive** messages invoking their member functions. An object that receives a message is called the **receiver**.
- A receiver of a message decides whether and how to execute the corresponding member function.
- Provided they are suitably authorised to do so, objects can **send** messages invoking the member functions of other objects.

Key Principle of Object Orientation #5: Inheritance

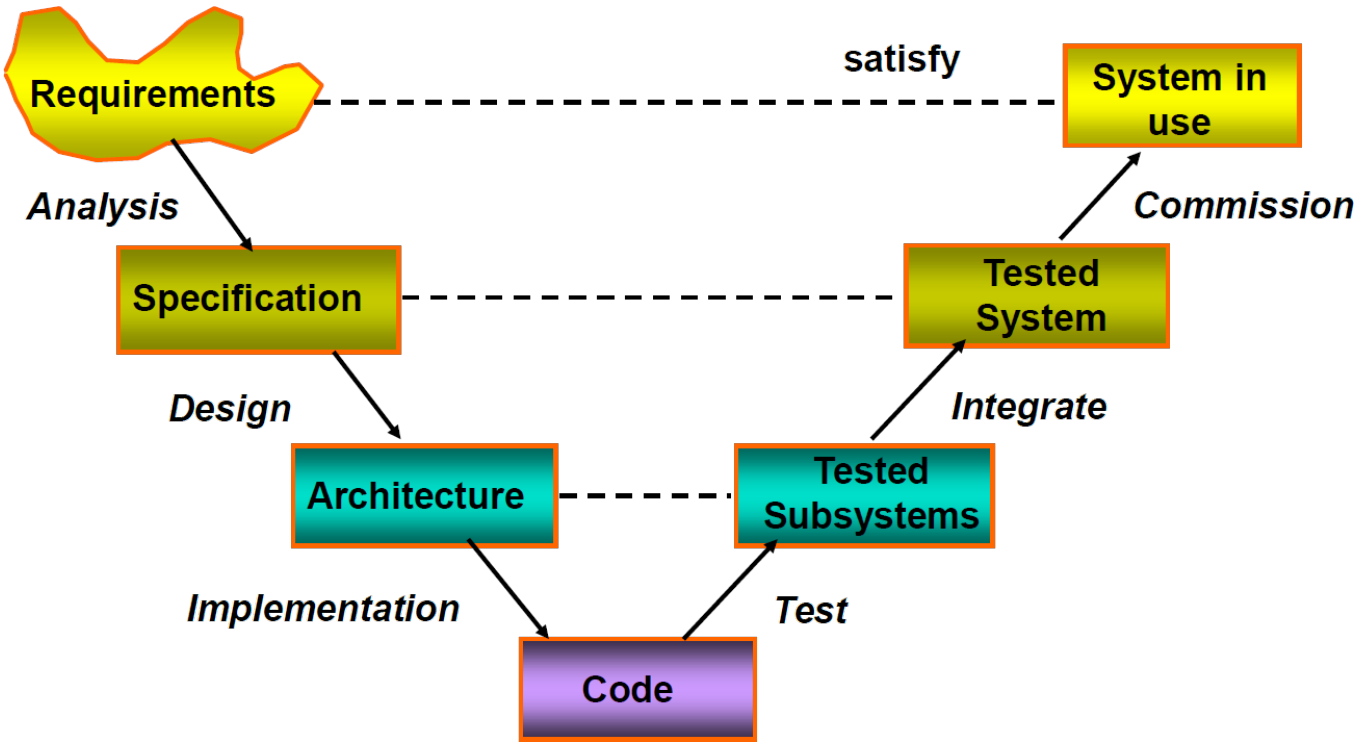
- Inheritance provides a mechanism for deriving new classes from existing ones in a way that promotes elegance and reuse
- Existing class is called the parent class, the base class, the general class or the superclass
- New class is called the child class, the derived class, the specific class or the subclass
- Derived classes **inherit** all visible attributes and operations of the superclass. In addition they may (a) **define additional attributes or operations** and (b) **override** the implementation of inherited operations

Key Principle of Object Orientation #6: Polymorphism

- Attributes and member function parameters can be polymorphic in the sense that they might refer to different classes at run-time.
- Member functions and operators can be polymorphic in the sense that they operate on different parameter types.
- Polymorphism is most commonly encountered in the context of inheritance hierarchies. Thus a pointer to an object of a particular superclass e.g. `Person` may actually turn out to be a pointer to an object of a subclass, e.g. `an Employee` or a `Student` at run-time.
- Type coercion (casting) can be used to convert parameters to enforce correct polymorphic behaviour.

Why use a model

The software production process is complex:



Key Question

How are many stakeholders with different backgrounds going to communicate unambiguously with each other about the system?

Why use a model (cont.)

- To help, we need a **model** (cf. an architectural model for a building) that stakeholders can read, interpret and implement.

- A model allows us to **describe systems clearly, efficiently and elegantly**. Models deliberately **do not include all details** of the system, but only those relevant to a **particular perspective or purpose**.

What is UML

- Unified Modelling Language or UML is a sophisticated general-purpose modelling language developed over the last quarter of a century or so.
- The main purpose of UML is to describe different aspects of systems using a standardised graphical notation.
- UML is standardised by the Object Management Group, see `https://www.omg.org/spec/UML` for the latest standard.

UML Diagram Types

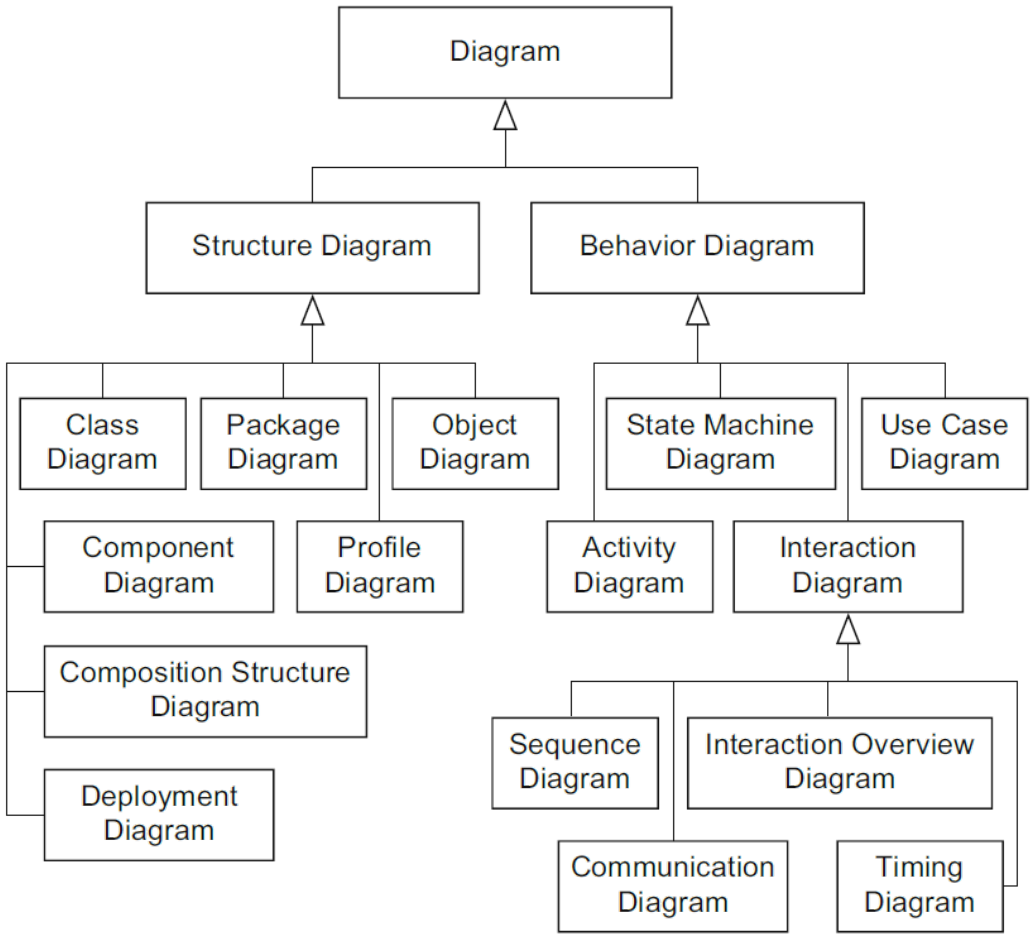


Figure: UML Diagram Types

UML Diagram Types

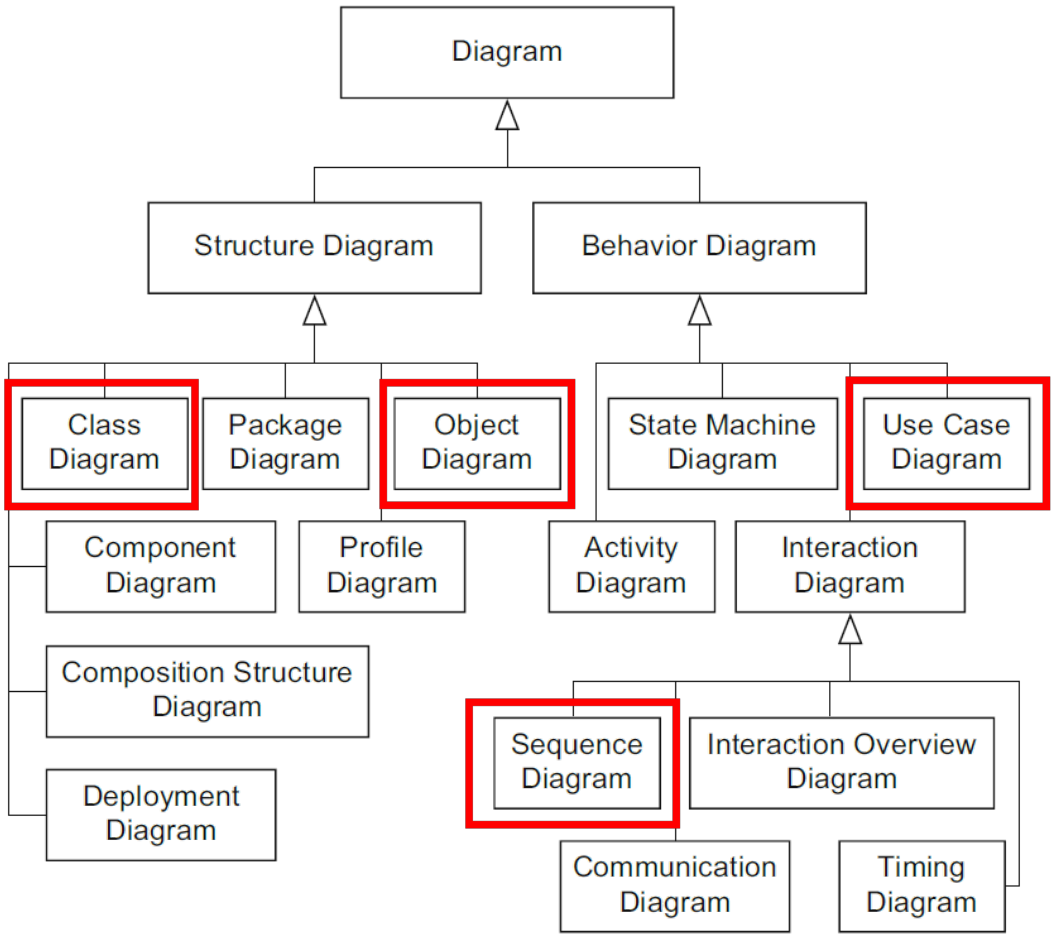


Figure: UML Diagram Types

Relevant UML Diagram Types

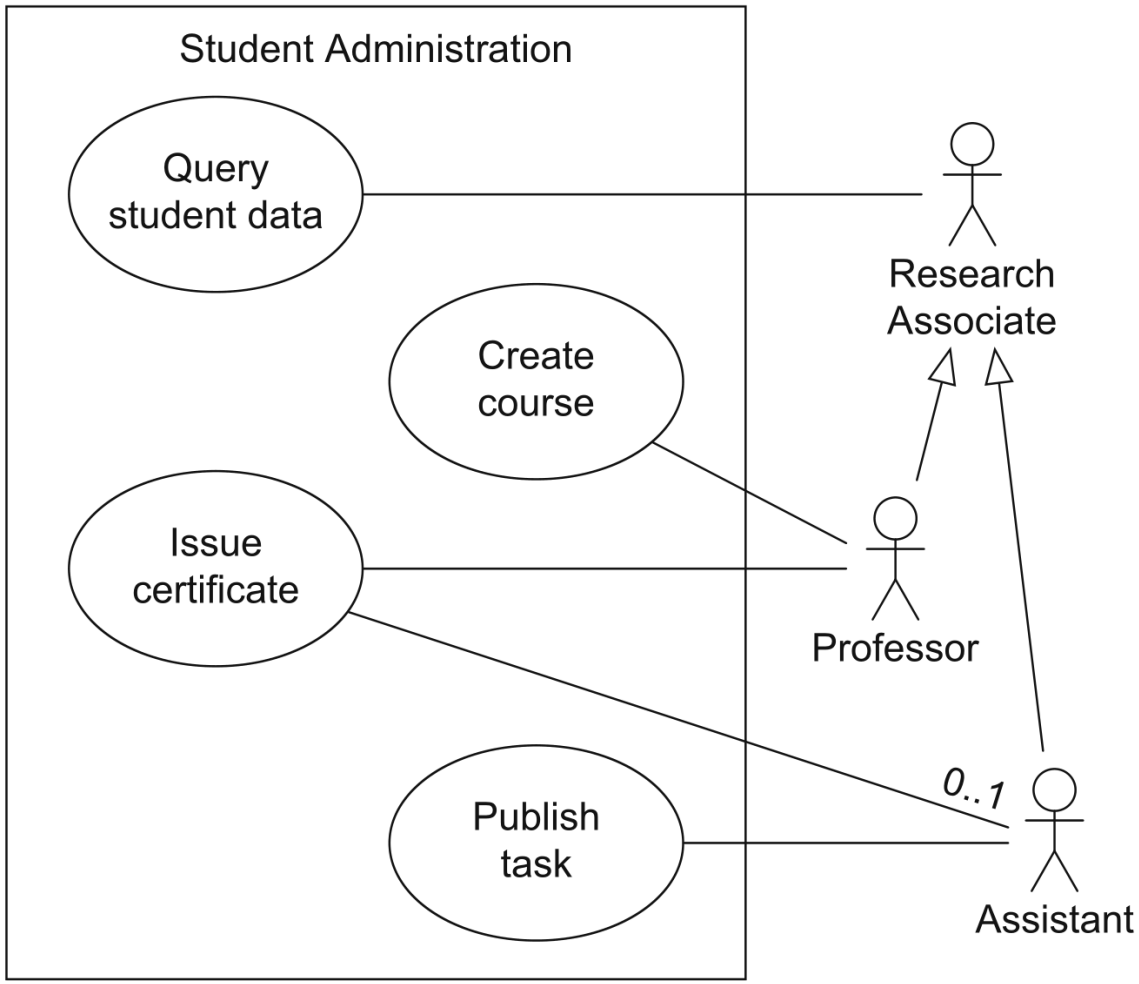


Figure: Use Case Diagram

Relevant UML Diagram Types

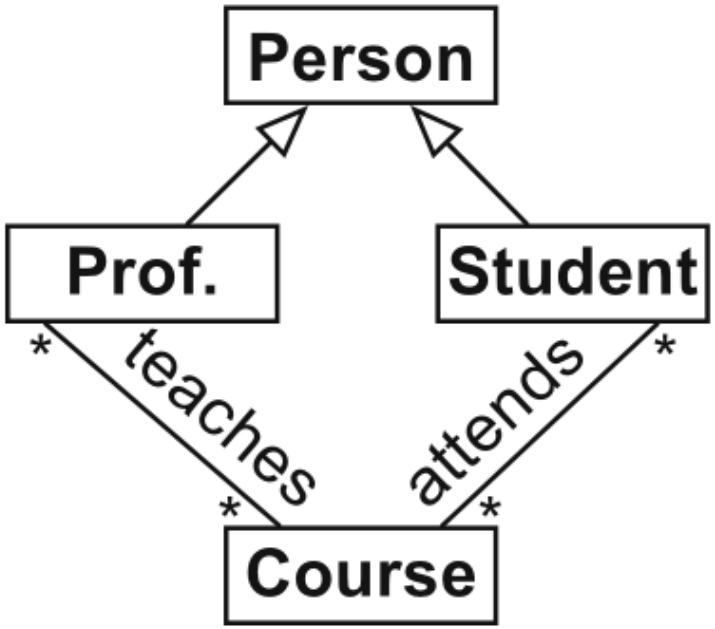


Figure: Class Diagram

Relevant UML Diagram Types

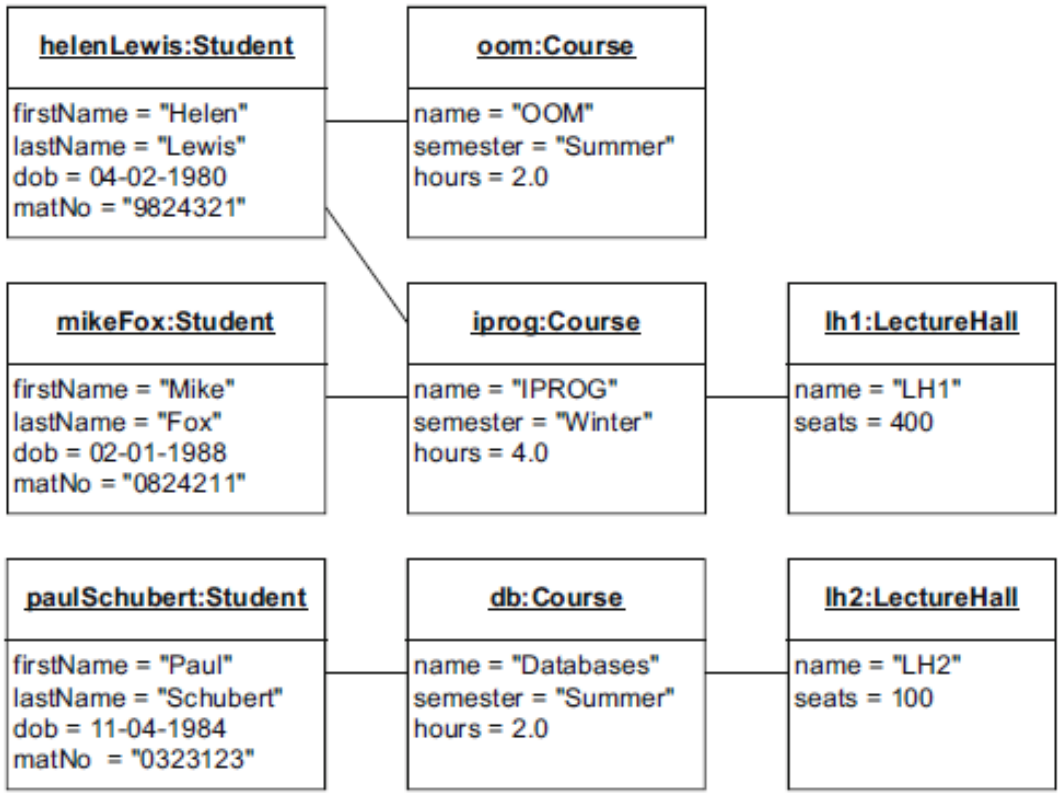


Figure: Object Diagram

Relevant UML Diagram Types

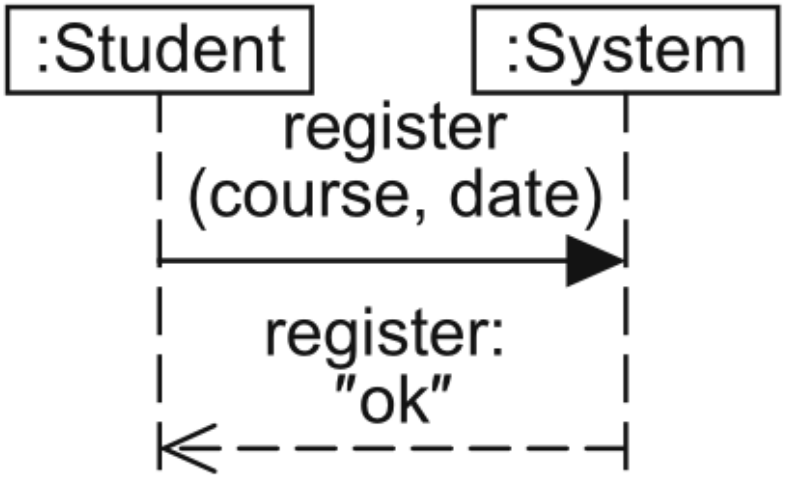


Figure: Sequence Diagram

Class Diagrams: Key Components

A class diagram may contain:

- Classes, with attributes (data members) and operations (member functions)
- Binary or N-ary associations between classes
- Aggregations between classes
- Association classes.
- Inheritance between classes.

Class Diagrams: Example

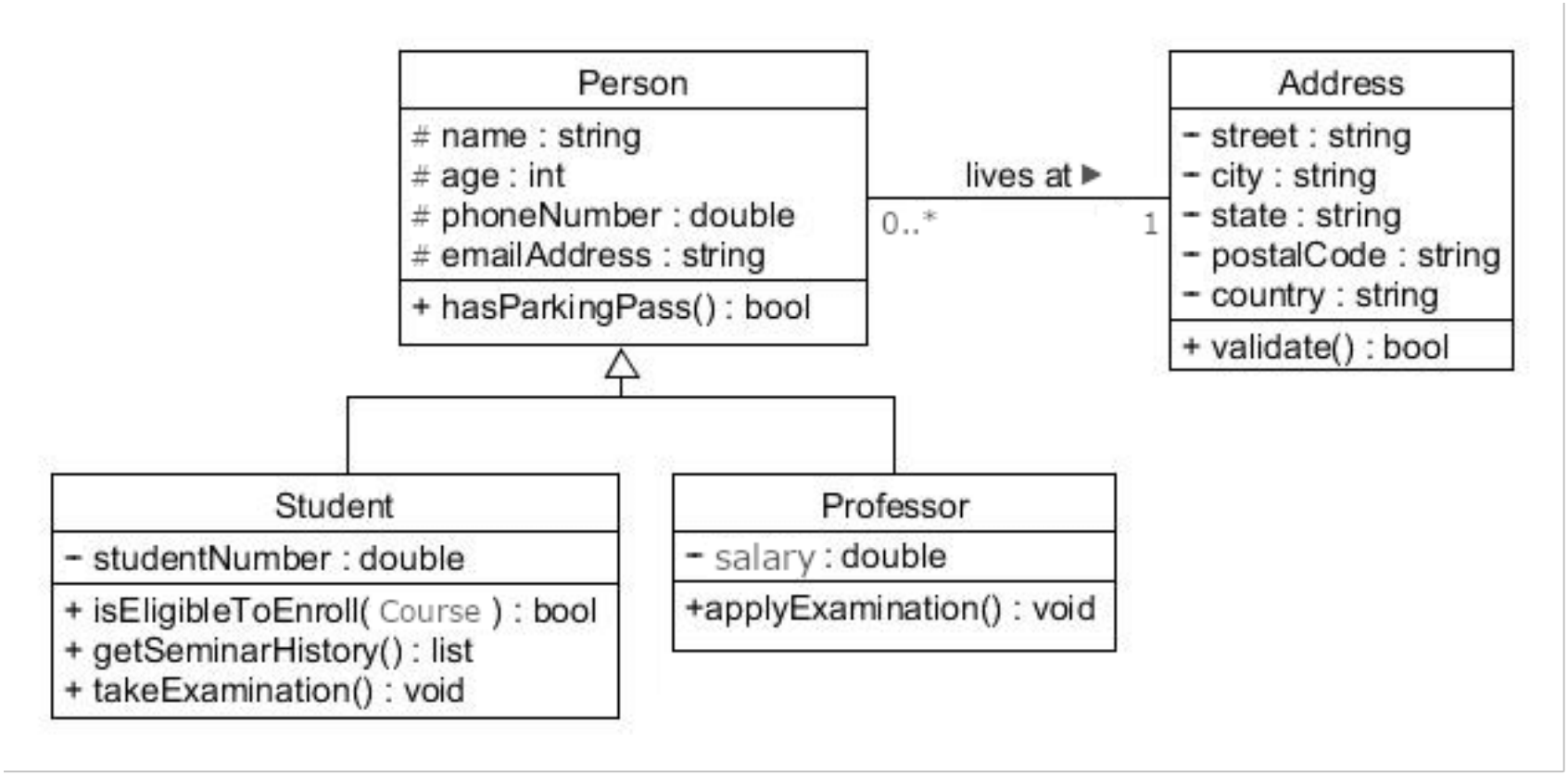


Figure: Class Diagram Example

Class Diagram: Visibilities

Name	Symbol	Description
public	+	Access by objects of any classes permitted
private	-	Access by other classes forbidden
protected	#	Access by objects of the same class and its subclasses permitted

About Operations

- Operations are functions that may be applied to objects.
- Each operation has its target object (receiver) as an implicit argument.
- Behaviour of an object depends on its class.
- Like ordinary functions, operations take parameters of a certain type, and return a result of a certain type.

Important Note

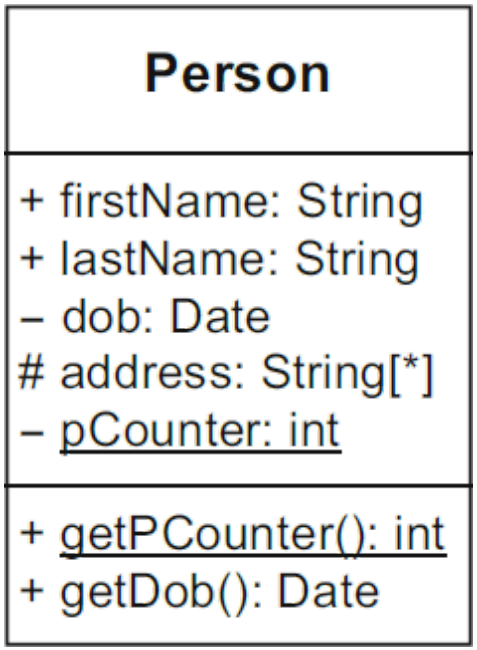
It is not necessary to pass a parameter to an operation if the object has a way to access or calculate the information.
Example: When an amount is paid into a bank account, then the balance of the account increases by the amount.

More about Operations

- Some operations involve two or more objects. In such cases a decision must be made as to which class should contain the operation, considering:
  - Who does the processing
  - Who holds the information
  - Who decides the outcome of the operation
- **Example 1**: When a person inflates a tyre, their heart rate increases to 110bpm, their respiration rate increases to 18 breaths per minute, their temperature goes up 2 degrees centigrade, and the tyre's pressure goes up by 3 psi.
- **Example 2**: When a person inflates a tyre, the tyre's pressure goes up by 3 psi, its temperature increases by 10 degrees centigrade, and if the maximal tyre pressure is reached, the tyre explodes

Static Attributes and Operations

- It is possible to have attributes which belong to a class as a whole rather than each separate instance of the class
- These are known as **static attributes** or **class attributes** and are underlined in the UML diagram.
- **Static operations** or **class operations** can be used even if no corresponding instances of the class have been created.



Associations

- Associations denote a relationship between concepts (classes).
- Associations often correspond to phrases including: physical location (next to, part of, contained in), directed actions (drives), ownership (has, part of), or satisfaction of some condition (works for, married to, manages).
- There can be multiple associations between the same classes.

Binary Associations

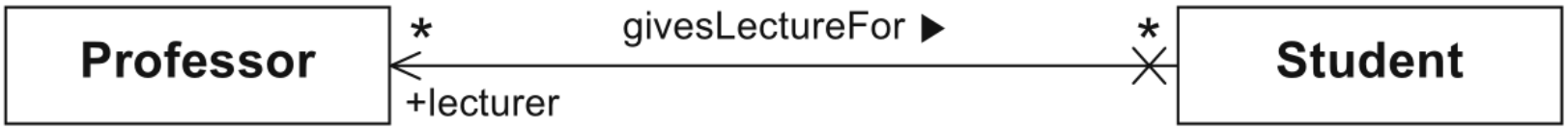


Figure: Associations between classes

- Notice how the **reading direction** appears next to the association name as a solid triangle (◀ or ▶).
- The symbols on the line indicating an association between Professor and Student relate to **navigation directions**. The arrow indicates that Students can access the visible attributes of Professors. The cross indicates that Professors cannot access any attributes of Students.
- We do not insist that you consider navigability in this course.

Binary Associations: Examples

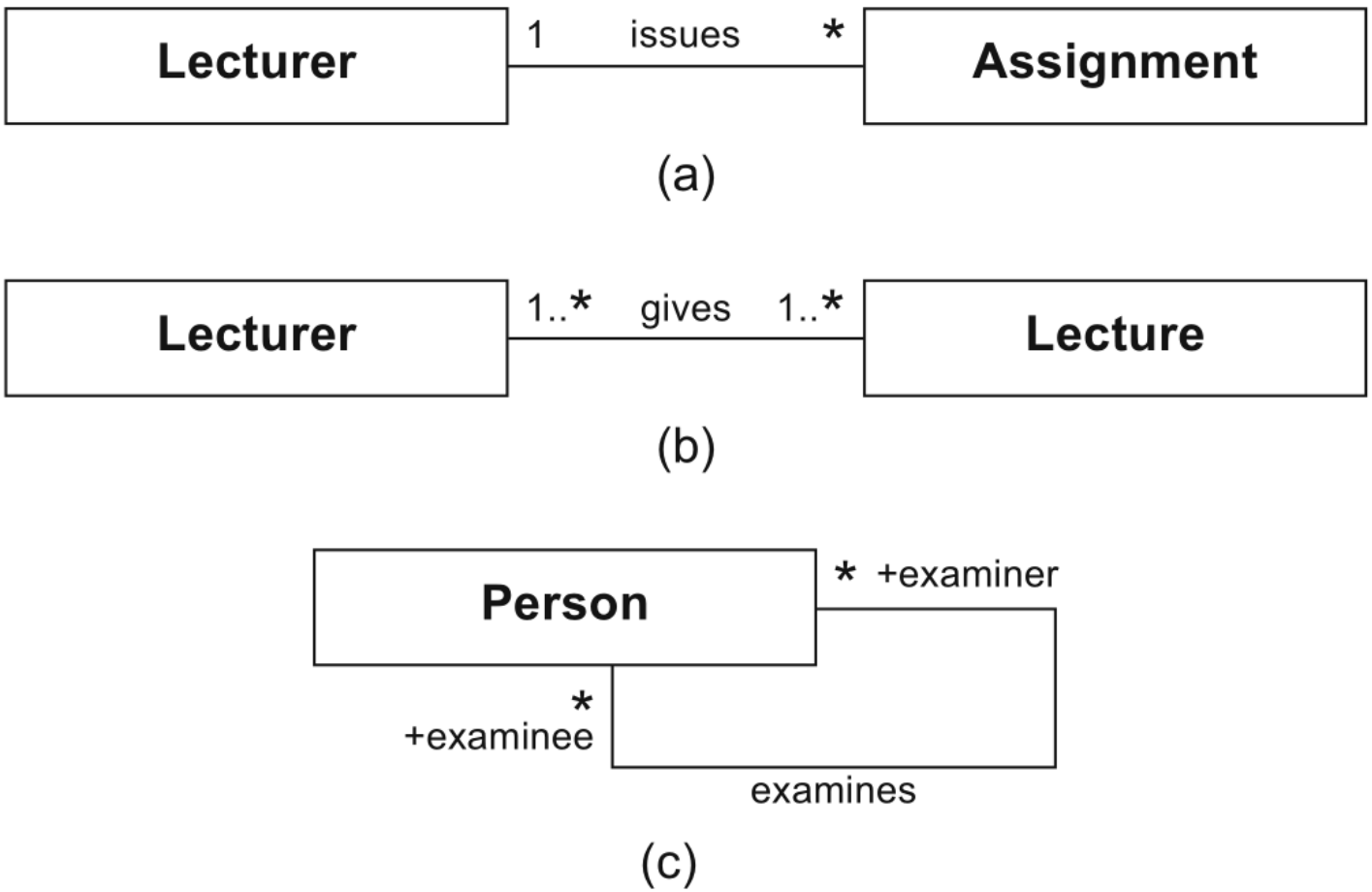
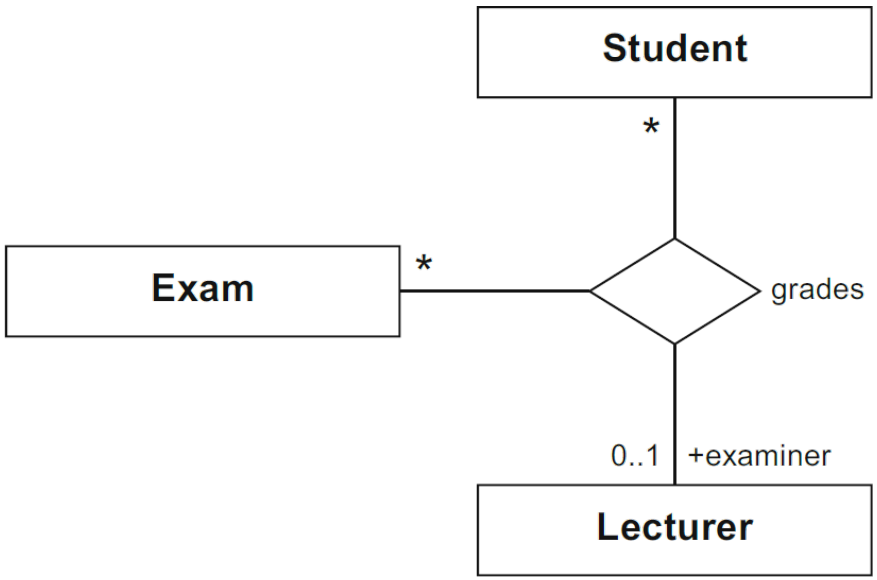


Figure: Binary association examples

N-ary Associations

- Model situations where more than two partner objects are involved in a relationship
- Represented by a labelled diamond and connected to partner objects by undirected edges
- Navigation directions are not possible but multiplicities are.



Associations vs Attributes

- Associations exist between classes
- Attributes have simple type
- Therefore:
  - Being the mother of a person is represented by an association
  - Students having an age is represented by an attribute

Important Note

When we translate into C++, both associations and attributes might be represented by fields in C++ classes. This decision does not concern the design phase and is not reflected in UML.

Associations vs Operations

- Associations represent persistent relationships
- Operations may establish or break associations

Which of the following are operations and which are associations?

- being married to
- applying for a driver's license
- meeting a colleague
- being friends with
- owning a car
- riding a bicycle
- a plane landing at airport
- marrying somebody
- divorcing somebody
- belonging to a club
- working for a company
- selling a house
- owning a house
- moving into a flat

Aggregations

- An aggregation is a specialisation of an association denoting a 'part of' (or 'has a') relationship between the objects of the respective classes.
- There are two different types of aggregation:
  - **Composition** is a type of aggregation which links the lifetimes of the aggregate and its parts. In other words: (a) The parts cannot exist if the aggregate no longer exists and (b) An entity can exist as part of only one aggregate
  - **Shared Aggregation** under which the aggregate and the parts have independent lifetimes.

Exercise

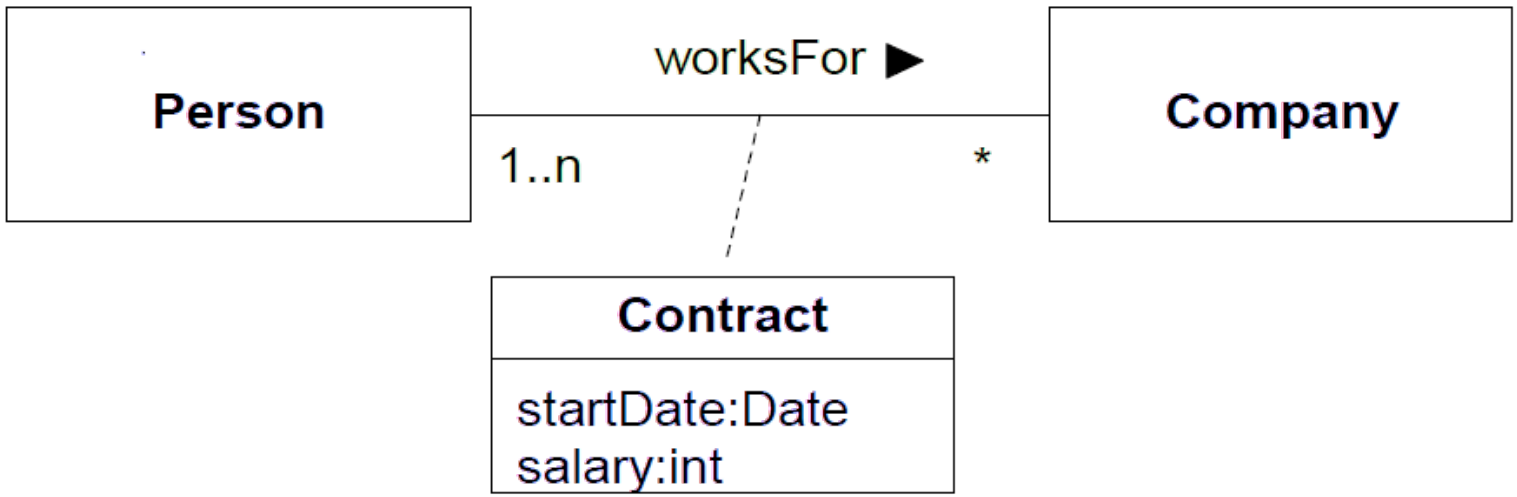
Write a UML class diagram to represent that a workstation consists of the following:

- At least one monitor
- One system box, which consists of
  - a chassis
  - a CPU
  - several RAMs
  - a fan
- possibly a mouse
- a keyboard

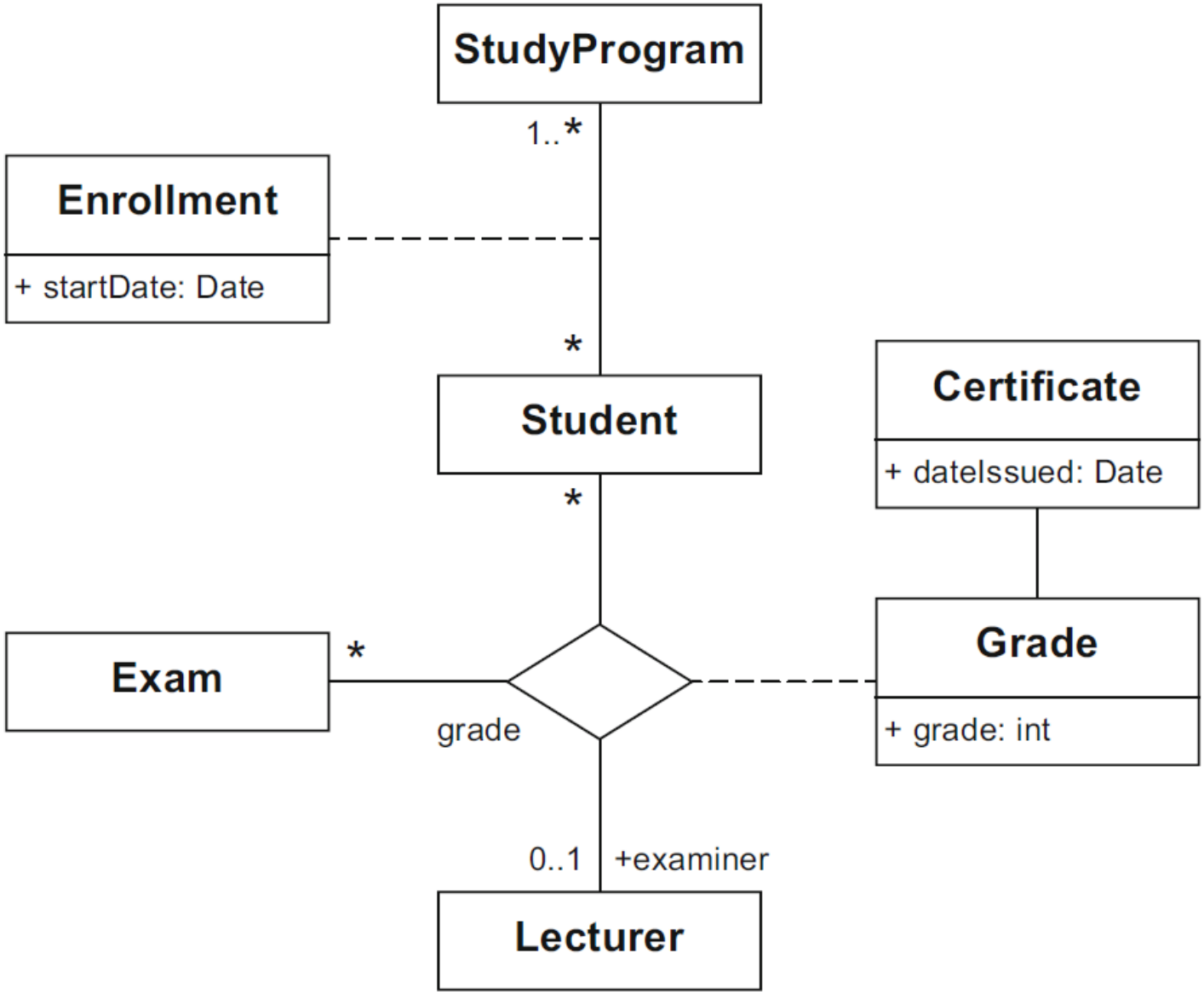
Association Classes

Associations may have their own properties.

- An association class is an element that has both association and class properties.
- Occurs frequently in many-to-many associations because it is difficult to position the property at either end of the association



Association Classes: Another example



Exercise

Develop a UML class diagram to express the following:

- A user may be authorized on several workstations
- Several users may be authorized on one workstation
- Each authorization has a priority (a number), a start session which is run when user logs on, and a directory which is the default directory when the user logs on.
- A directory has a name, and is contained in another directory.

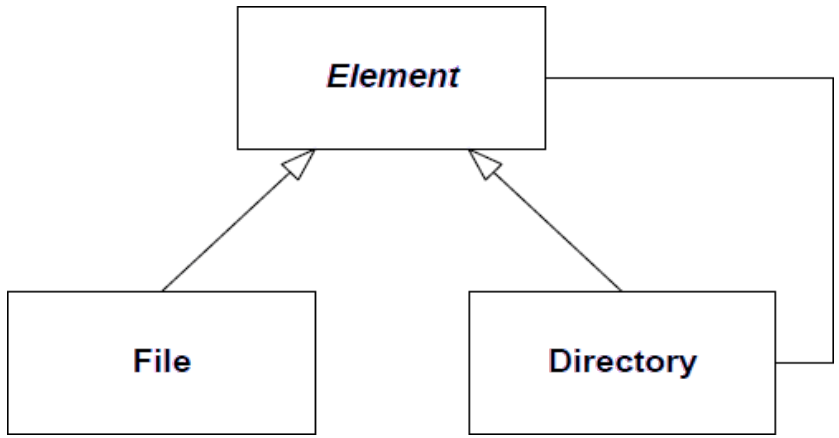
Inheritance

Generalisation is a relationship between a class and its more refined (specialised) versions

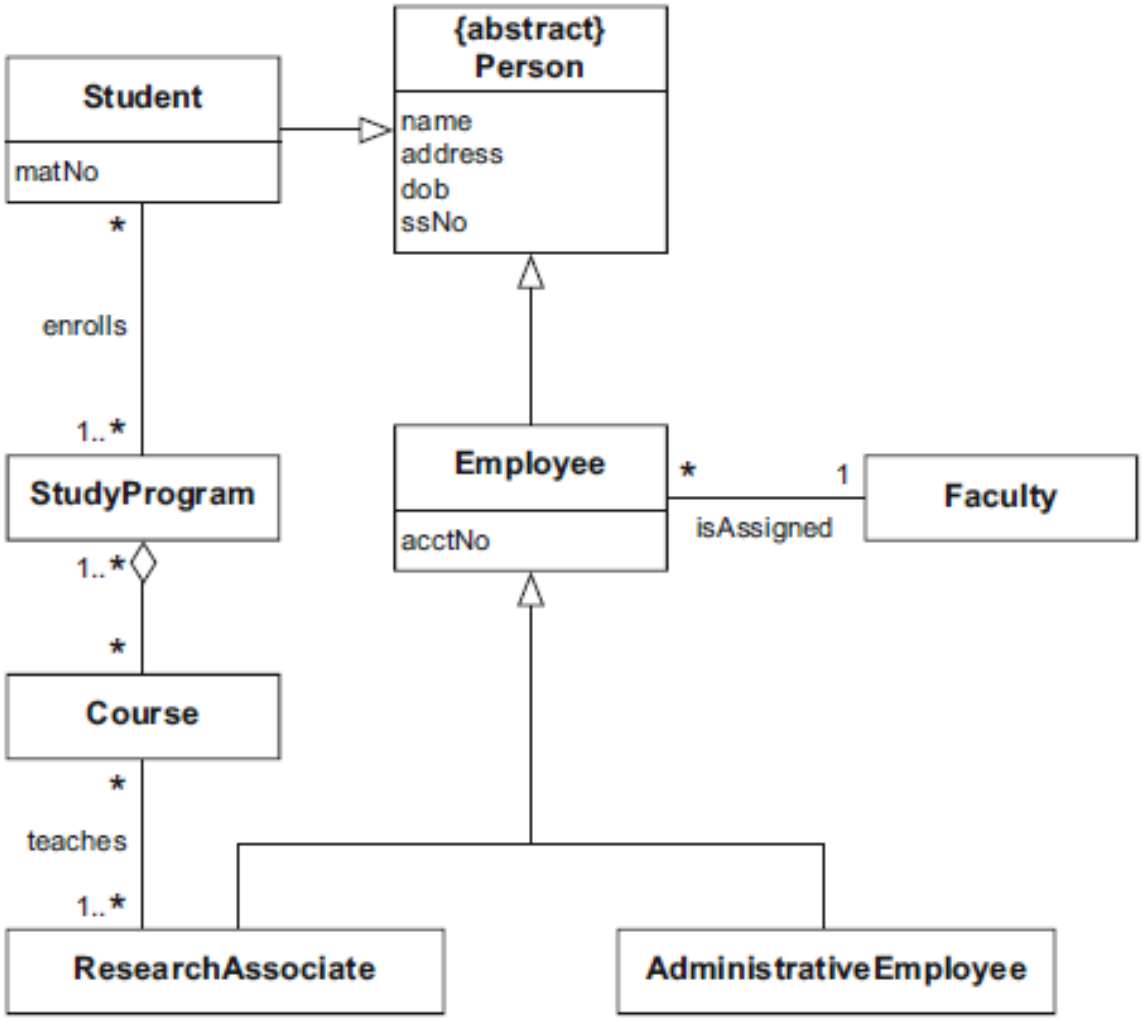
- Subclasses **inherit** all features of their ancestors (i.e. attributes, operations, aggregations, and associations)
- Subclassses can **override** operations in superclasses
- Generalisation is effective at capturing similarities efficiently

Important note

A superclass should not include elements which are only applicable to some of its subclasses

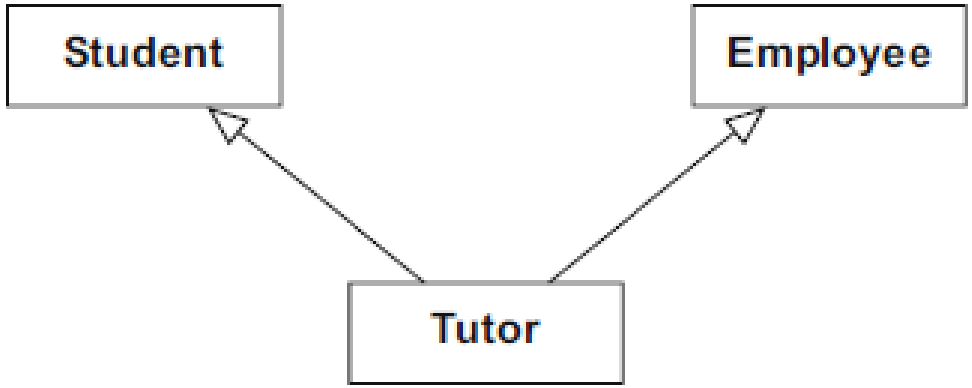


Inheritance: Example



Multiple Inheritance

It is possible for a class to have multiple superclasses.



Basic Exercise

Express the following in a UML class diagram:

- Employees may be managers or executives.
- Employees have a salary, and may be given a payrise.
- When a manager is given a payrise, their salary is incremented by 100, but
- When an executive is given a payrise, their salary is incremented by 300.

Advanced Exercise

Develop a UML diagram to model the following scenario:

- A library offers two kinds of items for loan to its users: books and periodicals.
- All items have a title and a shelf location number.
- Each book may be borrowed for a certain maximum number of days that is different for each book. All periodicals may be borrowed for the same maximum number of days; this period is set by the library, and is currently 5 days.

What's Next

Next we will take a look at some C++ language features which are useful when translating UML designs into C++ code.