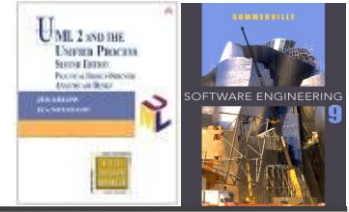


Lecture 3

ANALYSIS AND DESIGN

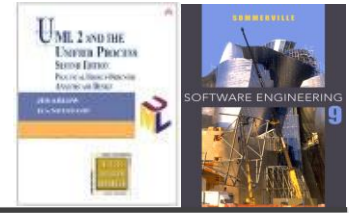
PB007 Software Engineering I
Faculty of Informatics, Masaryk University
Fall 2020

Outline



- ✧ Software analysis and design
- ✧ Structured vs. object-oriented methods

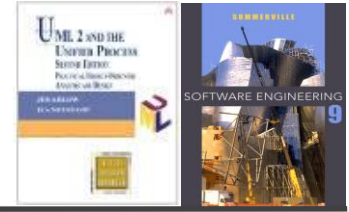
- ✧ UML Objects and classes, finding analysis classes
- ✧ Relationships between objects and classes
 - Links
 - Associations
 - Dependencies
- ✧ Inheritance and polymorphism



Software Analysis and Design

Lecture 3/Part 1

Analysis, design and implementation

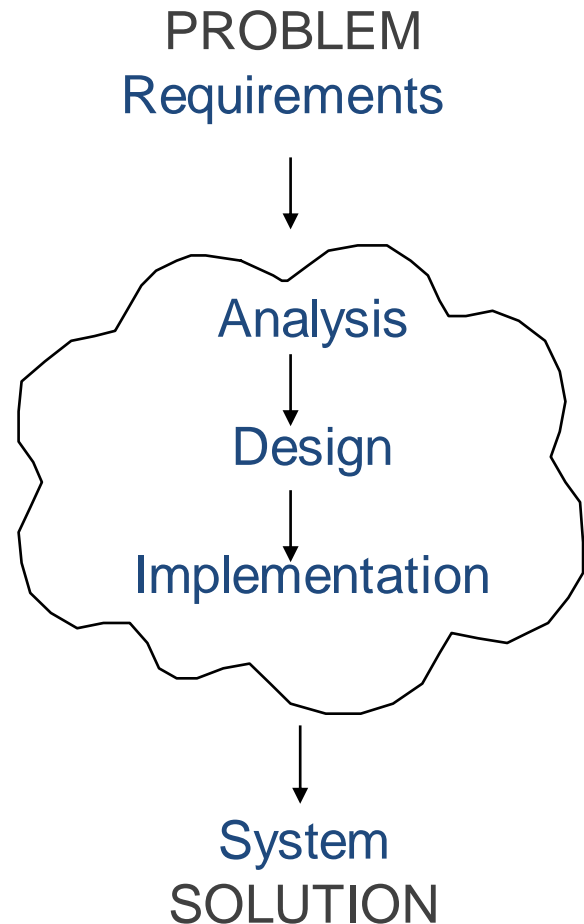


✧ Software development

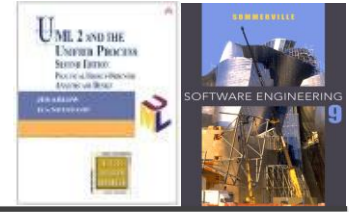
- **analysis, design and implementation**
- the stage in the software engineering process at which an **executable software system** is developed

“There are two ways of constructing a software design: One way is to make it so simple that there are **obviously no deficiencies**, and the other way is to make it so complicated that there are **no obvious deficiencies.**”

– C.A.R. Hoare

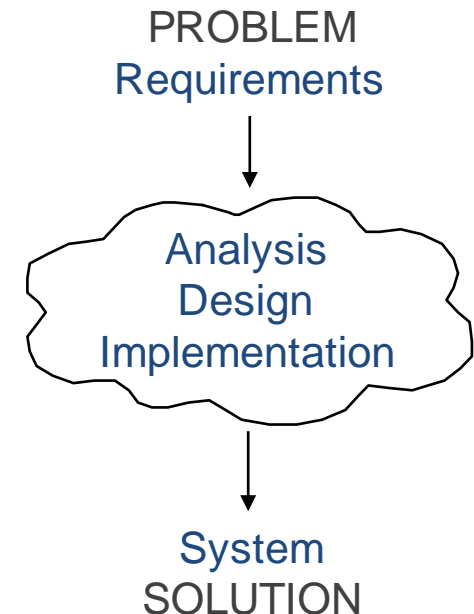


Analysis, design and implementation



✧ Software analysis, design and implementation are invariably inter-leaved with blurred border in between.

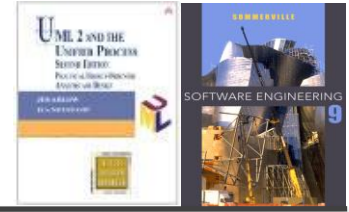
- **Software analysis** is a creative activity in which you identify software processes, entities (objects) and their relationships.
- **Software design** refines analytical models with implementation details.
- **Implementation** is the process of realizing the design as a program.



✧ Where is the line between the **problem** domain and the **solution** domain?

✧ Why do we **distinguish them** when the line is blurred anyway?

Process stages



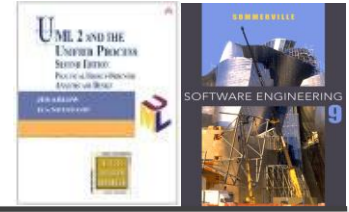
- ✧ There is a variety of different design processes that depend on the organization using the process.
- ✧ Common activities in these processes include:
 1. Define the context and modes of use of the system;
 2. Draft the system architecture;
 3. Identify the principal system processes and entities;
 4. Develop design models;
 5. Specify component/object interfaces;
 6. Finalize system architecture.
- ✧ What activities are part of analysis/design/implementation?

1. System context and interactions

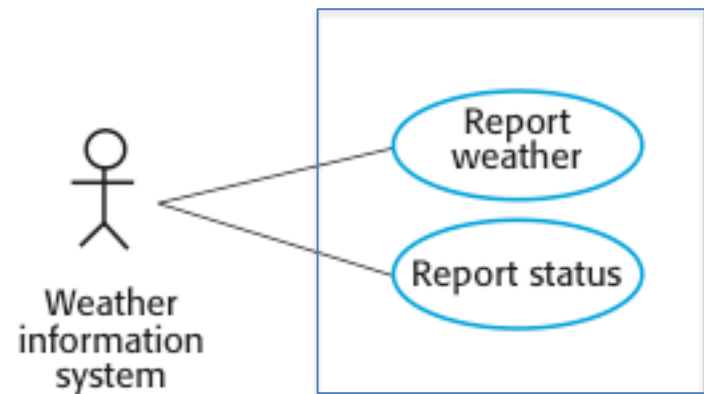
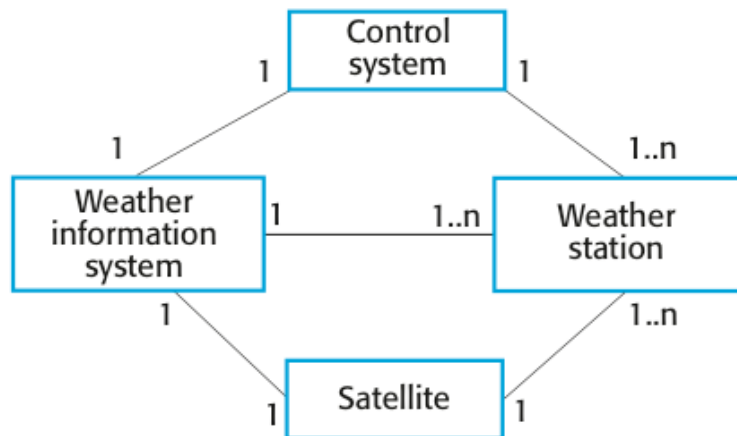


- ✧ Understanding the **relationships between the software and its external environment** is essential for deciding
 - how to provide the required system **functionality** and
 - how to **structure the system** to communicate with its environment.
- ✧ Understanding of the context also lets you establish the **boundaries** of the system.
 - Setting the system boundaries helps you decide what features are **implemented in the system** being designed and what features are **in other associated systems**.

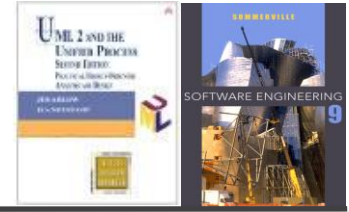
Context and interaction models



- ✧ A **system context** model is a structural model that demonstrates the users and other systems in the environment of the system being developed.
- ✧ An **interaction model** is a dynamic model that shows how the system interacts with its environment as it is used.
- ✧ Do we really need visual models for that? What is their role in A&D?



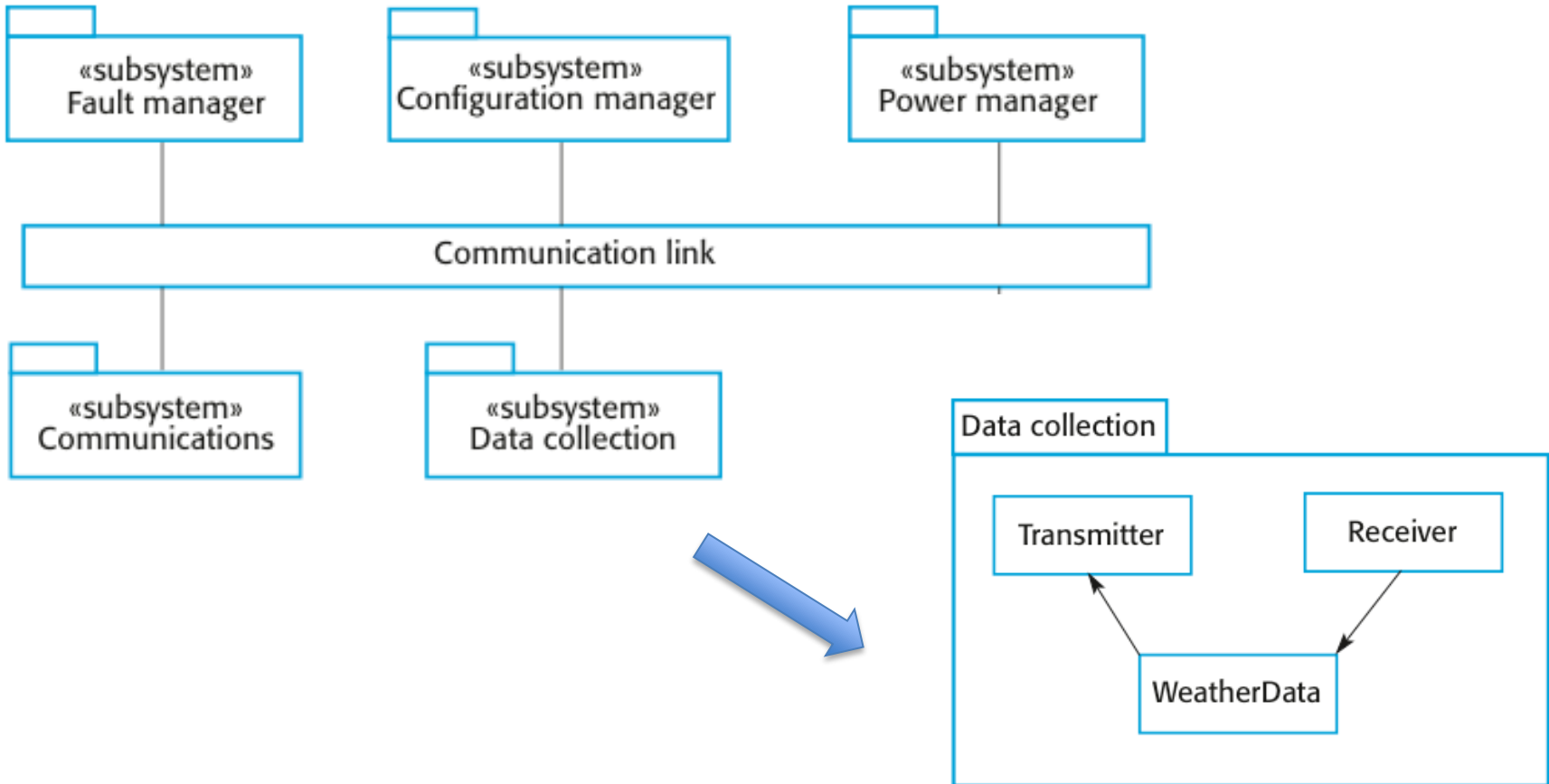
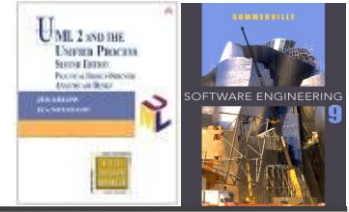
2. Architectural design



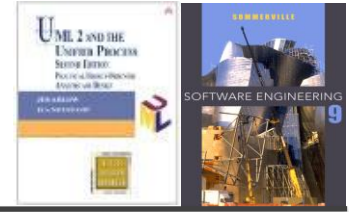
- ✧ Starts system **analysis** and/or finishes system **design**.
 - Is it the same architecture design in both cases?
- ✧ Involves **identifying major system components and their communications**.
 - Represents the link between requirements specification and analysis/design processes.
 - E.g. The weather station is composed of independent subsystems that communicate via (asynchronous) messaging.
- ✧ **Software architecture** gives answers to the most expensive questions.

– heard from O. Krajíček

High-level architecture of the weather station



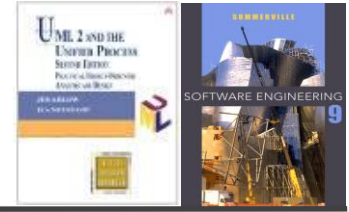
Architectural abstraction



- ✧ **Architecture in the small (analysis)** is concerned with the architecture of individual programs.
 - At this level, we are concerned with the way that an individual program is decomposed into components.

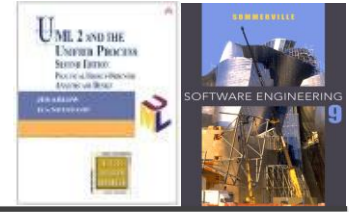
- ✧ **Architecture in the large (design)** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.
 - These systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture



- ✧ Stakeholder communication and project planning
 - Architecture may be used to facilitate the discussion by system stakeholders.
- ✧ System analysis
 - Means that analysis of whether the system can meet its non-functional requirements is possible.
- ✧ System documentation
 - Via a complete system model that shows the different components in a system, their interfaces and their connections.
- ✧ Large-scale reuse
 - The architecture may be reusable across a range of systems
 - Product-line architectures may be developed.

3. System analysis



- ✧ **Identification of system entities** (object classes in object-oriented analysis) playing the key roles in the system's problem domain, **and their relationships.**
- ✧ Distillation and documentation of key **system processes.**
- ✧ System analysis is a difficult **creative activity.**
 - There is no 'magic formula' for good analysis. It relies on the skill, experience and domain knowledge of system analysts.
- ✧ Object/relationships/processes identification is an **iterative process.** You are unlikely to get it right first time.

Weather station object classes



WeatherStation
identifier
reportWeather () reportStatus () powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

WeatherData
airTemperatures groundTemperatures windSpeeds windDirections pressures rainfall
collect () summarize ()

Ground thermometer
gt_Ident temperature
get () test ()

Anemometer
an_Ident windSpeed windDirection
get () test ()

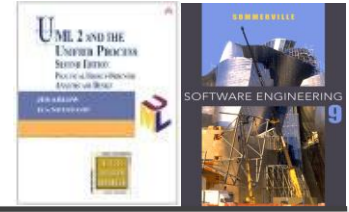
Barometer
bar_Ident pressure height
get () test ()

5. Design models



- ✧ Design models refine analysis models with the information required to **communicate and document the intended implementation** of the system.
 - E.g. Dependencies, interfaces, data-access classes, GUI classes.
- ✧ **Static models** describe the static structure of the system in terms of system entities and relationships.
 - Can you list some static UML diagrams?
- ✧ **Dynamic models** describe the dynamic interactions between entities.
 - Can you list some dynamic UML diagrams?

Key points



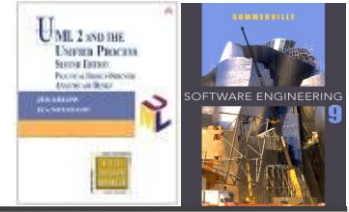
- ✧ The process of analysis and design includes activities to design the system **architecture**, identify **entities** in the system, describe the **design** using different models and document the **component interfaces**.
- ✧ **Software analysis** is a creative activity in which you identify software processes, entities (objects) and their relationships.
- ✧ **Software design** refines analytical models with implementation details.
- ✧ Software analysis and design are **inter-leaved activities**.



Structured vs. Object-Oriented Methods

Lecture 3/Part 2

Fundamental views of software systems



✧ Function oriented view

- System as a set of interacting functions. Functional transformations based in processes, interconnected with data and control flows.

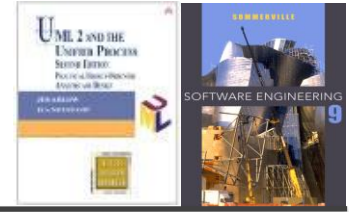
✧ Data oriented view

- Searches for fundamental data structures in the system. Functional aspect of the system (i.e. data transformation) is less significant.

✧ Object oriented view

- System as a set of interacting objects, encapsulating both the data and operations performed on the data.

Structured vs. object-oriented analysis



✧ Structured analysis

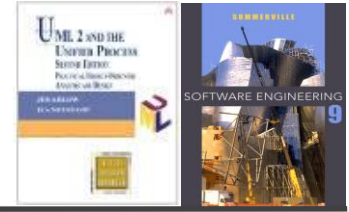
- Driven by the **function oriented view**, in synergy with **data oriented view**, through the concept of functional decomposition.

✧ Object-oriented analysis

- Driven by the **object oriented view**.

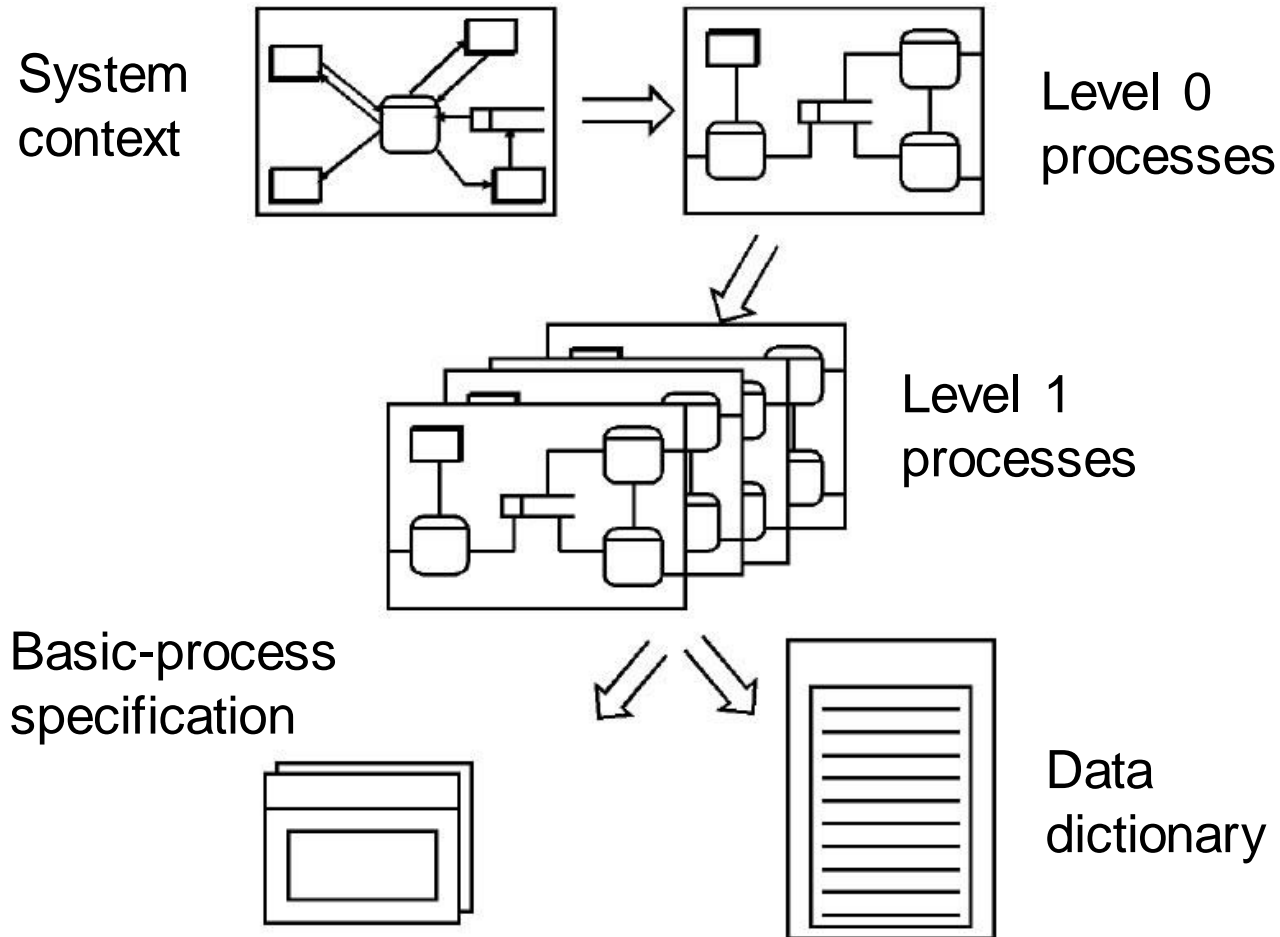
Do they have anything in common?

Structured analysis and design

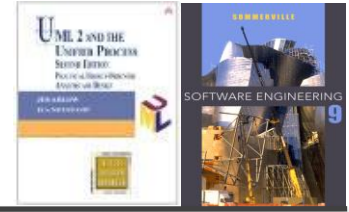


- ✧ Divides a project on small, well defined activities and defines the order and interaction of the activities.
- ✧ Using **hierarchical graphical techniques**, resulting in a detailed structured specification, which can be understood by both system engineers and users.
- ✧ Effective in project **structuring to smaller parts**, which simplifies time and effort estimates, deliverables control and project management as such.
- ✧ Aimed at increasing system quality.

Functional decomposition



Core notations of structured methods



✧ Context diagram

- Models system boundary and environment.

✧ Data flow diagram (DFD)

- Models the system as a network of processes completing designated functions and accessing system data.

✧ Entity relationship diagram (ERD)

- Models system's data.

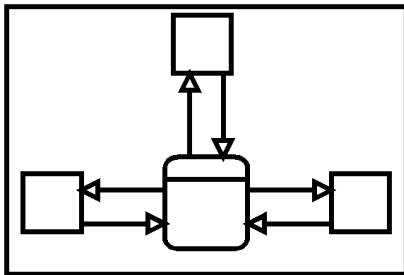
✧ State diagram (STD)

- Models system states and actions guarding transitions from one state to another.

E. Yourdon: Modern structured analysis

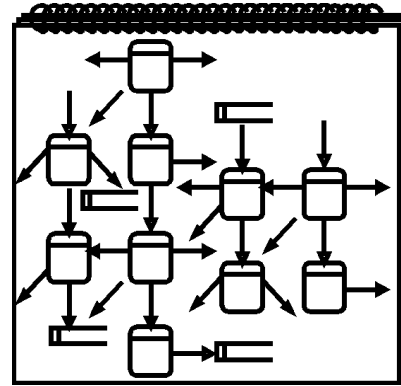


Environment model

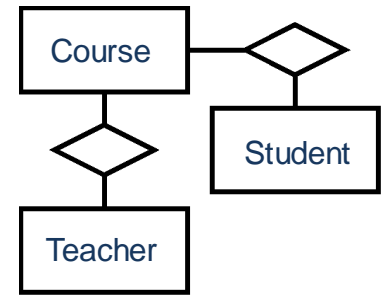


- | Events: |
|----------------|
| E1: registered |
| E2: rolled in |
| E3: rolled out |
| E4: started |
| E5: ended |

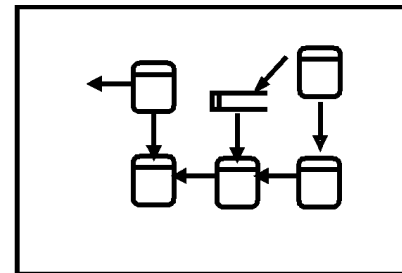
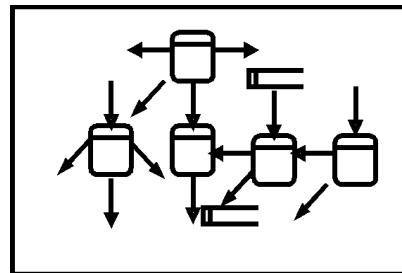
Behavioral model



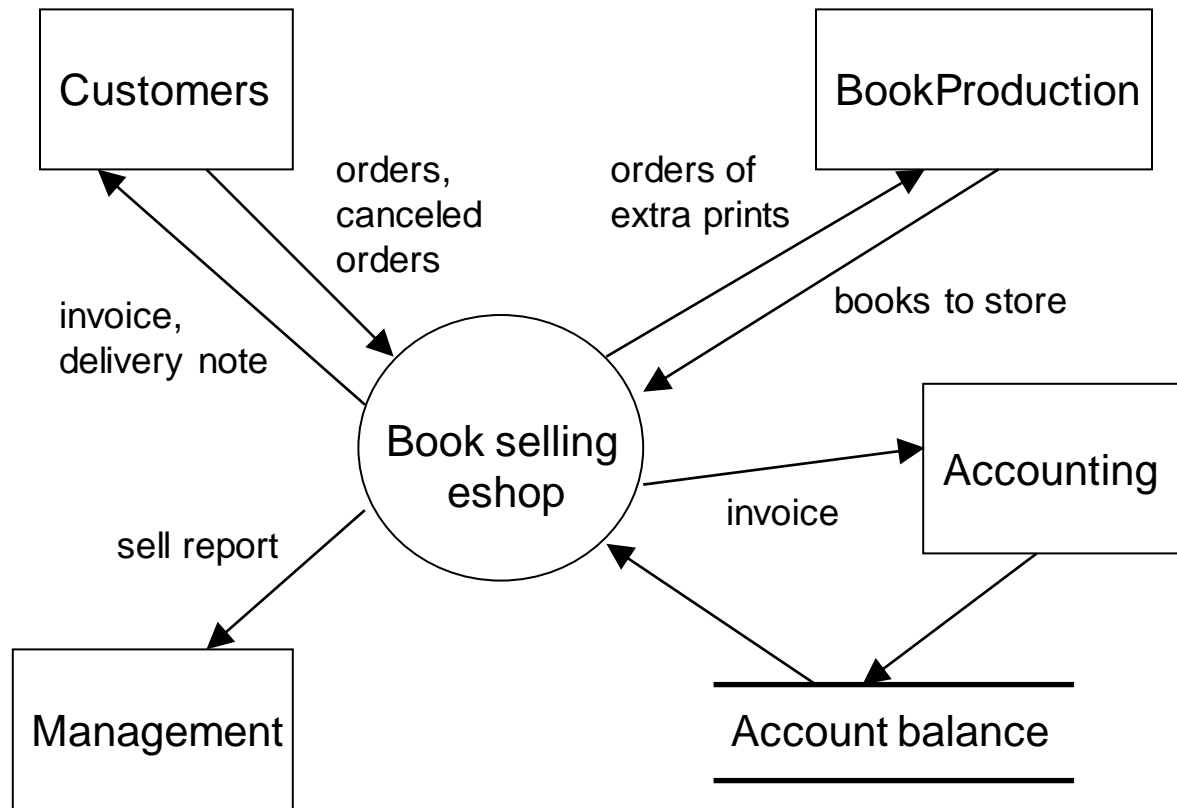
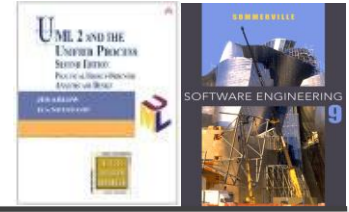
Data model



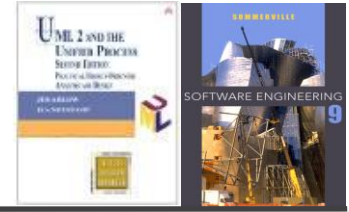
Functional decomposition



Context diagram example

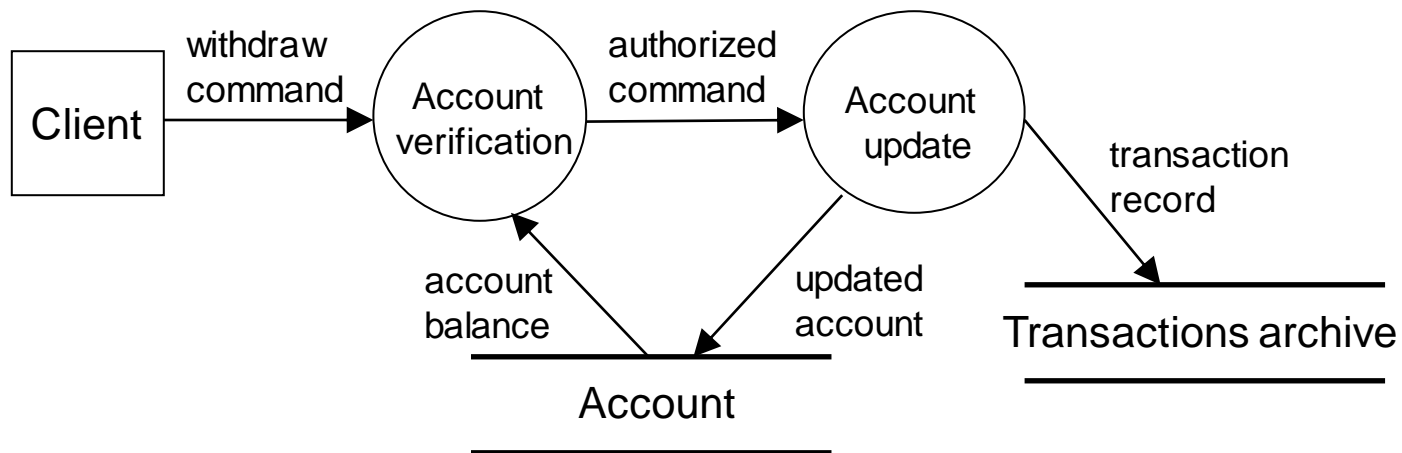


Data flow diagram (DFD) example

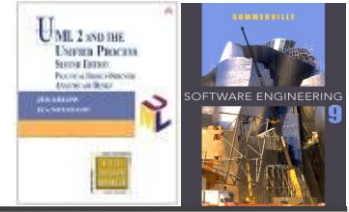


✧ DFD consists of four types of elements:

- Processes
- Data flows
- Data stores
- Terminators

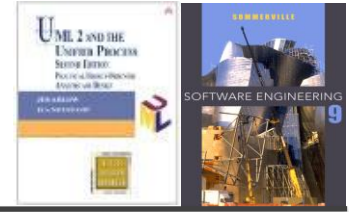


Object-oriented analysis and design



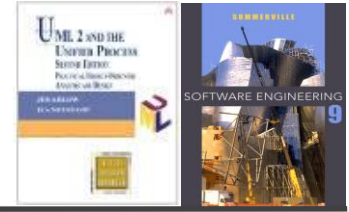
- ✧ Software engineering approach that models a system as a group of **interacting objects**.
- ✧ Each **object** represents some entity of interest in the system being modeled, and is characterized by its **class**, its **state** (data elements), and its **behavior**.
- ✧ **Various models** can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects.
- ✧ There is a number of **different methods**, defining the ordering of modeling activities. The **modeling notation** uses to be unified (UML).

UML notation for object-oriented methods



- ✧ External perspective
 - **Use case diagram**
- ✧ Structural perspective
 - **Class diagram**, Object diagram, Component diagram, Package diagram, Deployment diagram, Composite structure diagram
- ✧ Interaction perspective
 - **Sequence diagram**, Communication diagram, Interaction overview diagram, Timing diagram
- ✧ Behavioral perspective
 - **Activity diagram**, State diagram

Exemplary method (Unified Process, analysis and design excerpt)



1. Requirements

- System boundary, actors and requirements modelling with **Use Case diagram**.

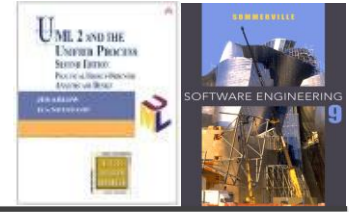
2. Analysis

- Identification of analysis classes, relationships, inheritance and polymorphism, and their documentation with a **Class diagram**.
- Use Case realization with **Interaction** and **Activity diagrams**.

3. Design

- Design classes, interfaces and components, resulting in refined **Class diagrams** and **Component diagrams**.
- Detailed Use Case realization with **Interaction** and **State diagrams**.

Key points



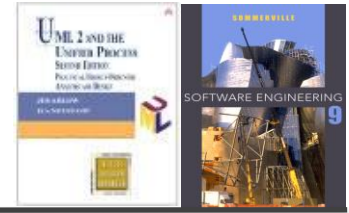
✧ Structured methods

- System as a set of nested processes accessing system data.

✧ Object-oriented methods

- System as a set of interacting objects (functions and data).

	Structured analysis	Object-oriented analysis
System boundary	Context diagram	Use case diagram
Functionality	Data flow diagram	Activity diagram Interaction diagrams
Data	Entity-relationship diagram	Class and Object diagram
Control	State diagram	State diagram



Object-Oriented Analysis in UML

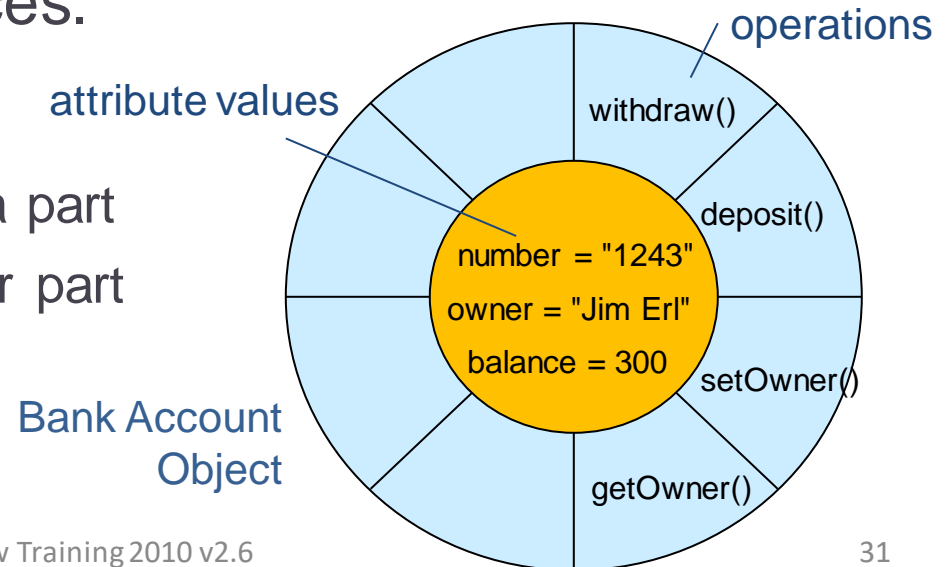
Lecture 3/Part 3

Analysis objects and classes

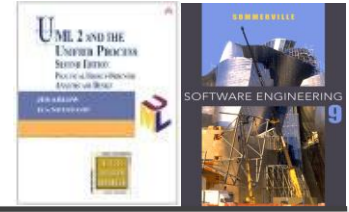


What are objects?

- ✧ Objects consist of data and function packaged together in a reusable unit. Objects **encapsulate** data.
- ✧ Every object is an instance of some **class** which defines the common set of **features** (attributes and operations) shared by all of its instances.
- ✧ Objects have:
 - **Attribute values** – the data part
 - **Operations** – the behaviour part

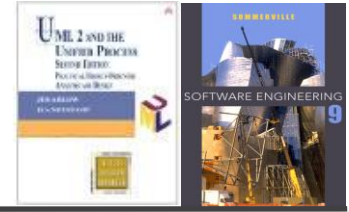


All objects have

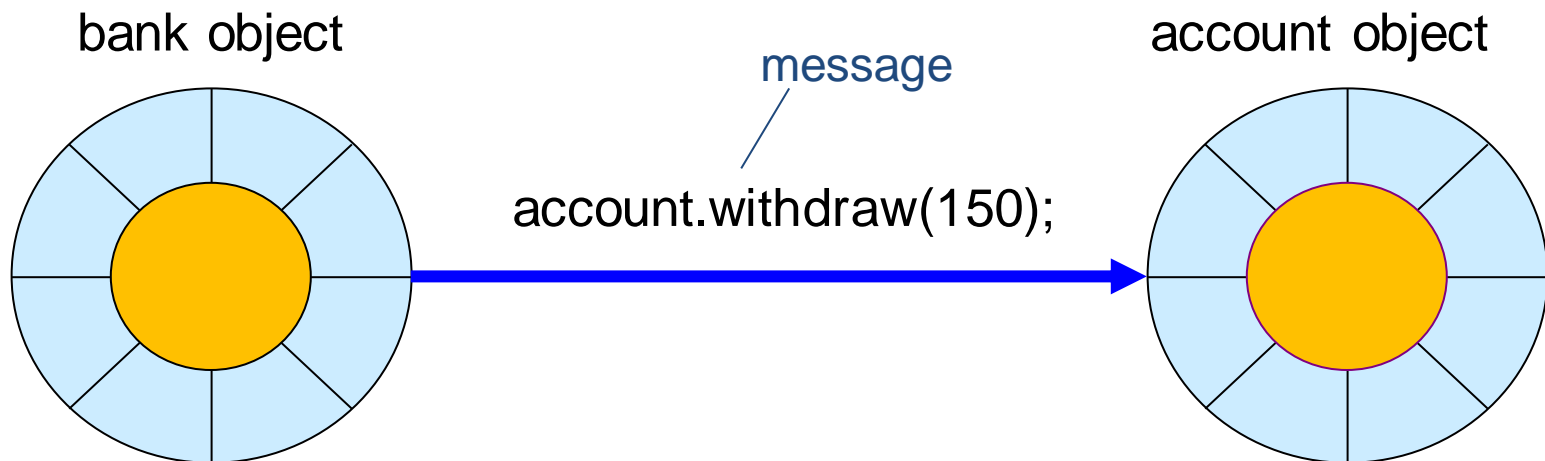


- ✧ **Identity:** Each object has its own unique identity and can be accessed by a unique handle
 - Distinguish two cars of the same type and one car referenced from two places.
- ✧ **State:** This is the actual data values stored in an object at any point in time
 - On and off for a light bulb (one attribute).
 - On + busy, on + idle, off for a printer (two attributes).
- ✧ **Behaviour:** The set of operations that an object can perform

Messaging



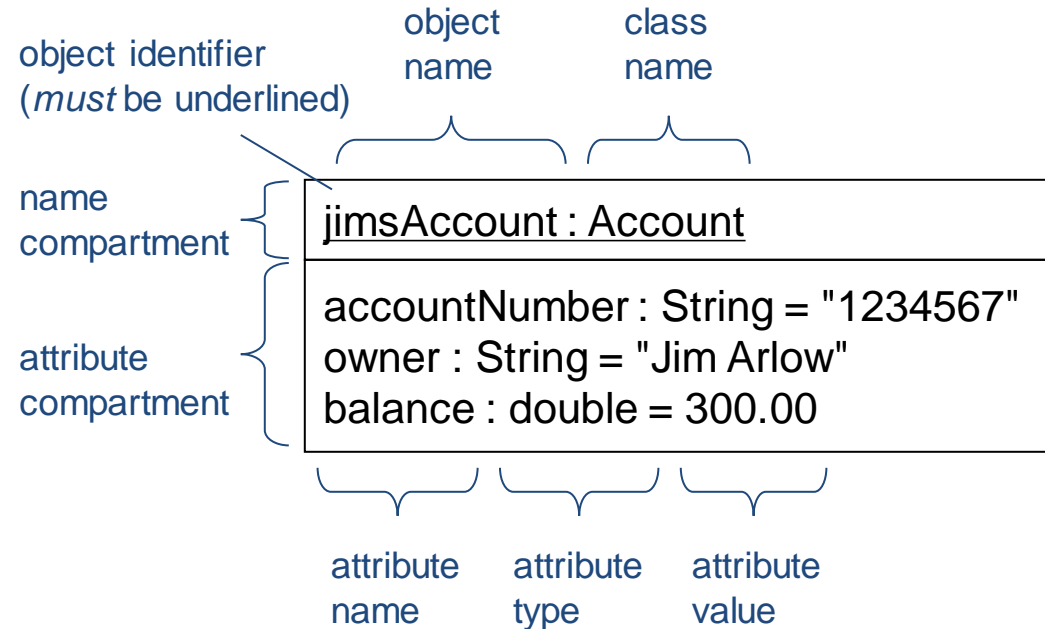
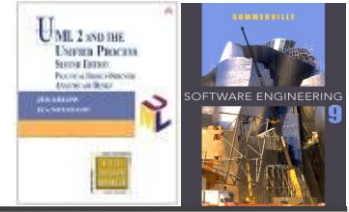
- ✧ In OO systems, objects send messages to each other over links
- ✧ These messages cause an object to invoke an operation



the bank object sends the message “withdraw 150” to an account object.

the account object responds by invoking its withdraw() operation. This operation decrements the account balance by 150.

UML Object Syntax



variants

(N.B. we've omitted the attribute compartment)

object and class name

jimsAccount : Account

object name only

jimsAccount

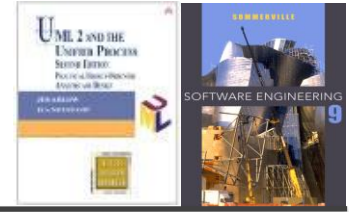
class name only

: Account

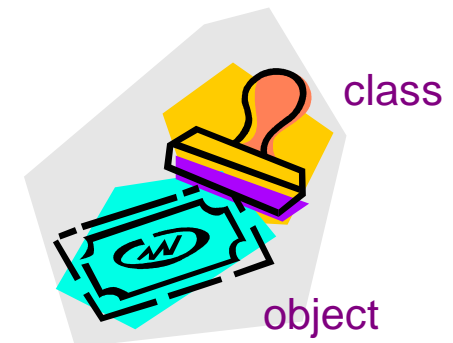
an anonymous object

- ✧ All objects of a particular class have the same set of operations. They are not shown on the object diagram, they are shown on the class diagram (see later)
- ✧ Attribute types are often omitted to simplify the diagram
- ✧ Naming: object and attribute names in lowerCamelCase, class names in UpperCamelCase

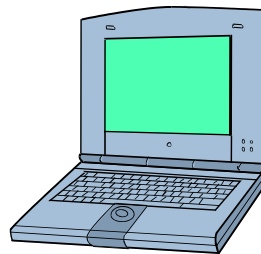
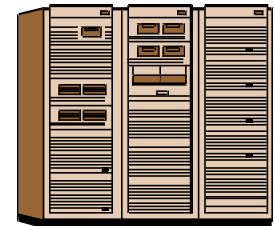
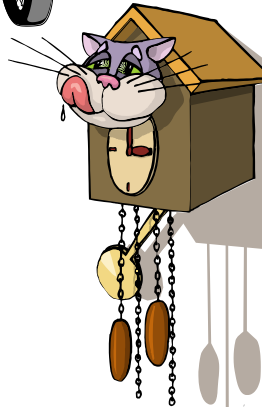
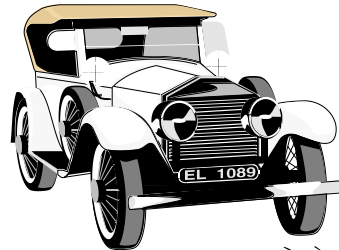
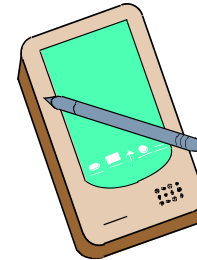
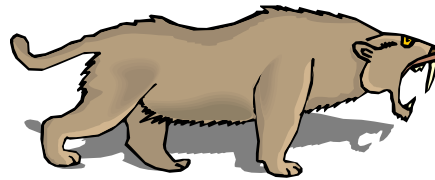
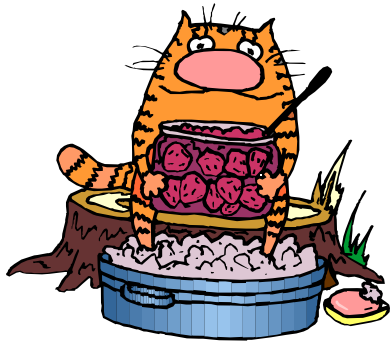
What are classes?



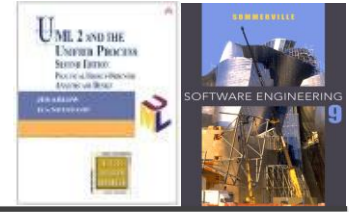
- ✧ Every object is an instance of one class - the class describes the "type" of the object
- ✧ Classes allow us to model sets of objects that have the same set of features - **a class acts as a template for objects:**
 - The class determines the structure (set of features) of all objects of that class
 - All objects of a class **must** have the same set of operations, **must** have the same attributes, but **may** have different attribute values
- ✧ **Classification** is one of the most important ways we have of organising our view of the world
- ✧ Think of classes as being like:
 - Rubber stamps
 - Cookie cutters



Exercise - how many classes?



Classes and objects



- ✧ Objects are instances of classes.
- ✧ **Instantiation** is the creation of new instances of model elements.
- ✧ Most classes provide special operations called **constructors** to create instances of that class.
- ✧ These operations have **class-scope** i.e. they belong to the class itself rather than to objects of the class.

class

Account
accountNumber : String owner : String balance : double
withdraw() deposit()

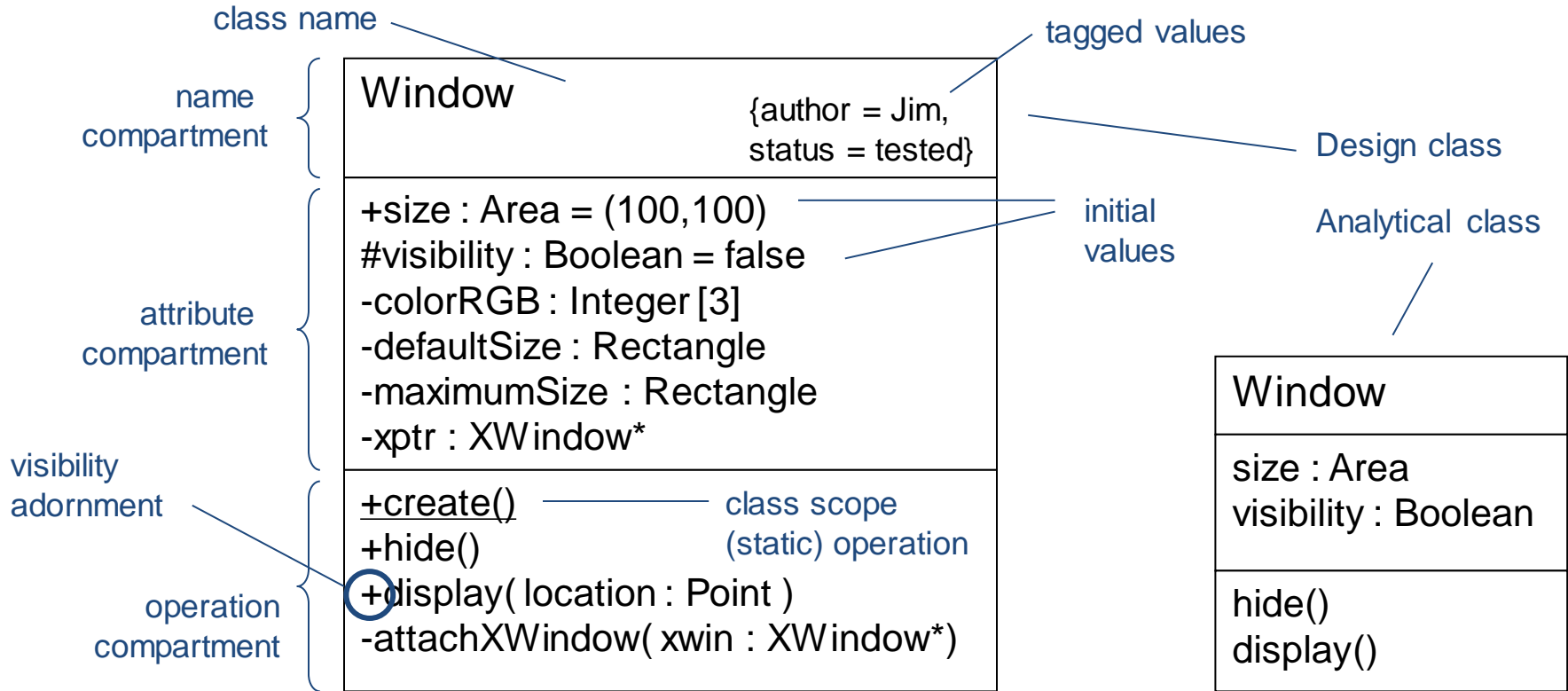
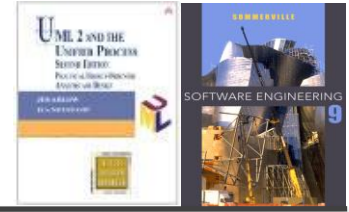
<u>JimsAccount:Account</u>
accountNumber : "801" owner : "Jim" balance : 300.00

<u>fabsAccount:Account</u>
accountNumber : "802" owner : "Fab" balance : 1000.00

<u>ilasAccount:Account</u>
accountNumber : "803" owner : "Ila" balance : 310.00

objects

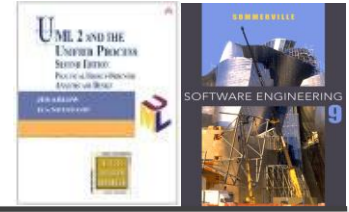
UML class notation



✧ Classes are named in UpperCamelCase – **avoid abbreviations!**

✧ Use descriptive names that are nouns or noun phrases

Operation compartment



Operation signature

visibility name (direction parameterName : parameterType = default, ...) : returnType
parameter list / or a list r1, r2, ... rn

Direction

in input value, default
out repository for system output
inout modifiable input value
return operation return value(s)

Scope

instance scope defaults
class scope underlined

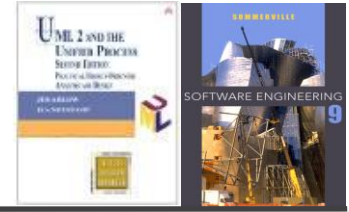
Constructors

generic constructor name or
Java/C++ standard
+BankAccount(aNumber : int)

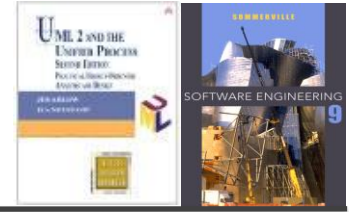
operation
compartment

BankAccount
-accountNumber : int <u>-count : int = 0</u>
<u>+create(aNumber : int)</u> <u>+getNumber() : int</u> <u>-incrementCount()</u> <u>+getCount() : int</u>

Key points



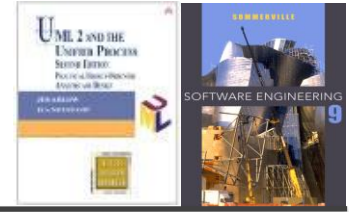
- ✧ We have looked at objects and classes and examined the relationship between them
- ✧ We have explored the UML syntax for modelling classes including:
 - Attributes
 - Operations
- ✧ We have seen that scope controls access
 - Class scope attributes are shared by all objects of the class and are useful as counters
 - Attributes and operations are normally instance scope
 - We can use class scope operations for constructor and destructors



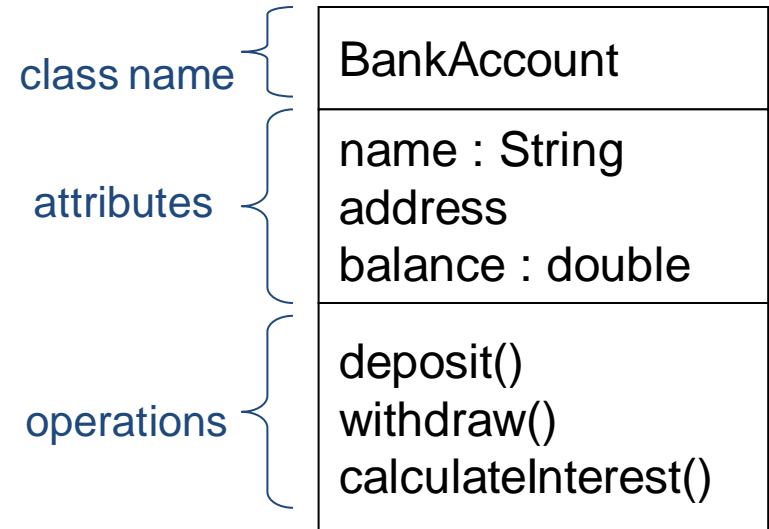
Finding Analysis Classes

Lecture 3/Part 4

What are Analysis classes?

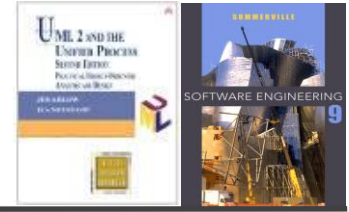


- ✧ Analysis classes represent a crisp abstraction in the problem domain
 - They may ultimately be refined into one or more design classes
- ✧ Analysis classes have:
 - A very “high level” set of attributes. They indicate the attributes that the design classes might have.
 - Operations that specify at a high level the key services that the class must offer. In Design, they will become actual, implementable, operations.



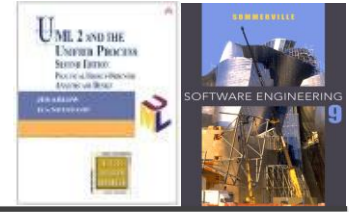
Specify attribute types if you know what they are.

What makes a good analysis class?



- ✧ Its name reflects its intent
- ✧ It is a **crisp abstraction** that models one specific element of the problem domain
 - It maps onto a clearly identifiable feature of the problem domain
- ✧ It has **high cohesion**
 - Cohesion is the degree to which a class models a single abstraction
 - Cohesion is the degree to which the responsibilities of the class are semantically related
- ✧ It has **low coupling**
 - Coupling is the degree to which one class depends on others

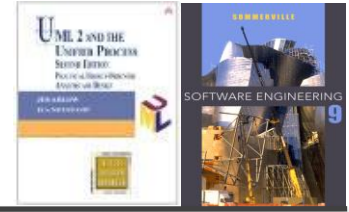
Rules of thumb



- ✧ 3 to 5 operations per class
- ✧ Each class collaborates with others
- ✧ Beware many very small classes
- ✧ Beware few but very large classes
- ✧ Beware of “functoids”
- ✧ Beware of “omnipotent” classes
- ✧ Avoid deep inheritance trees

A *responsibility* is a contract or obligation of a class - it resolves into operations and attributes

Finding classes



✧ Perform noun/verb analysis on documents:

- Nouns are candidate **classes**
- Verbs are candidate **responsibilities**

What documents can be studied?

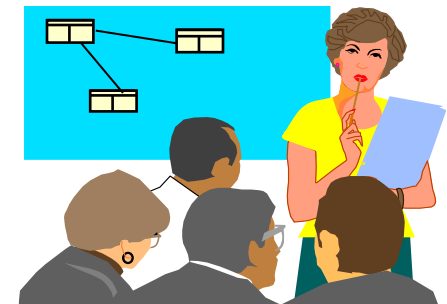
✧ Perform CRC card analysis

- Class, Responsibilities and Collaborators
- A two phase brainstorming technique using sticky notes – first brainstorm and then analyse the data

Class Name: BankAccount	
Responsibilities:	Collaborators:
Maintain balance	Bank

things the class does

things the class works with



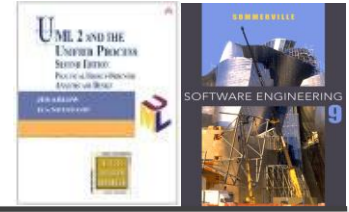
Other sources of classes



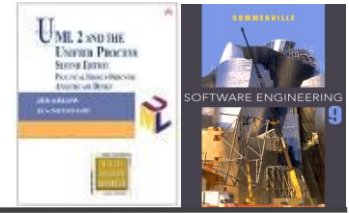
- ✧ Physical objects
- ✧ Paperwork, forms
 - Be careful when relying on processes that need to change
- ✧ Known interfaces to the outside world
- ✧ Conceptual entities that form a cohesive abstraction

- ✧ **With all techniques, beware of spurious classes**
 - Look for **synonyms** - different words that mean the same
 - Look for **homonyms** - the same word meaning different things

Key points



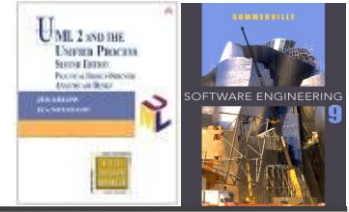
- ✧ We've looked at what constitutes a well-formed analysis class
- ✧ We have looked at two analysis techniques for finding analysis classes:
 - Noun verb analysis of use cases, requirements, glossary and other relevant documentation
 - CRC analysis



Relationships Between Objects and Classes

Lecture 3/Part 5

What is a link?



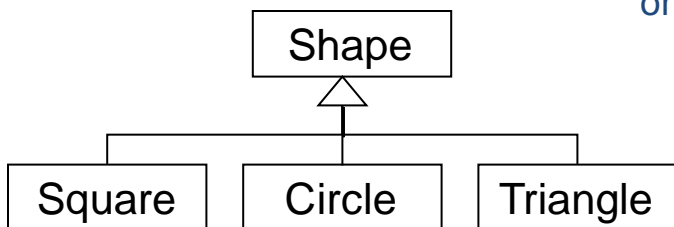
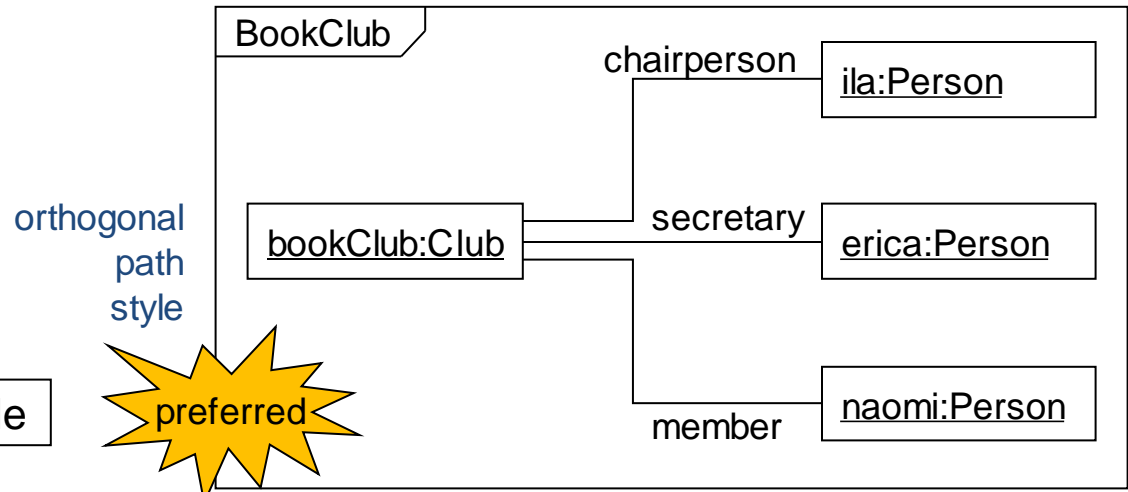
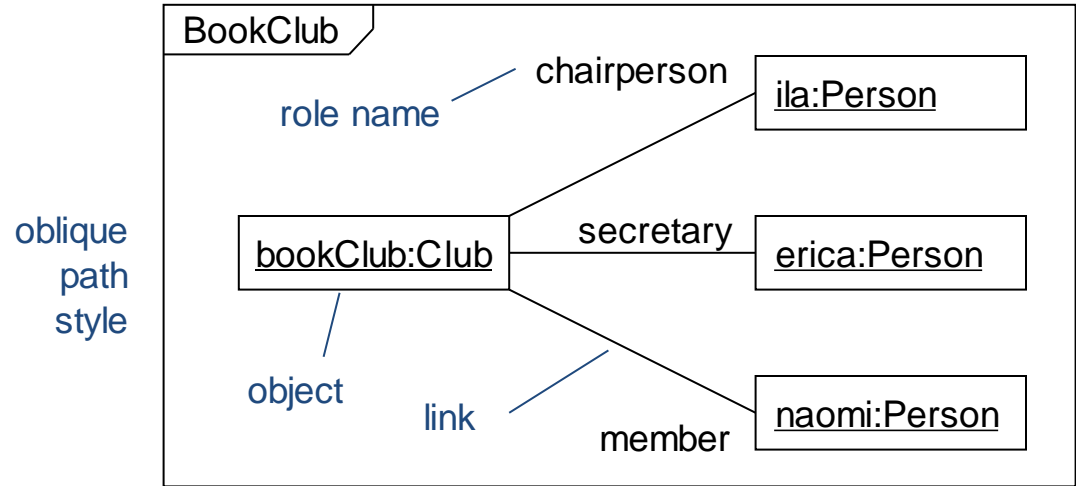
- ✧ Links are connections between objects
 - Think of a link as a **telephone line** connecting you and a friend. You can send messages back and forth using this link
- ✧ Links are the way that objects communicate
 - Objects **send messages to each other** via links
 - Messages **invoke operations**
- ✧ OO programming languages implement links as object references or pointers
 - When an object has a stored reference to another object, we say that there is a **link** between the objects

Object diagrams

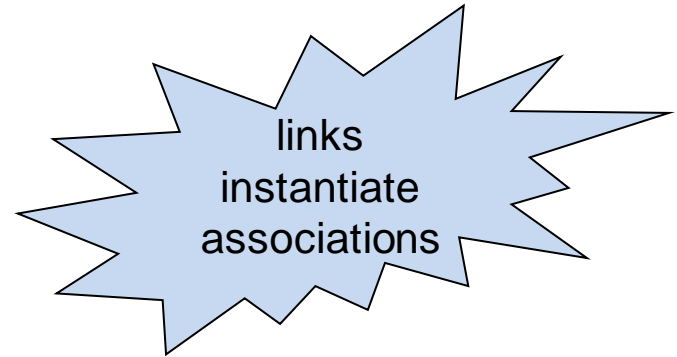
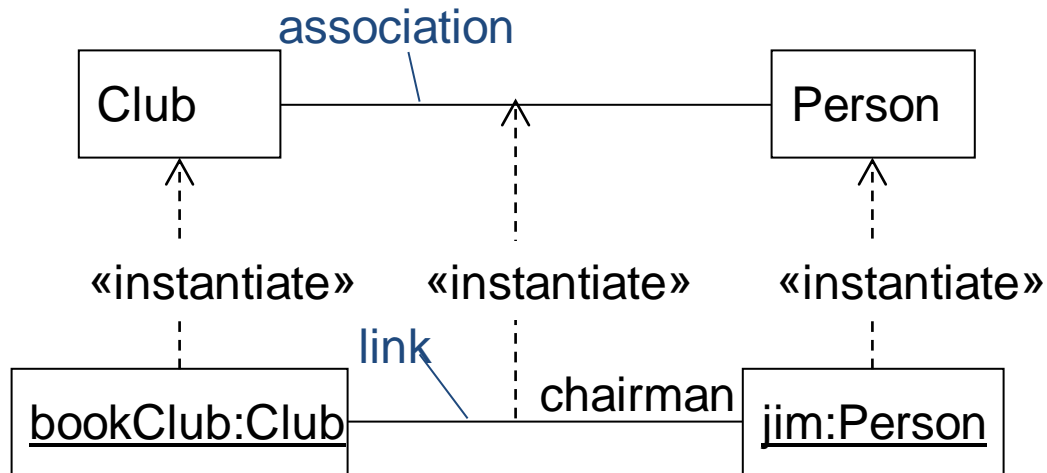
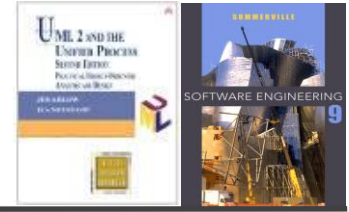


❖ Paths in UML diagrams can be drawn as orthogonal, oblique or curved lines

❖ We can combine paths into a tree if each path has **the same properties**

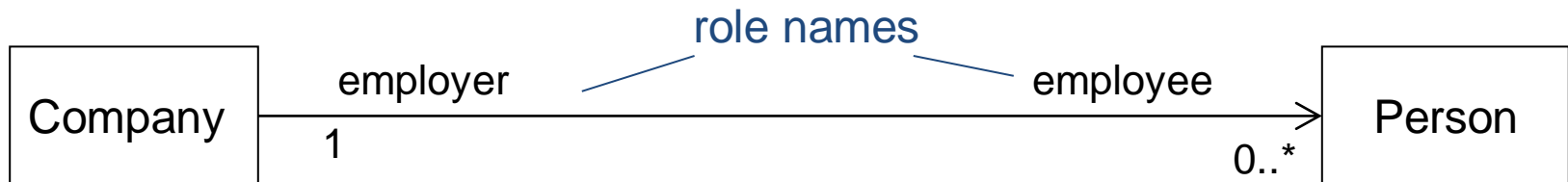


What is an association?



- ✧ Associations are relationships between classes
- ✧ Associations between classes indicate that there **may be** links between objects of those classes, while links indicates that there **must be** associations
- ✧ Can there be a communication between objects of two classes that have no association between them?

Association syntax



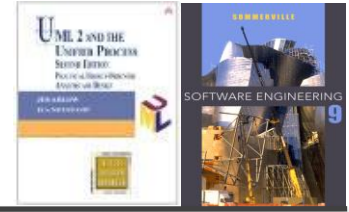
✧ An association can have **role names** OR an **association name**

✧ **Multiplicity** is a constraint that specifies the number of objects that can participate in a relationship at any point in time

- If multiplicity is not explicitly stated in the model then it is undecided – *there is no default multiplicity*

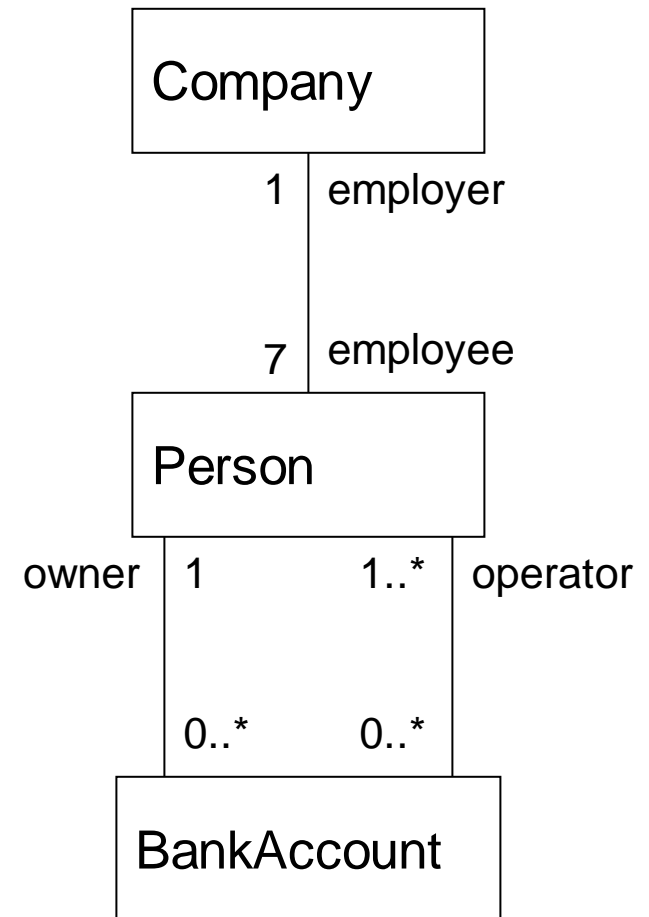
multiplicity: min..max	
0..1	zero or 1
1	exactly 1
0..*	zero or more
1..*	1 or more
1..6	1 to 6

Multiplicity exercise

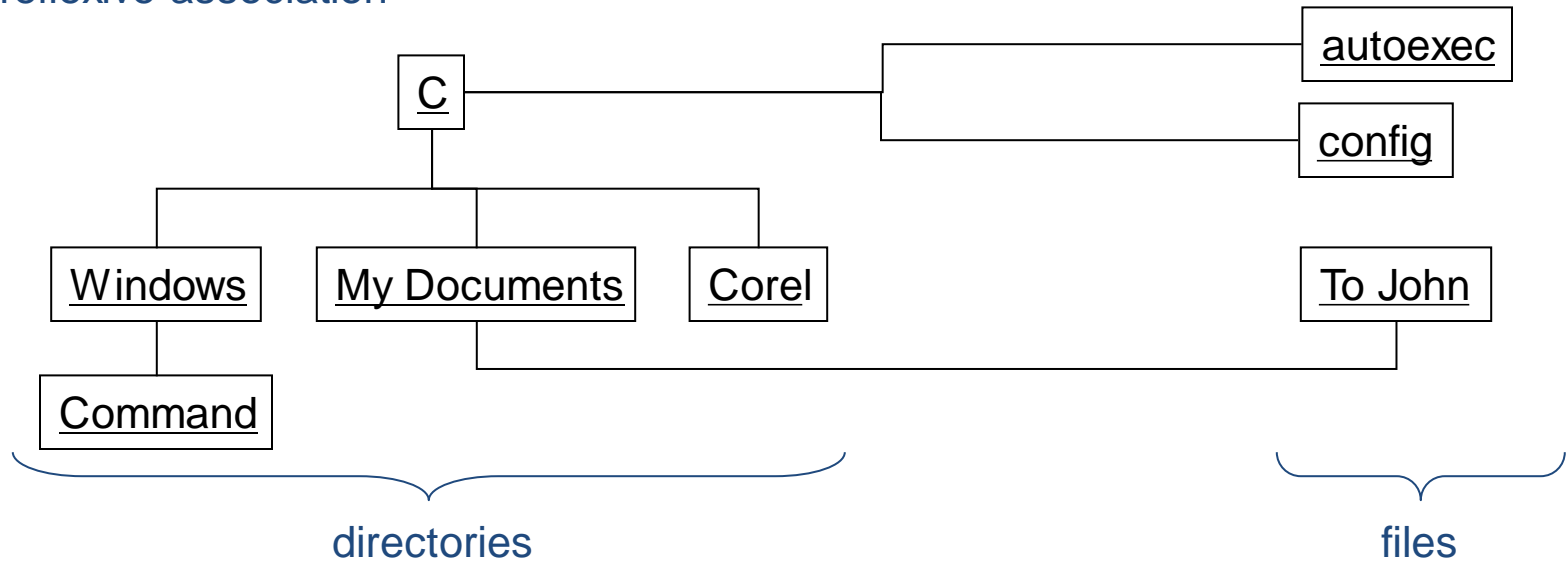
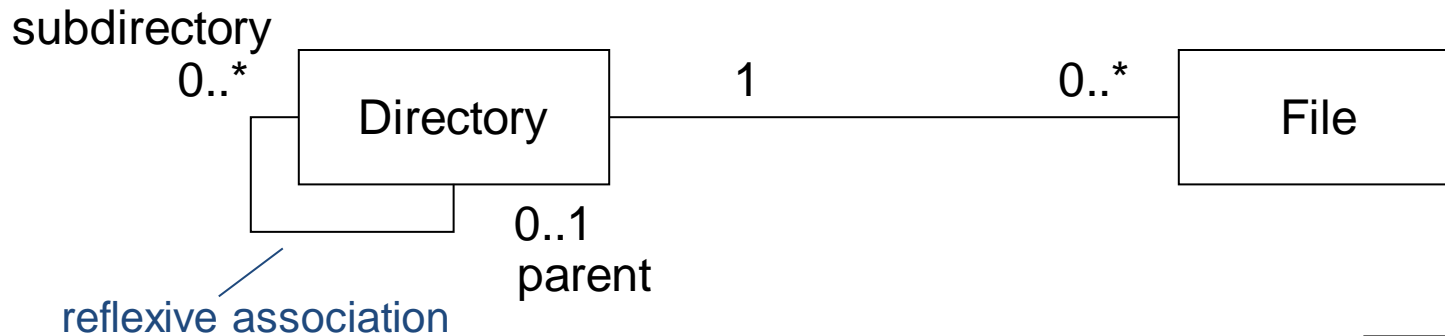


✧ How many

- Employees can a Company have?
- Employers can a Person have?
- Owners can a BankAccount have?
- Operators can a BankAccount have?
- BankAccounts can a Person have?
- BankAccounts can a Person operate?



Reflexive associations: file system example



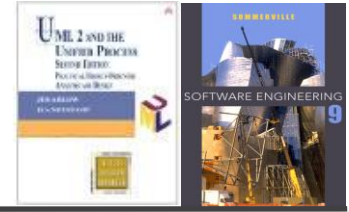
✧ If ToJohn was a directory, would it still conform to the class diagram?

Hierarchies and networks



hierarchy	network
<p>In an association hierarchy, each object has zero or one object directly above it.</p>	<p>In an association network, each object has zero or many objects directly above it.</p>

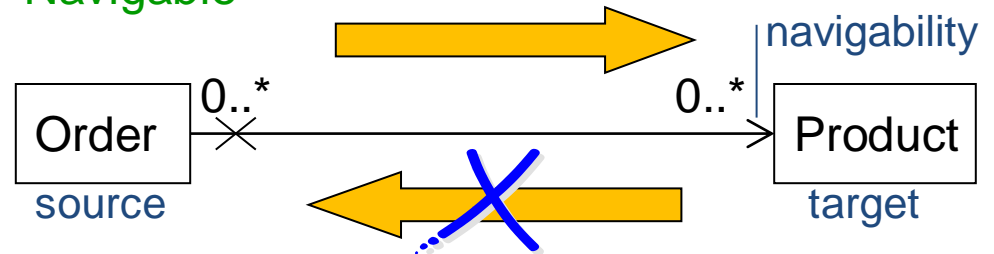
Navigability



- ✧ Navigability indicates that it is possible to traverse from an object of the **source** class to objects of the **target** class

An Order object stores a list of Products

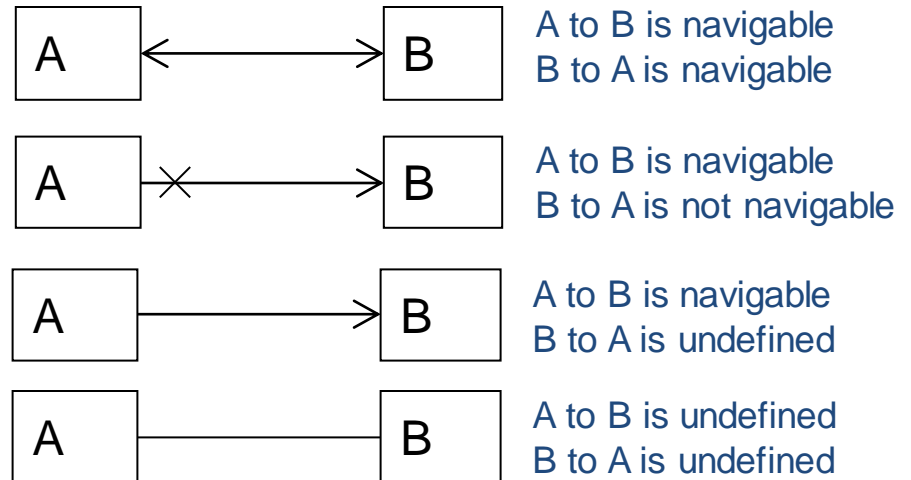
Navigable



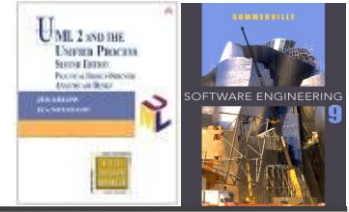
Not navigable

A Product object does not store a list of Orders

- ✧ Can there be a communication in a direction not supported by the navigability?
- ✧ Are some of the cases on the right equivalent?

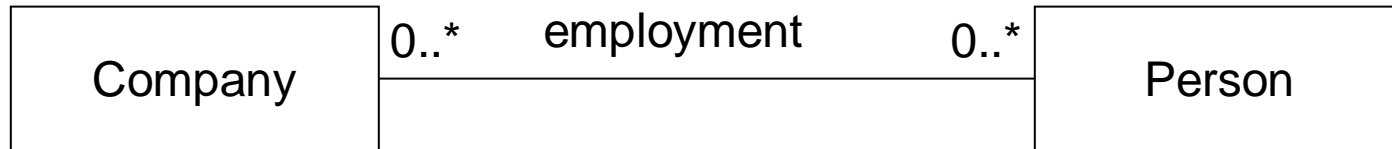
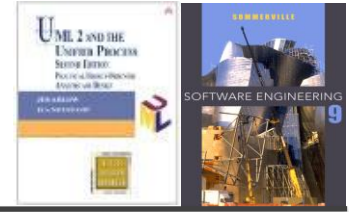


Associations and attributes

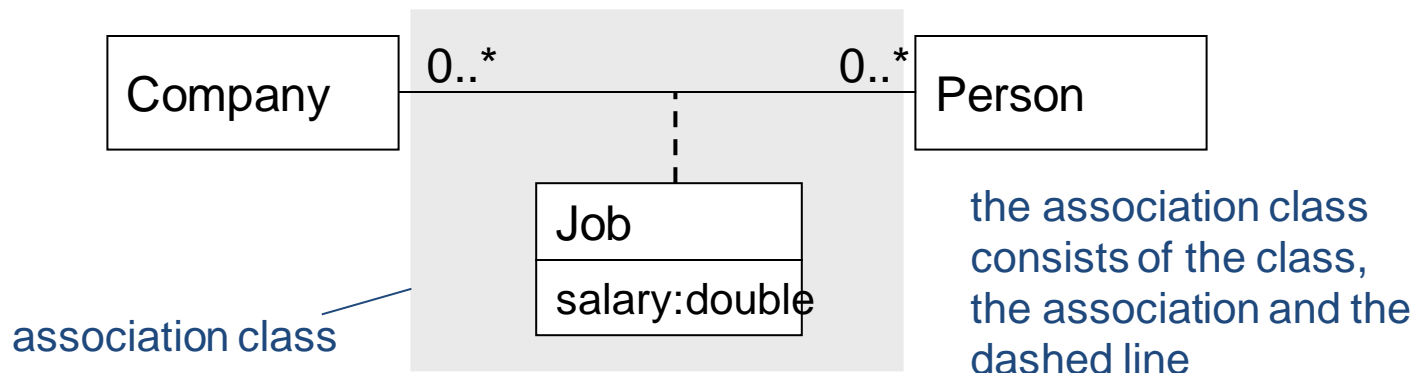


- ✧ An association is (through its role name) a **representation of an attribute**
- ✧ **Use associations when:**
 - The target class is an important part of the model
 - The target class is a class that you have designed yourself
- ✧ **Use attributes when:**
 - The target class is not important, e.g. a primitive type such as number, string
 - The target class is just an implementation detail such as a bought-in component or a library component e.g. `Java.util.Vector` (from the Java standard libraries)

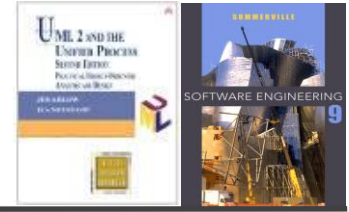
Association classes



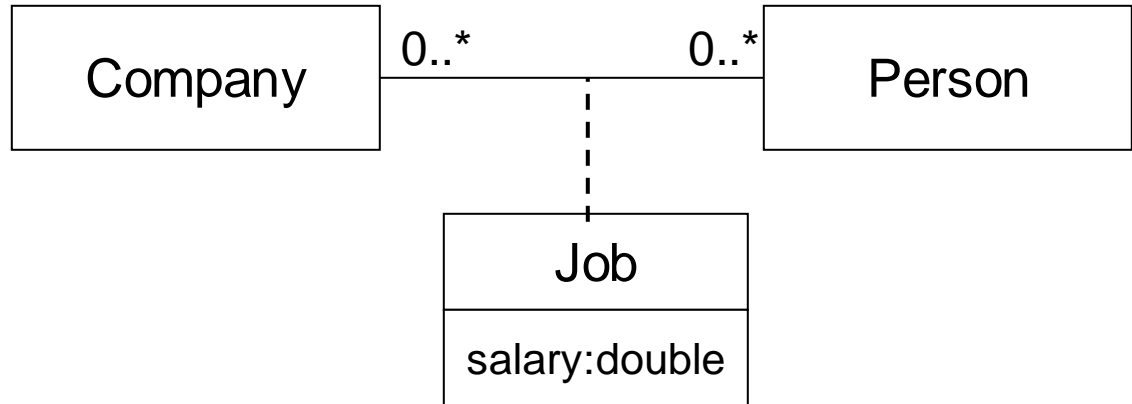
- ✧ Where do we record the Person's salary?
- ✧ We model the association itself as an association class. Exactly one instance of this class exists for each link between a Person and a Company.
- ✧ We can place the salary and any other attributes or operations which are really features of the association into this class



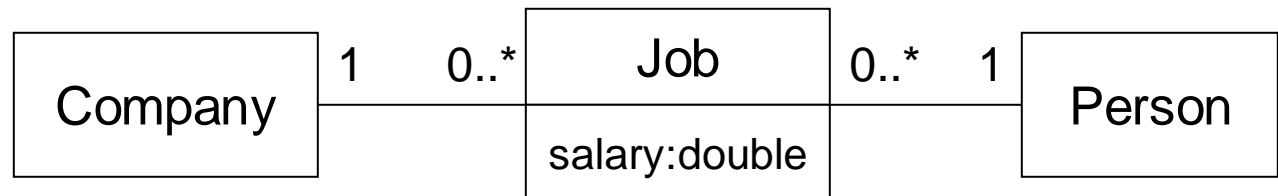
Using association classes



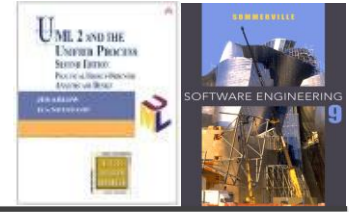
If we use an association class, then a particular Person can have only **one** Job with a particular Company



If, however a particular Person can have **multiple** jobs with the same Company, then we must use a reified association

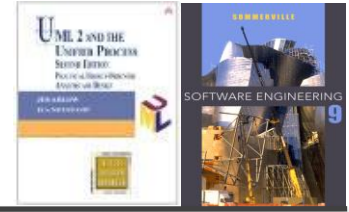


Dependencies



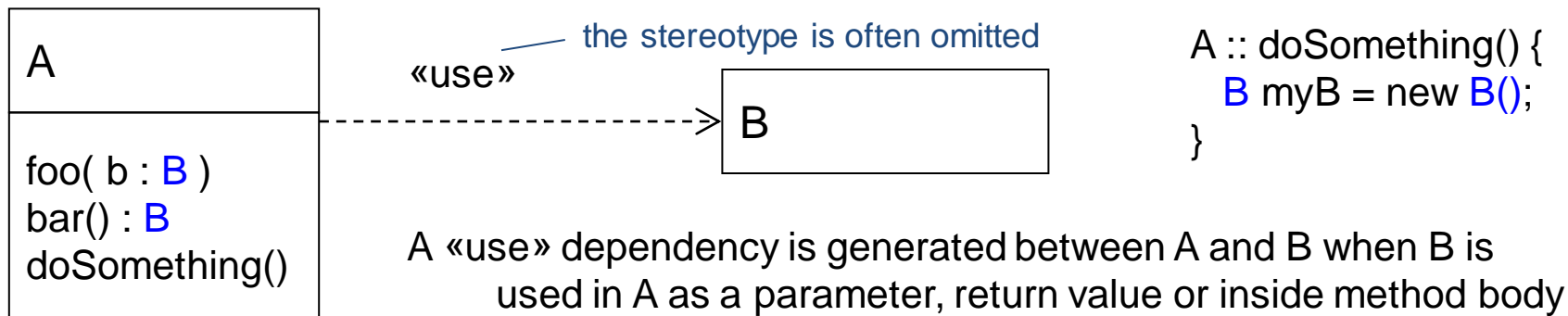
- ✧ "A dependency is a relationship between two elements where a change to one element (the supplier) may affect or supply information needed by the other element (the client)".
 - In other words, the client **depends** in some way on the supplier
 - Weaker type of relationship than **association**
 - Can there be both association and dependency between two classes?
- ✧ Three types of dependency:
 - **Usage** - the client uses some of the services made available by the supplier to implement its own behavior – this is the most commonly used type of dependency
 - **Abstraction** - a shift in the level of abstraction. The supplier is more abstract than the client
 - **Permission** - the supplier grants some sort of permission for the client to access its contents – this is a way for the supplier to control and limit access to its contents

Usage dependencies



❖ Stereotypes

- «**use**» - the client makes use of the supplier to implement its behaviour
- «**call**» - the client operation invokes the supplier operation
- «**parameter**» - the supplier is a parameter of the client operation
- «**send**» - the client (an operation) sends the supplier (a signal) to some unspecified target
- «**instantiate**» - the client is an instance of the supplier



Abstraction and permission dependencies



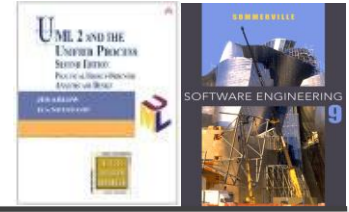
✧ Abstraction dependencies

- «**trace**» - the client and the supplier represent the same concept but at different points in development
- «**substitute**» - the client may be substituted for the supplier at runtime. The client and supplier must realize a common contract. Use in environments that *don't* support specialization/generalization
- «**refine**» - the client represents a fuller specification of the supplier
- «**derive**» - the client may be derived from the supplier. The client is logically redundant, but may appear for implementation reasons

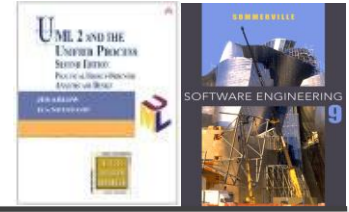
✧ Permission dependencies

- «**access**» the public contents of the supplier package are added as private elements to the namespace of the client package
- «**import**» the public contents of the supplier package are added as public elements to the namespace of the client package
- «**permit**» the client element has access to the supplier element despite the declared visibility of the supplier

Key points



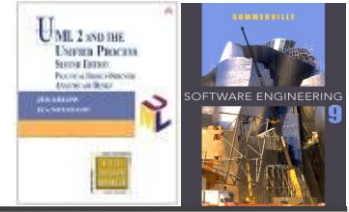
- ✧ Links – relationships between objects
- ✧ Associations – relationships between classes
 - role names
 - multiplicity
 - navigability
 - association classes
- ✧ Dependencies – relationships between model elements
 - usage
 - abstraction
 - permission



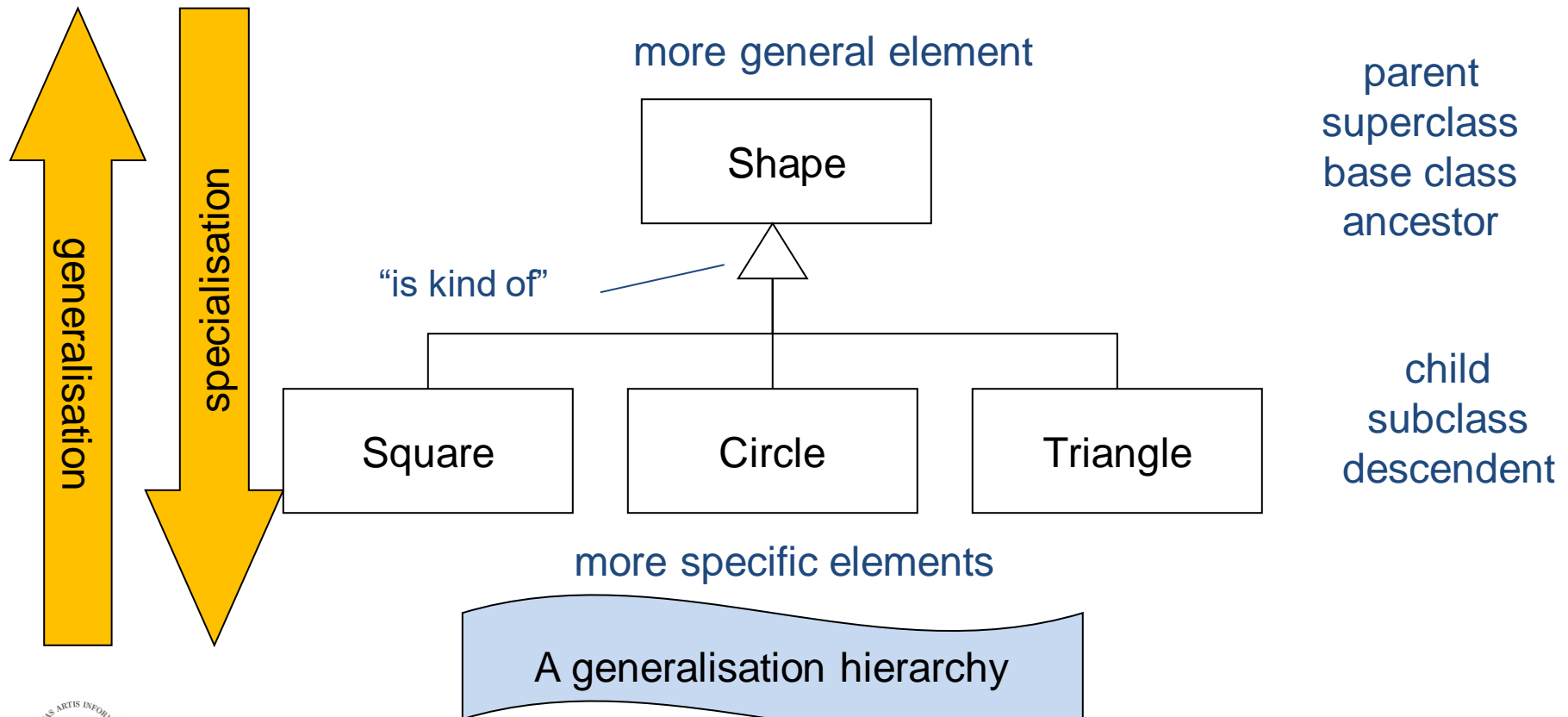
Inheritance and polymorphism

Lecture 3/Part 6

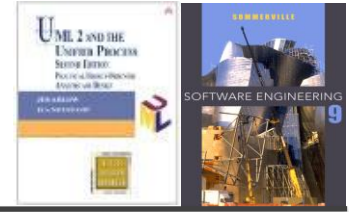
Generalisation



A relationship between a more general element and a more specific element (with more information)

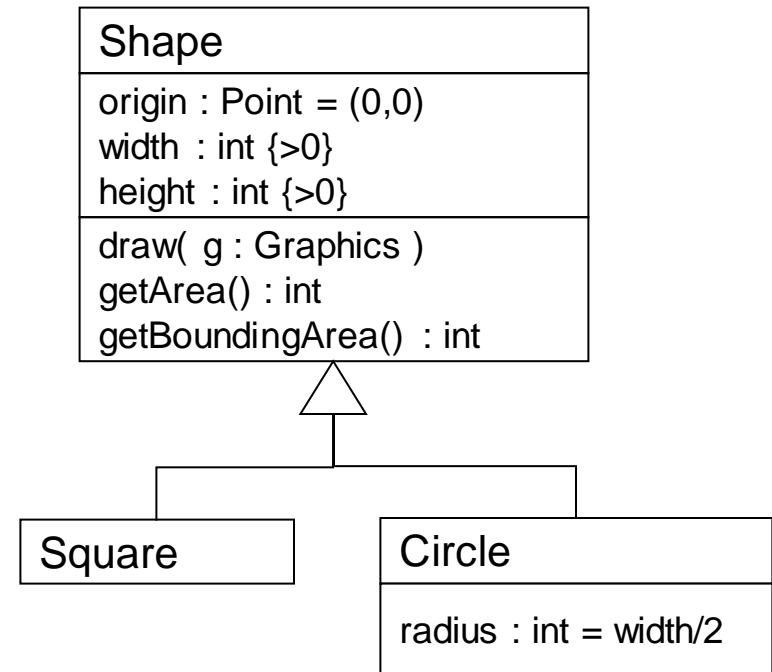


Class inheritance



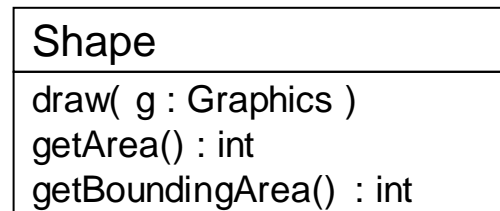
- ✧ Subclasses inherit **all** features of their superclasses:
 - attributes
 - operations
 - relationships
 - stereotypes, tags, constraints
- ✧ Subclasses can add new features
- ✧ Subclasses can override superclass operations
- ✧ We can use a subclass instance **anywhere** a superclass instance is expected

Substitutability Principle

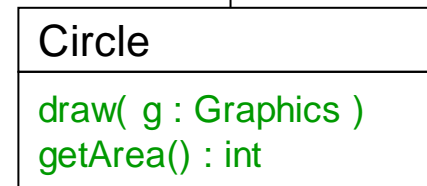
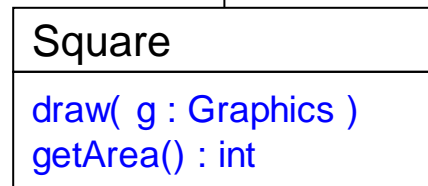
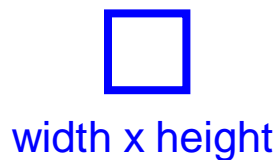


What's wrong with these subclasses?

Overriding



What's wrong with the superclass?



- ✧ Subclasses often need to **override** superclass behaviour
- ✧ To override a superclass operation, a subclass must provide an operation with the same signature
 - The operation signature is the operation name, return type and types of all the parameters

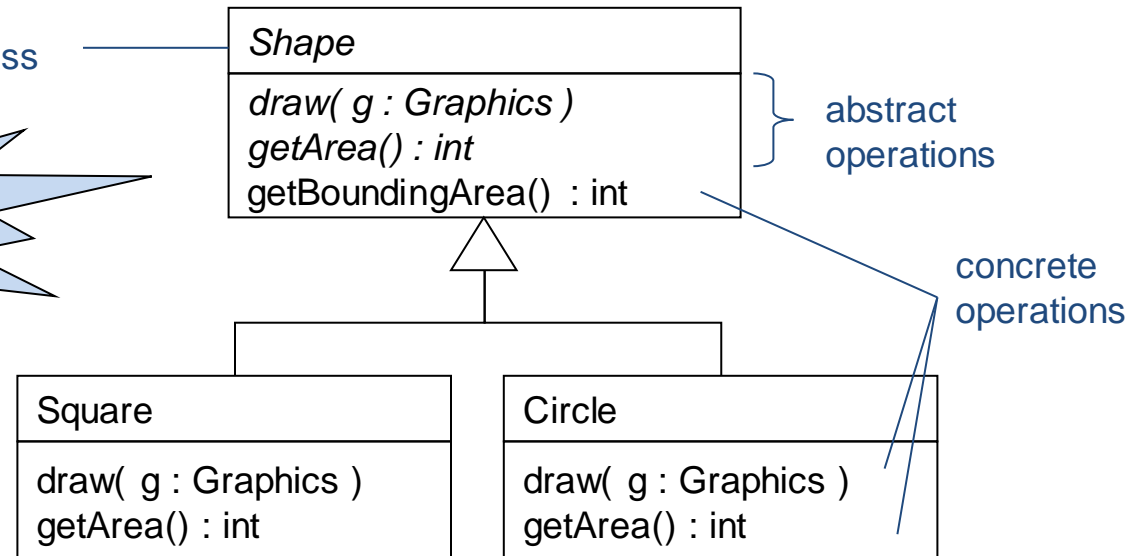
Abstract operations & classes



abstract class

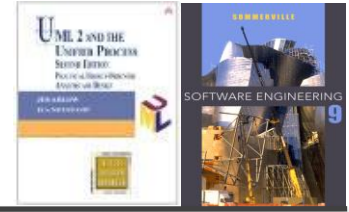
Abstract class and operation names must be in italics

concrete classes

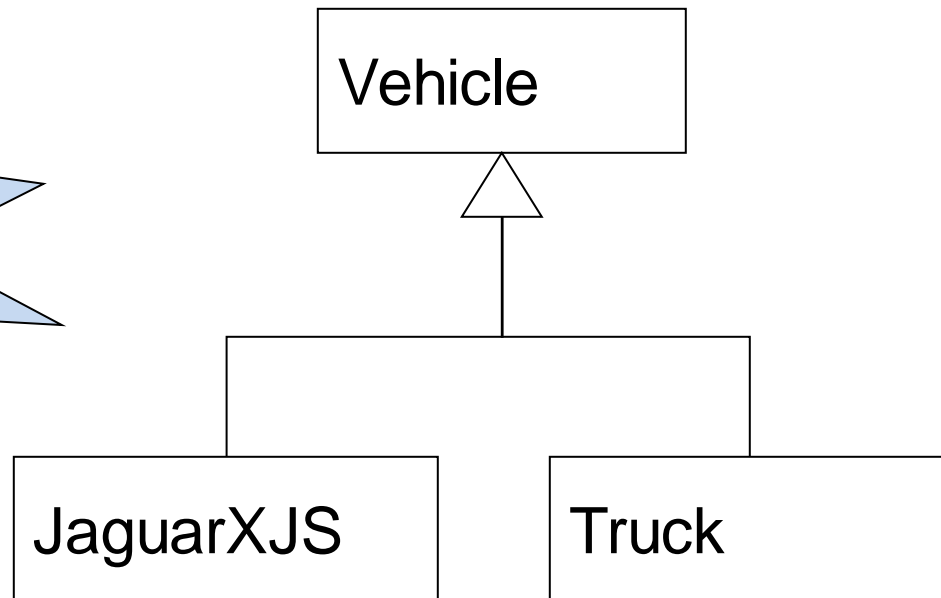


- ✧ We can't provide an implementation for *Shape :: draw(g : Graphics)* or for *Shape :: getArea() : int* because we don't know how to draw or calculate the area for a "shape"!
- ✧ Operations that lack an implementation are **abstract operations**
- ✧ A class with any abstract operations **can't be instantiated** and is therefore an **abstract class**

Exercise



what's wrong with this model?



Polymorphism

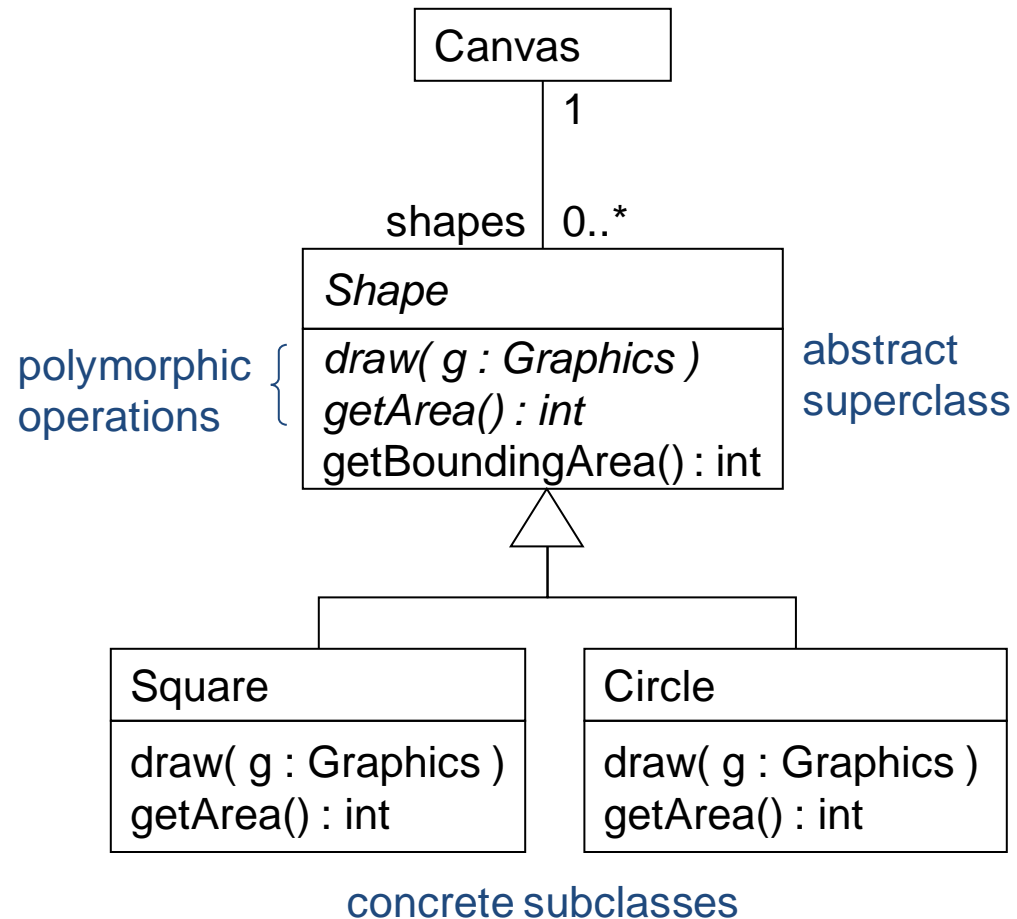


✧ Polymorphism = "many forms"

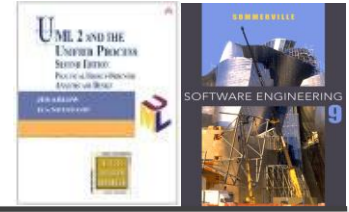
- A polymorphic operation has many implementations
- Square and Circle provide implementations for the polymorphic operations *Shape::draw()* and *Shape::getArea()*

✧ The operation in Shape superclass defines a contract for the subclasses.

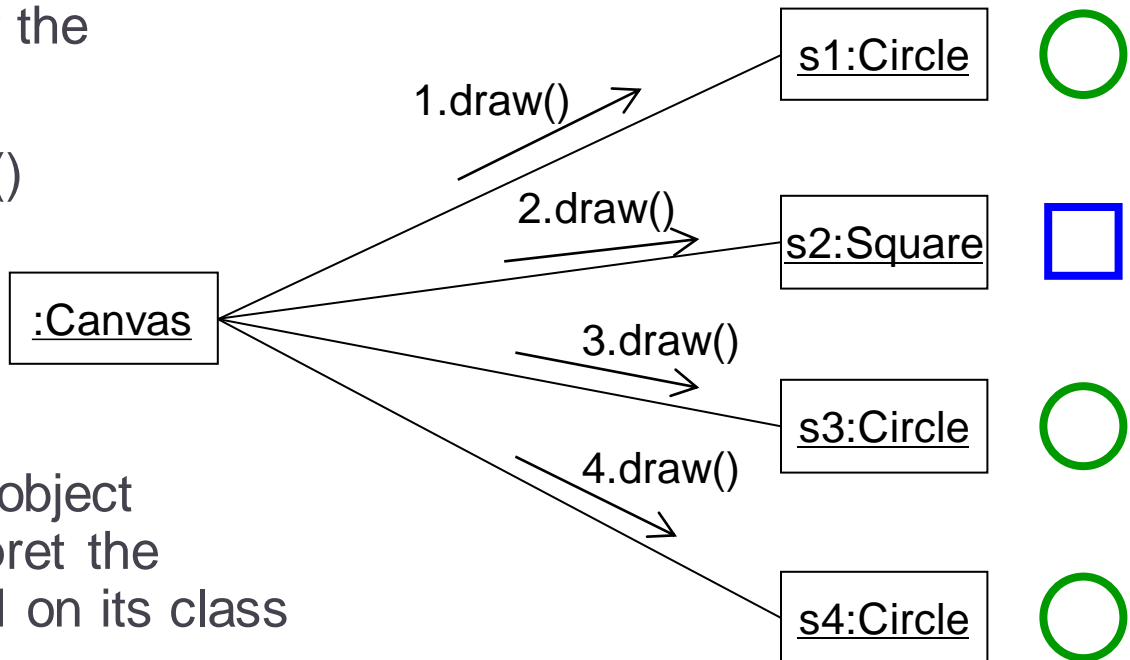
A Canvas object has a collection of *Shape* objects where each *Shape* may be a Square or a Circle



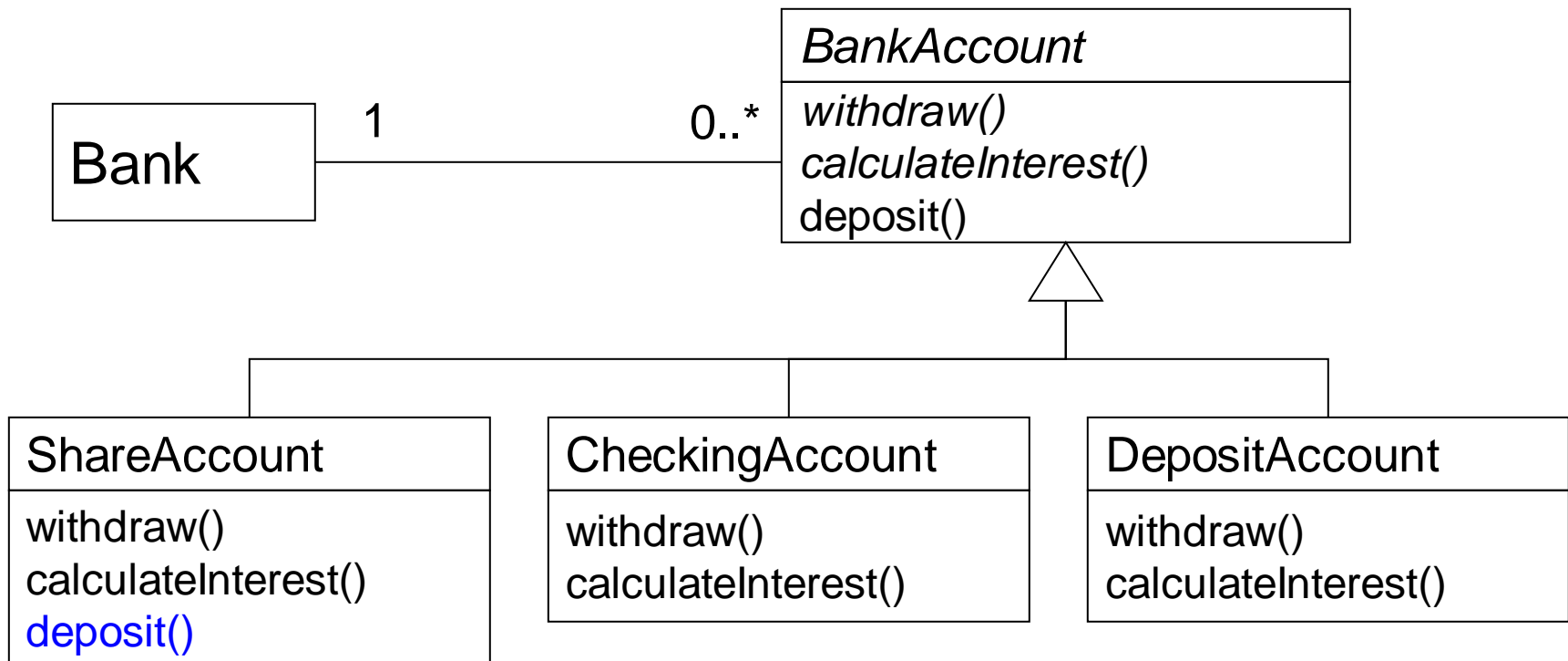
What happens?



- ✧ Each class of object has its own implementation of the draw() operation
- ✧ On receipt of the draw() message, each object invokes the draw() operation specified by its class
- ✧ We can say that each object "decides" how to interpret the draw() message based on its class

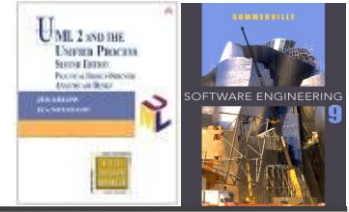


BankAccount example



- ✧ We have overridden the *deposit()* operation even though it is **not** abstract.

Key points



✧ Generalisation, specialisation, inheritance

✧ Subclasses

- inherit all features from their parents including constraints and relationships
- may add **new features**, constraints and relationships
- may **override** superclass operations

✧ A class that can't be instantiated is an abstract class