CS5030

Software Design - Modelling Structure

Learning objectives

- On completing this lecture and associated reading, you should
  - Be aware of UML diagrams for modelling structure at different levels of abstraction
  - Be familiar with the basics and more advanced features of UML class diagrams
  - Be able to apply structural modelling in practice

Software models

- From requirements to implementation

- At the design stage, typically
  - Structure
  - Behaviour

Modelling structure with UML

- Software architecture
  - Component diagram, deployment diagram
- Software design
  - **Class diagram**, object diagram
- Software
  - Package diagram
- Internal structure
  - Composite structure diagram

Object-oriented design

- Object
  - “an abstraction of something in a problem domain, reflecting the capabilities of the system to keep information about it, interact with it or both” – Coad and Yourdon (1990)
  - “a discrete entity with a well-defined boundary that encapsulates state and behaviour; an instance of a class”

    – UML Reference Manual
- System
  - a collection of interacting objects

Object

- Identity
  - Unique to each object
- State
  - Values of its attributes and the relationships it has to other objects at a particular point in time
- Behaviour
  - Operations that can be invoked on the object

Object – state and behaviour

- State
  - Some attribute values may change over time
  - Others will stay the same
- State can affect behaviour
- Behaviour can affect state
  - State transition

Examples

barbaraLiskov : Person	
name	“Barbara Liskov”
dob	07/11/1939
address	“MIT”
job	“Computer Scientist”
updateJob()	
calculateAge()	
getAddress()	

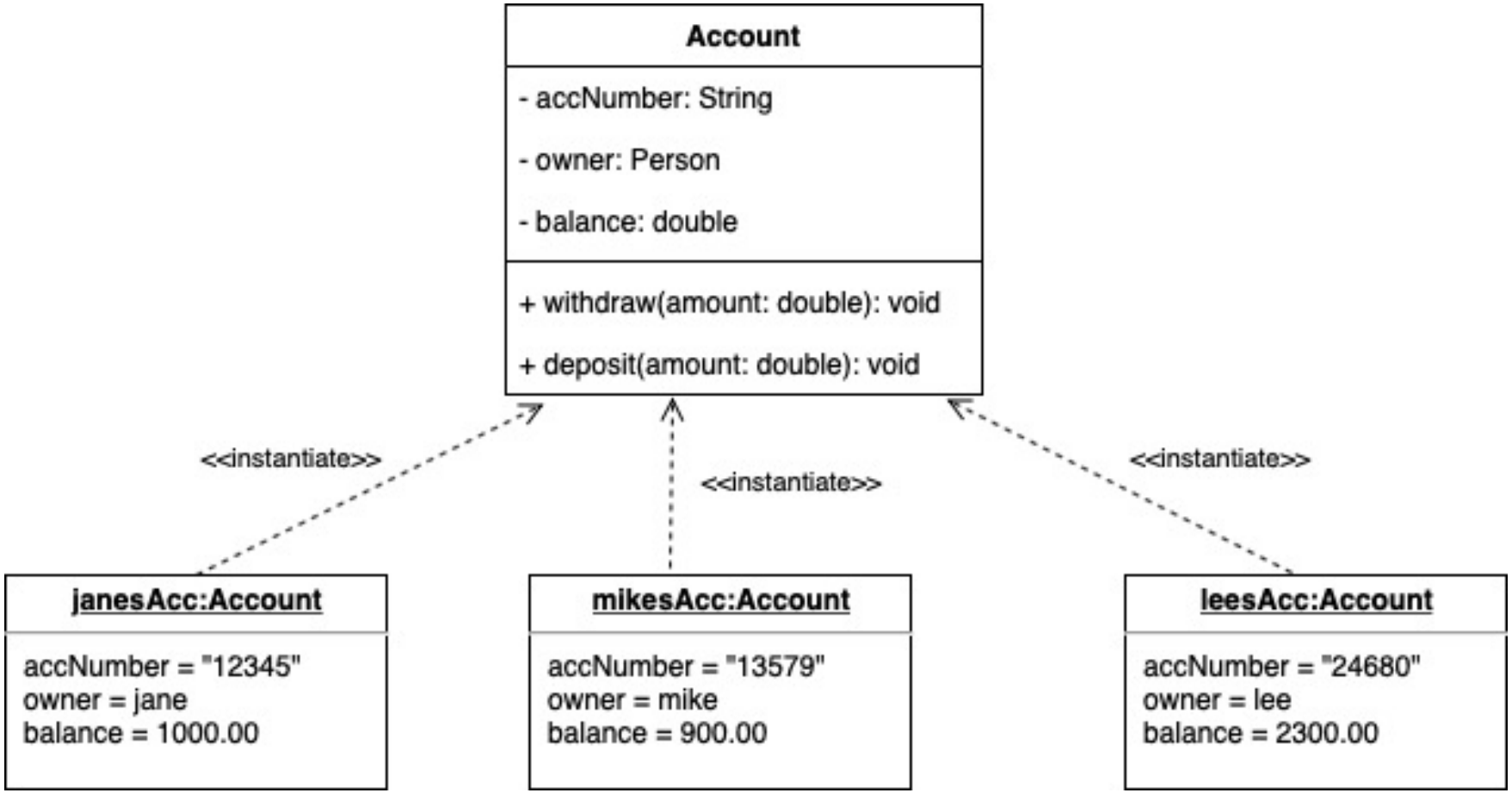
myCurrentAc : Account	
number	12345678
balance	1500.00
owner	barbaraLiskov
getBalance()	
withdraw()	
deposit()	

Methods / operations don’t show all details

Class

- A way of grouping objects with similar structure and behaviour
  - A template
- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics
  - But not necessarily the same attribute values
- Each object is an instance of exactly one class

Classes and objects - example



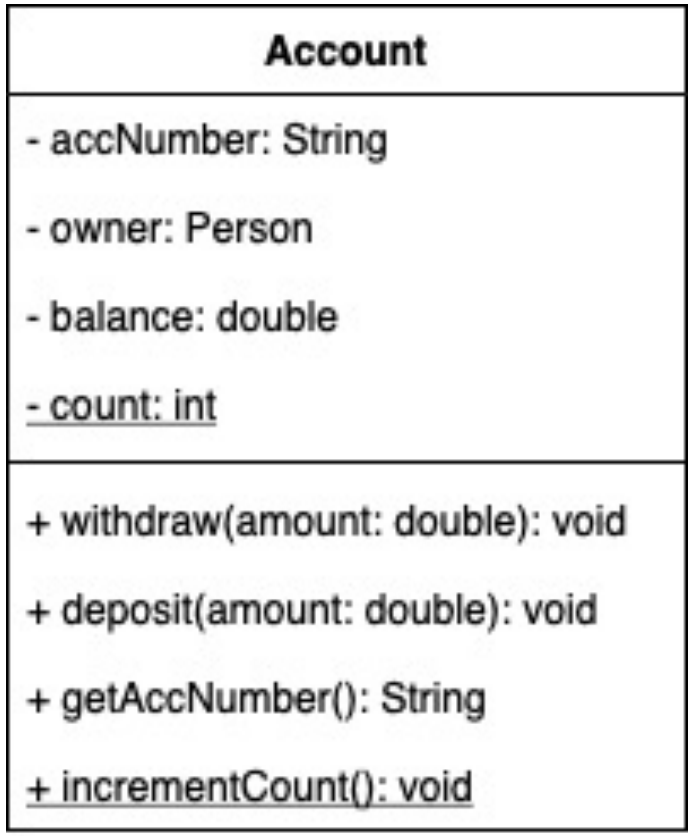
Class diagram

- Shows
  - Class name
  - Attributes and methods / operations
    - Scope, visibility
- Scope of attributes and methods
  - Object (no adornment)
  - Class (underlined)
- Visibility
  - Accessibility of element
- Derived attribute – denoted by /

Visibility

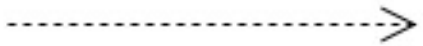
Modifier	Visibility	Semantics
+	public	Accessible to any class
-	private	Accessible only within the same class
#	protected	Accessible only within the same class or its subclasses
~	package	Accessible to any class in the same package

Example – expanded *Account*



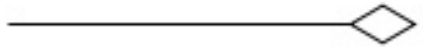
Relationships between classes

- In increasing order of strength:

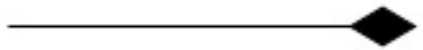


Dependency

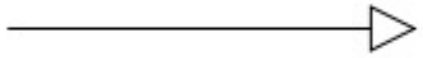
Association



Aggregation



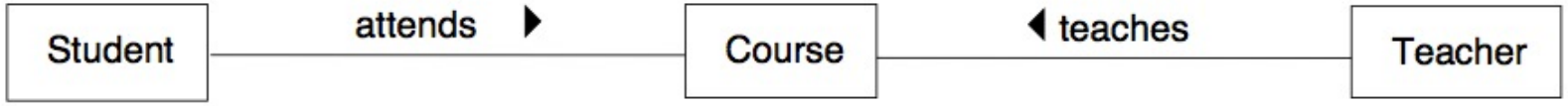
Composition



Generalisation / inheritance

Association

- A design-time relation between classes
- Specifies that at runtime instances of those classes have a link and be able to request services from one another
- Associations can have a name to describe the nature of the relationship
- An additional arrowhead can be used to indicate the direction in which to read the name



Syntax for association

- An association name
  - Verb phrases, indicating an action that the source object performs on the target object
- A role name
  - Names the class on one or both ends of the association, indicating the role that objects play when they are linked by this association
- Multiplicity
  - Constrains the number of objects of a class that can be involved in a particular relationship at any point in time
- Navigability
  - Direction from an object of the source class to one or more objects of the target class

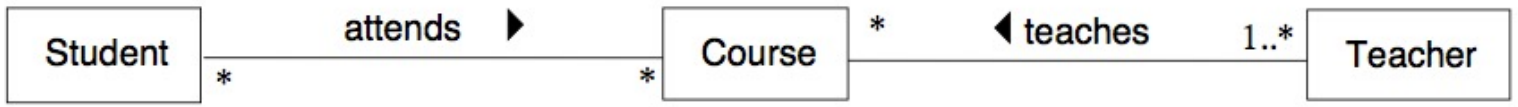
Association roles

- Objects can play a specific role in a relationship
- Roles are shown as a name placed near the end of an association

Associations: multiplicity

- Multiplicity on an association end indicates the possible number of instances from that end that can participate in the association at runtime

0..1	Zero or one
1	One only
0..*	Zero or more (usually truncated to *)
1..*	One or more
n	Only n (where n>1)
0..n	Zero to n (where n>1)
1..n	One to n (where n>1)



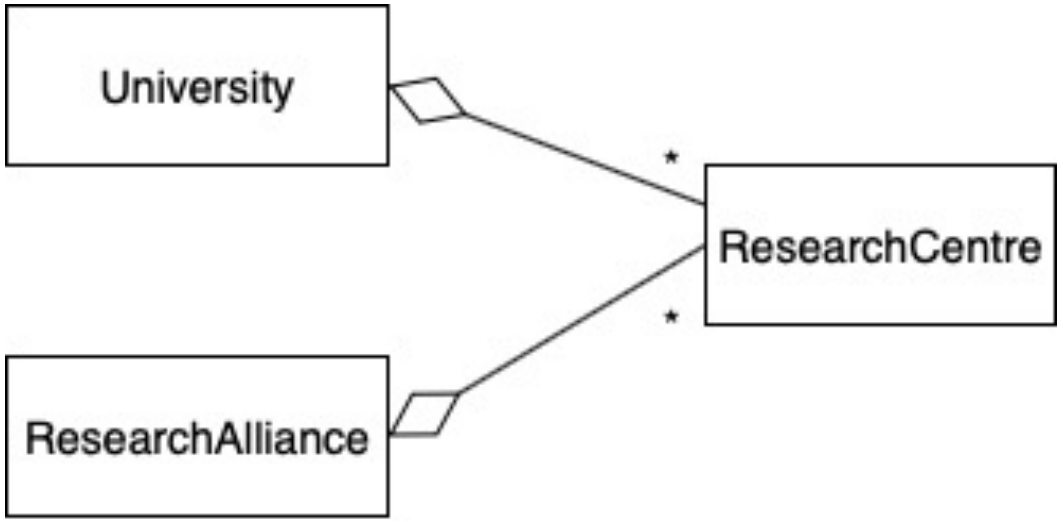
Associations: navigability

- The direction of navigability or message flow of the association may be specified by adding an open arrowhead
  - No arrowheads means that the association is bidirectional

Aggregation

- A special kind of association indicating the existence of a "whole-part" relation between two classes
  - Object of one class owns but may share objects of another class
- The whole end is marked with an empty diamond
- As a special form of association, all properties and adornments that apply to associations also apply to aggregations

Aggregation - example



Aggregation semantics

- A relatively weak form of “whole-part”
  - One object (the whole or the aggregate) uses the services of another object (the part)
- The aggregate can sometimes exist independently of the parts, sometimes not
- The parts can exist independently of the aggregate
- The aggregate is in some sense incomplete if some of the parts are missing
- It is possible to have shared ownership of parts by several aggregates
- Aggregation is transitive

Composition

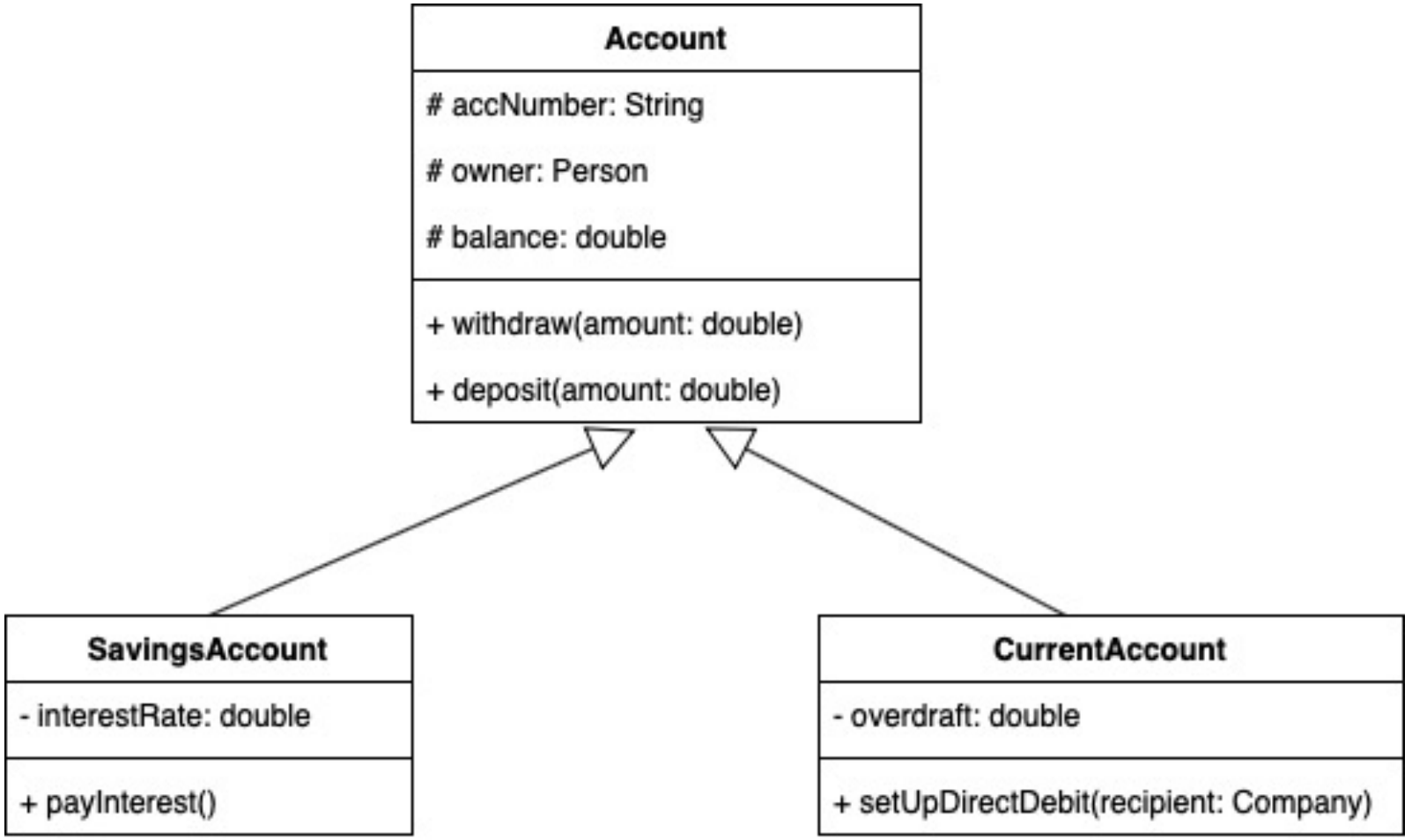
- A strong form of aggregation
  - A unique “whole” (“composite”) has explicit responsibility for the creation and destruction of the part objects
  - Represented by a filled diamond
  - The composite must exist before the parts come into existence
  - If the composite is destroyed, its parts will be destroyed too
  - Each part object can only be owned by a single composite object

Composition - example

Generalisation

- Generalisation is a relationship between a general element and a more specific element, where
  - The specific element is consistent with the general element
  - but contains more information
- Also called inheritance

Generalisation example



Generalisation in UML

- Can be interpreted as “is a kind of”
  - We can use the more specific element anywhere the more general element is expected without breaking the system

- Subclasses have all the attributes, operations and relations of the superclasses, and may specialise or extend them

Using inheritance (1)

- Beware of implementation inheritance
  - when a class inherits from another class simply to reuse parts of its behaviour
- Beware of multiple inheritance
  - Complications such as the diamond problem
  - Not supported by many programming languages
- Superclasses should know nothing about subclasses

Using inheritance (2)

- Factor commonality as high as possible in the class hierarchy
  - If ClassB and ClassC both inherit from ClassA, and they both need behaviour X, then X should be implemented in ClassA
    - If only ClassB needs X, then X belongs in ClassB
  - The higher a method is, the more reusable it becomes
  - However, this does not mean that you should place every single method in the root class

Good design – coupling and cohesion

- Coupling
  - describes the degree of interconnectedness between design elements
  - Aim for low interaction
- Cohesion
  - is a measure of the degree to which an element contributes to a single purpose
  - Aim for high cohesion

Criteria for good design

- Design clarity
- Do not over-design
  - Abstraction is key
- Clear inheritance hierarchies
- Keep interactions and operations simple
- Clear separation of classes

Key points

- In object-oriented design, a system is a collection of interacting objects
- Objects of a class share the same attributes, operations and relationships but may have different state
- Interactions between classes are modelled by relationships
- Relationships model how objects of different classes work together
  - Association, aggregation, composition, generalisation