COMP 1531
Software Engineering Fundamentals

Week 03

Domain Modelling using Object Oriented Design Techniques

Domain model

- Also referred to as a conceptual model or domain object model

- Provides a visual representation of the problem domain, through decomposing the domain into key concepts or objects in the real-world and identifying the relationships between these objects

Requirements Analysis vs Domain modelling

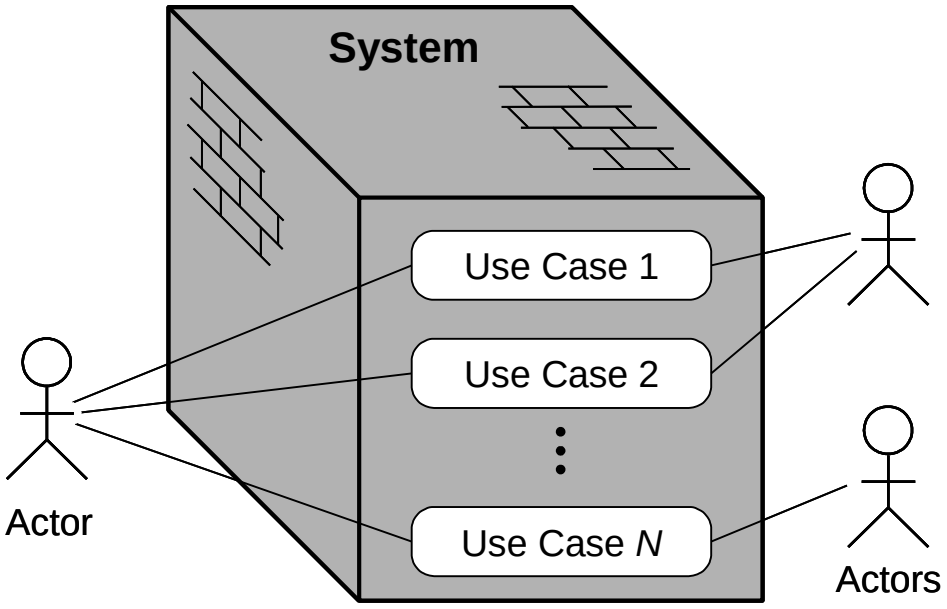
- Requirements analysis determines " external behaviour "
  "What are the features of the system-to-be and who requires these features (actors) "
- Domain modelling determines "internal behavior" - "how elements of system-to-be interact to produce the external behaviour"
- Requirements analysis and domain modelling are mutually dependent - domain modelling supports clarification of requirements, whereas requirements help building up the model.

Use Cases vs. Domain Model

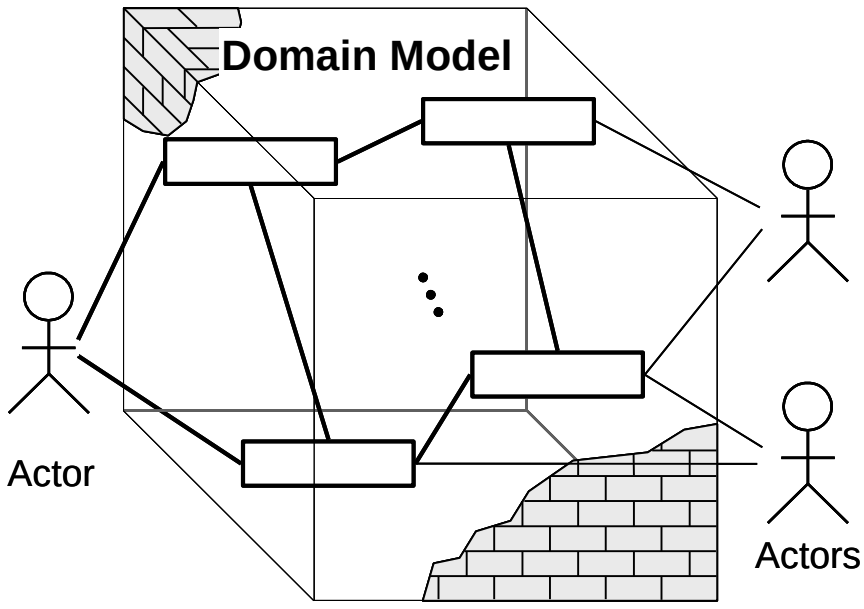
In **use case analysis**, we consider the system as a "**black box**"

In **domain analysis**, we consider the system as a "**transparent box**"

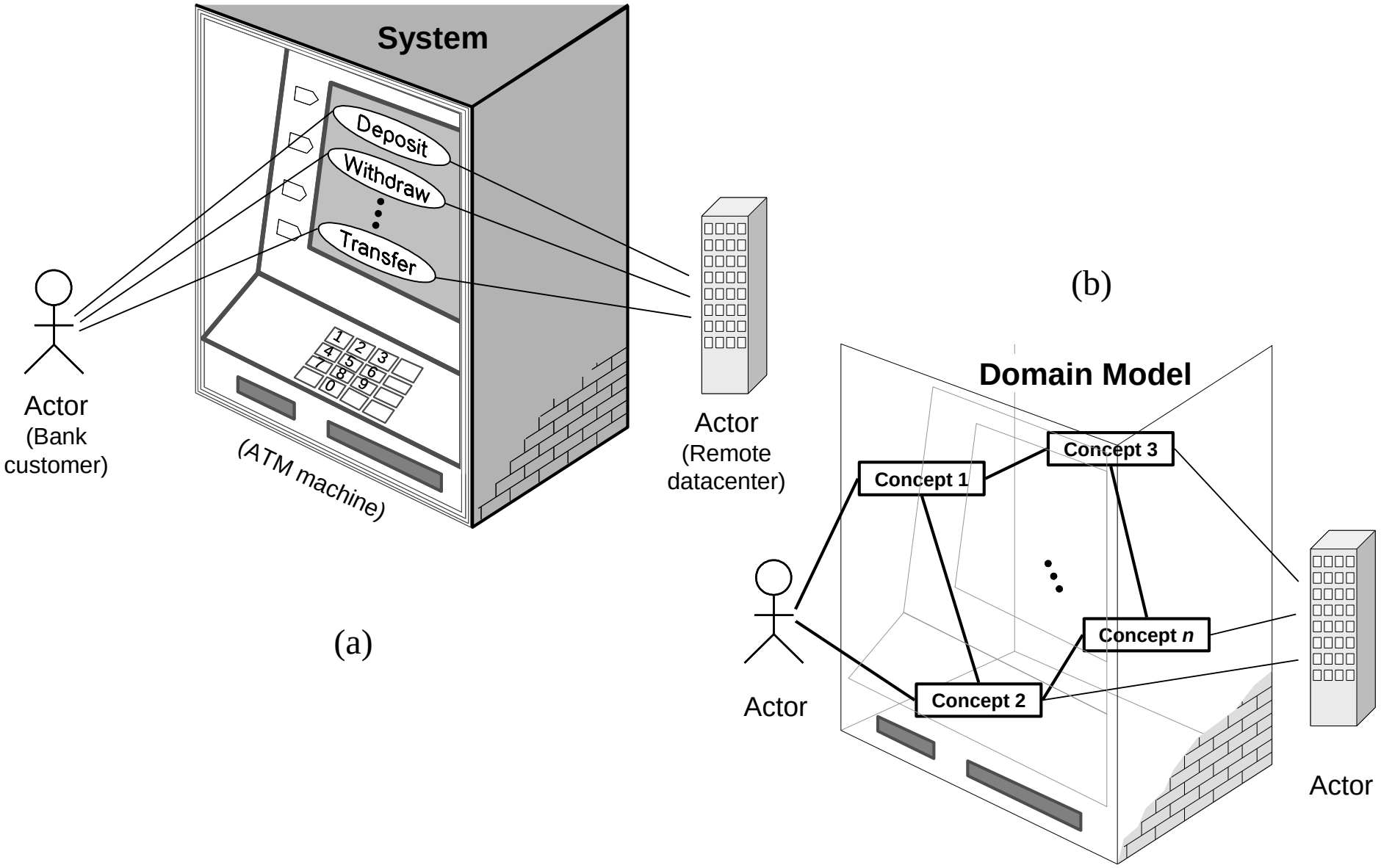
(a)



(b)



Example: ATM Machine



Benefits of a Domain model

- Triggers high-level discussions about what is central to the problem (the core domain) and relationships between sub-parts (sub-domains)

- Ensures that the system-to-be reflects a deep, shared understanding of the problem domain as the objects in the domain model will represent domain concepts

- Importantly, the common language resulting from the domain model, fosters unambiguous shared understanding of the problem domain and requirements among business visionaries, domain experts and developers

How do we create a domain model?

- One widely adopted technique is based on the **object-oriented design** paradigm

But, before we discuss OO, a brief introduction to UML

What is UML?

UML stands for **Unified Modelling Language** (http://www.uml.org/)

Programming languages not abstract enough for OO design

An open source, graphical language to model software solutions, application structures, system behaviour and business processes

Several uses:

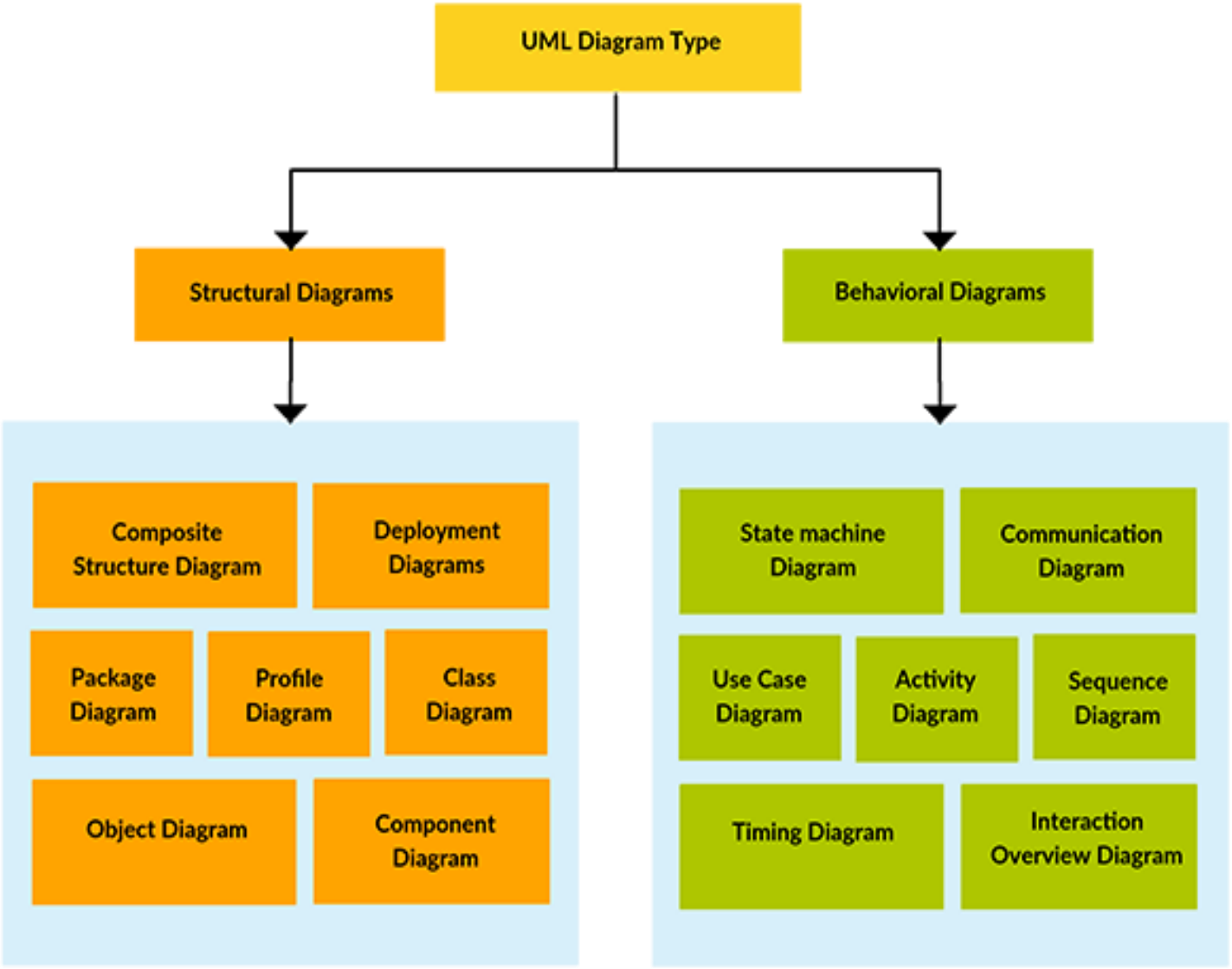
- As a design that communicates aspects of your system
- As a software blue print
- Sometimes, used for auto code-generation

UML diagram categories

- **Structure Diagrams**
  - show the static structure of the system and its parts and how these parts relate to each other
  - they are said to be static as the elements are depicted irrespective of time (e.g. class diagram)

- **Behaviour Diagrams**
  - show the dynamic behaviour of the objects in the system i.e. a series of changes to the system over a period of time (e.g. use case diagram or sequence diagram)
  - a subset of these diagrams are referred to as interaction diagrams that emphasis interaction between objects (e.g., an activity diagram)

UML Diagram Types



Domain Modelling using OO Design Paradigm

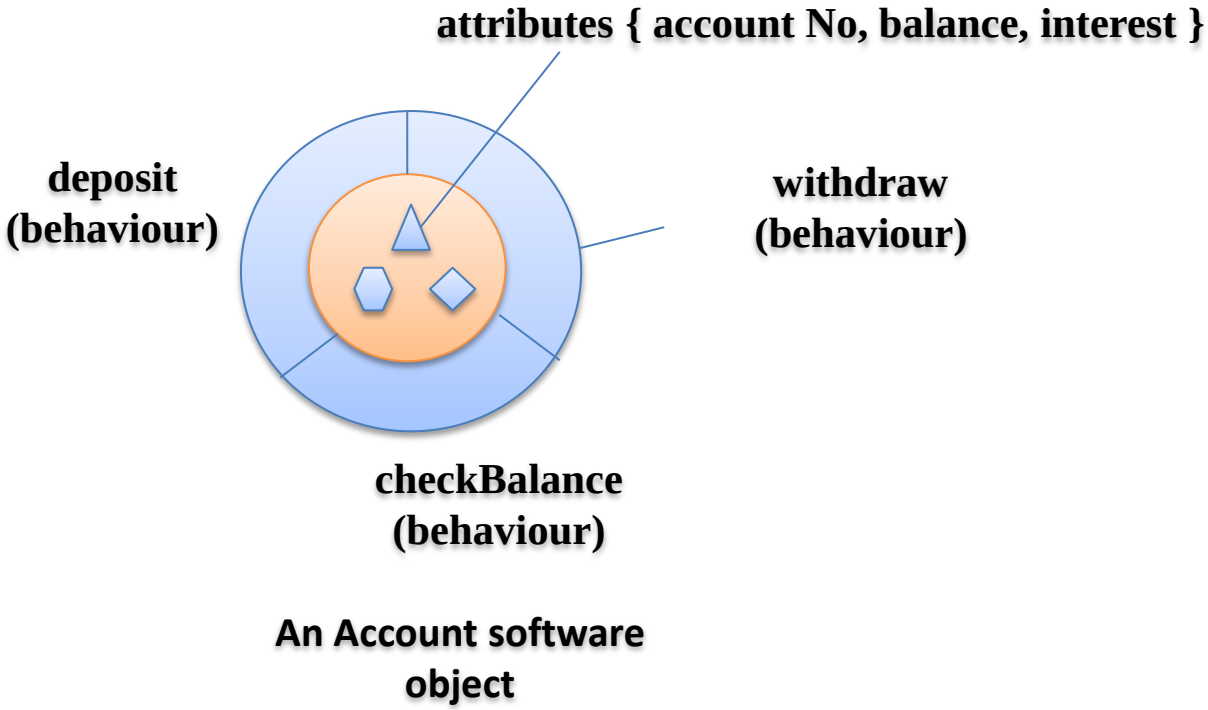
Object Oriented Design

- ***Objects*** are real-world entities and could be
  - Something tangible and visible e.g., your car, phone, apple or pet.
  - Something intangible (you can't touch) e.g., account, time

- Every object has:
  - *attributes:* properties of the object e.g., model number, colour, registration of a car or colour, age, breed of a dog
  - *behaviour* – what the object can do (or methods) e.g., a duck can *fly*, a dog can *bark,* you can *withdraw* or *deposit* into an account
- Each object encapsulates some *state* (the *currently assigned values* for its attributes) ; gives the object its identity *(as state of one object is independent of another)*

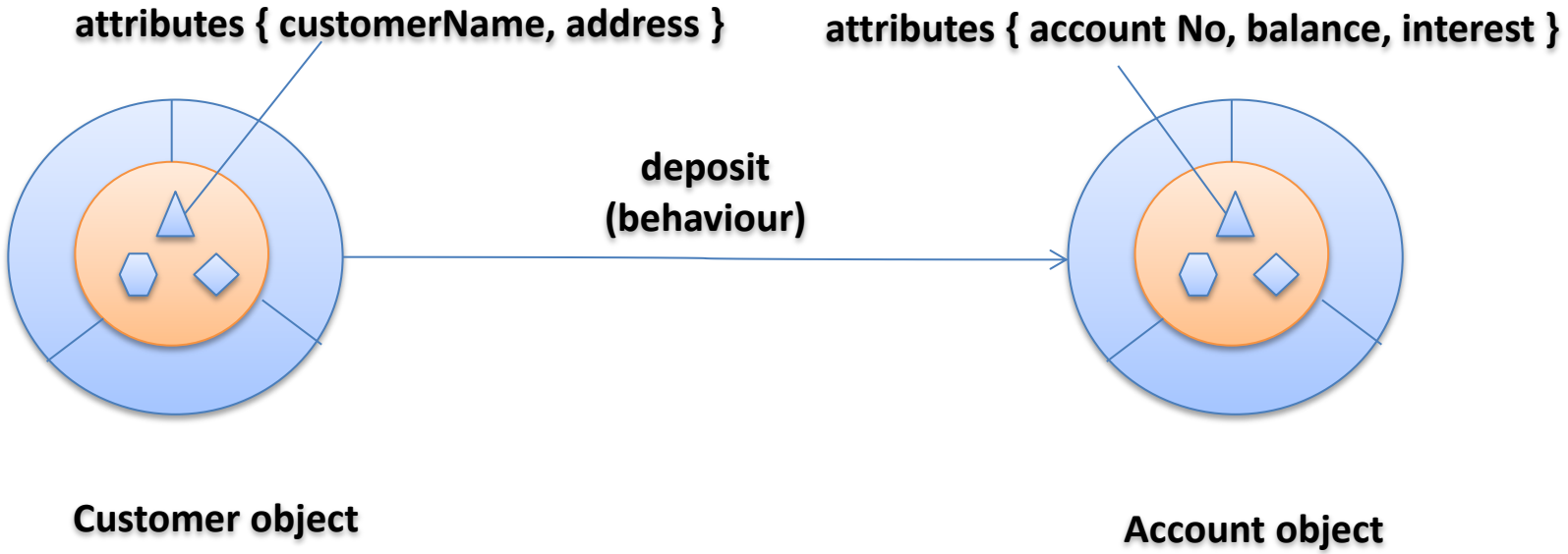
Object Oriented Design

- Identify your domain
- Identify objects



Object collaboration

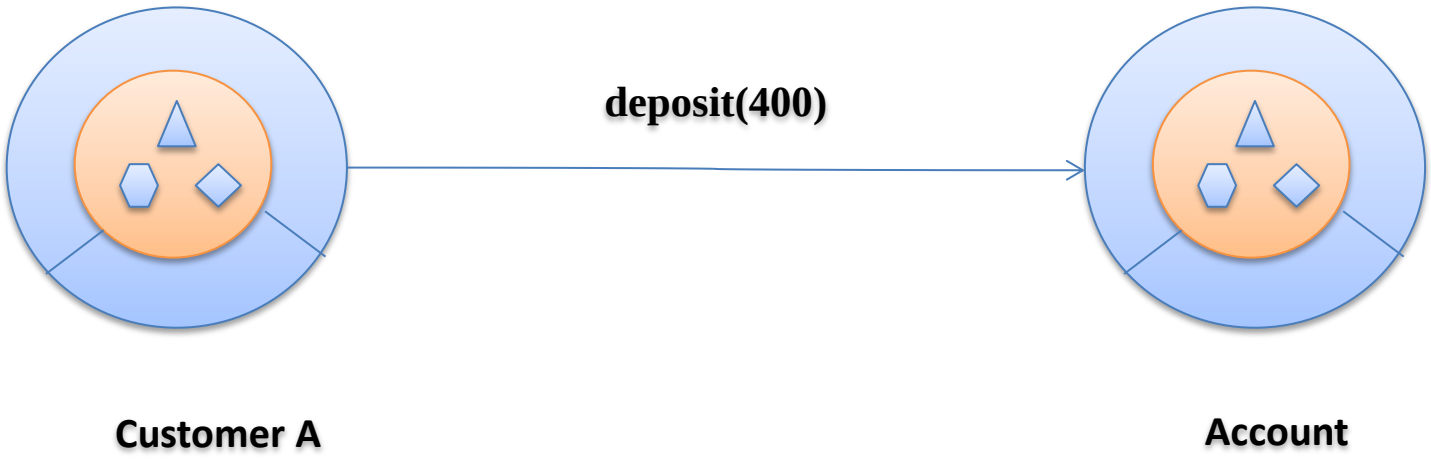
- Objects interact and communicate by sending *messages* to each other
- Objects play a *client* and *server* role and could be located in the same memory space or on different computers
- If *object A* wants to invoke a specific behaviour on *object B*, it sends a message to *B* requesting that behaviour



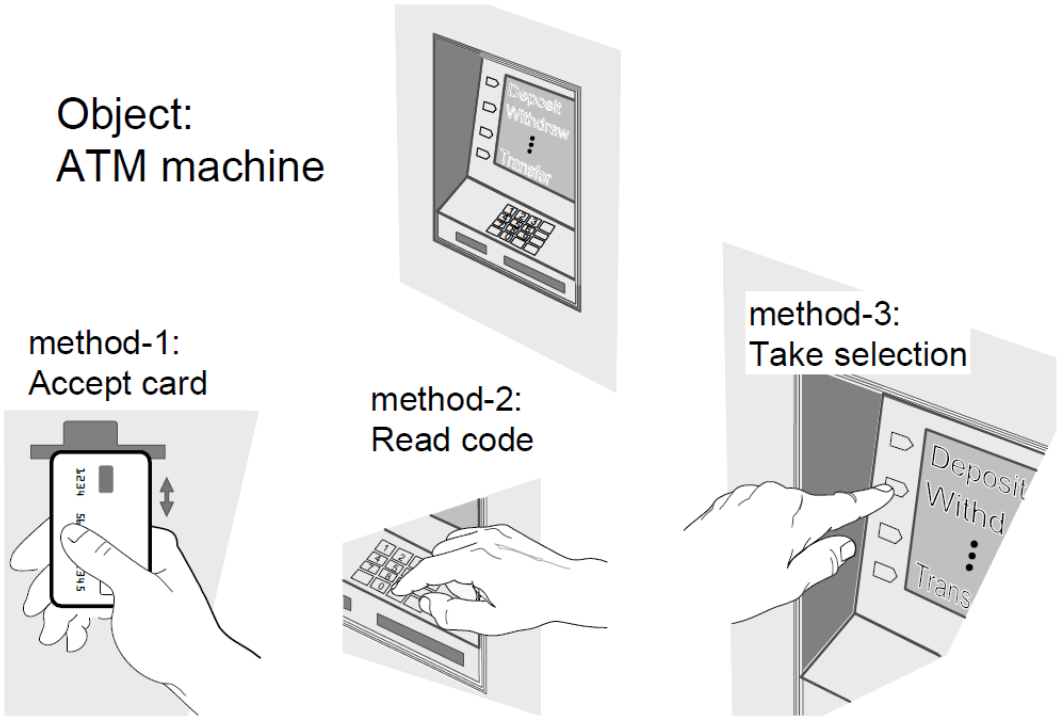
Object Collaboration

This message is typically made of three parts:

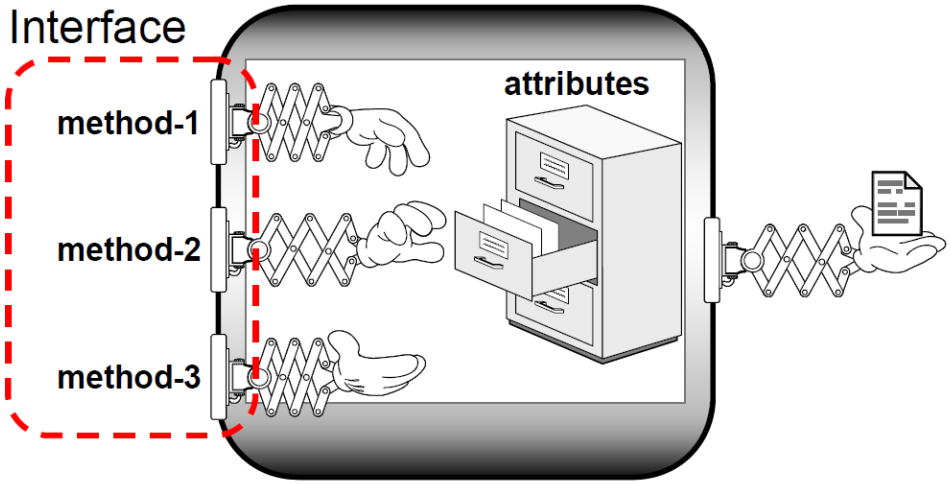
- The *object* to whom the message is addressed (e.g., John's "Account" object)
- The *method* you want to invoke on the object(e.g., deposit())
- Any *additional information* needed (e.g., amount to be deposited)



Object's Interface

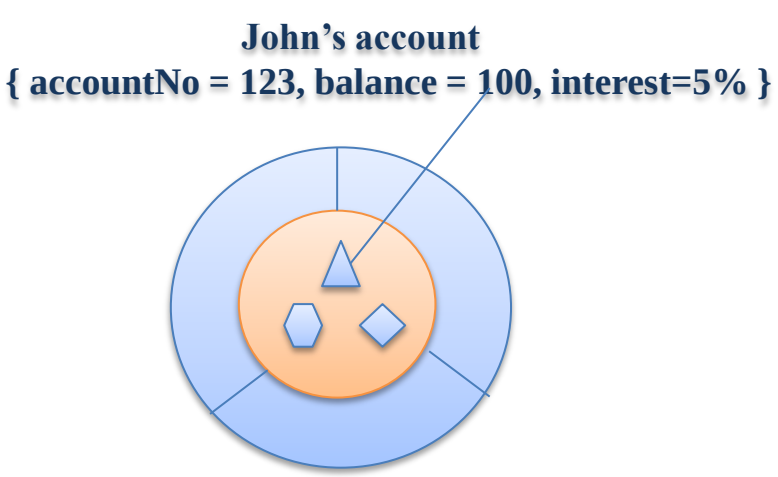


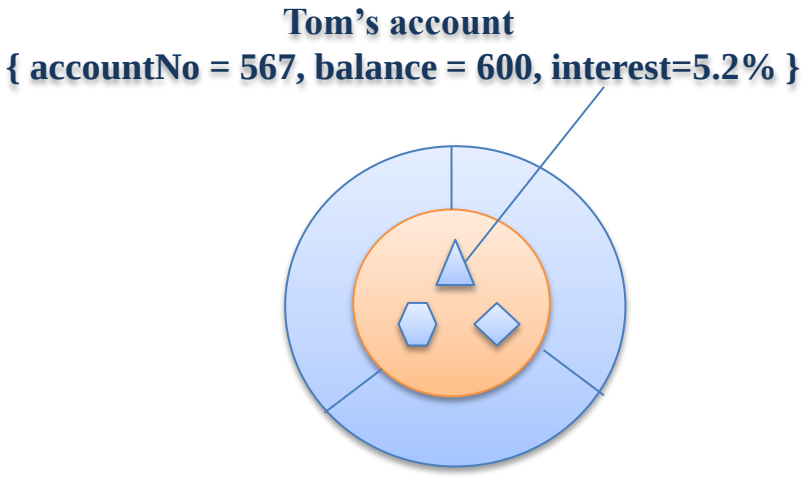
- An object's *interface* is the set of the object's methods that can be invoked on the object
- The interface is the fundamental means of communication between objects



Objects and Classes

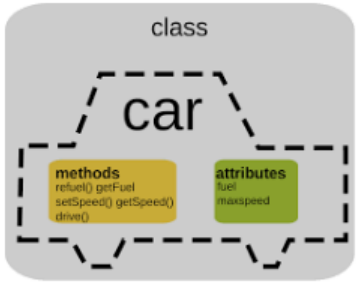
- Many objects are of the same "**kind**" but have different identity e.g., *there are many Account objects belonging to different customers, but they all share the same attributes and methods*
- "Logically group" objects that share some common properties and behavior into a **class**
- A class serves as a *blue-print* defining the attributes and methods (behaviour) of this logical group of objects

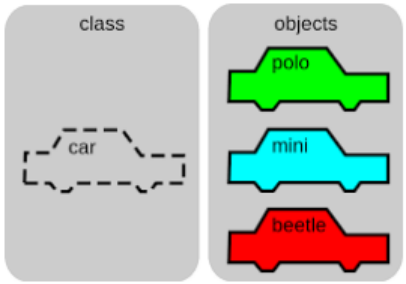




Objects and classes

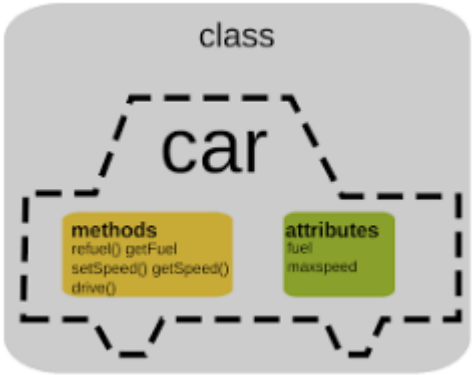
- Defining a class, does not actually create an object
- An object is instantiated from a class and the object is said to be an instance of the class
  - An object instance is a specific realization of the class
- Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each other
  - An object has state but a class doesn't
  - Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each other

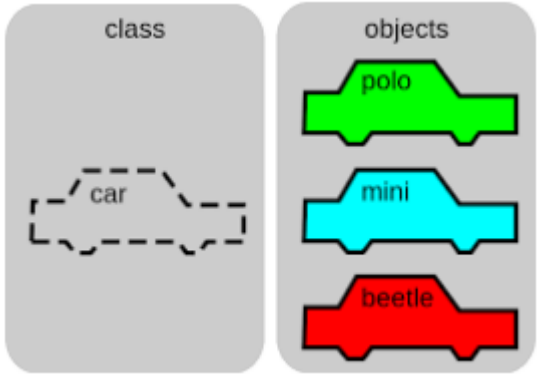




Objects and Classes

- We tend to “logically group” objects that share some common properties and behaviour. This logical group is called a **class**
- A class serves as a *blue-print* defining the attributes and methods (behaviour) of this logical group of objects
- Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each other

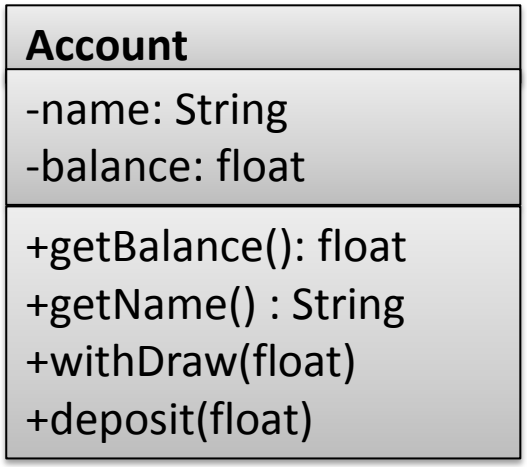




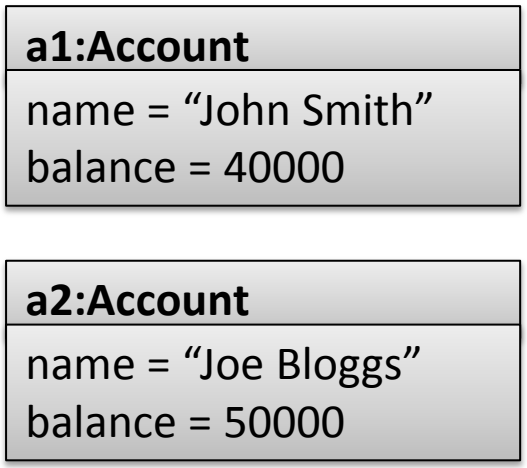
Representing classes in UML

- A class is sometimes referred to as an **object's type**.
- An object is instantiated from a class and the object is said to be an instance of the class)
- An object has state but a class doesn't

class (class diagram)



object instances (object diagram)



Two key principles of OO design

- Abstraction
- Encapsulation

Abstraction

- Helps you to focus on the common properties and behaviours of objects
- Good abstraction help us to accurately represent the knowledge we gather about the problem domain (discard anything unimportant or irrelevant)
- What comes to your mind when we think of a “car” ?

  Do you create a class for each brand (BMW, Audi, Chevrolet…) ?

  - write ***one*** class called Car

  and ***abstract***;

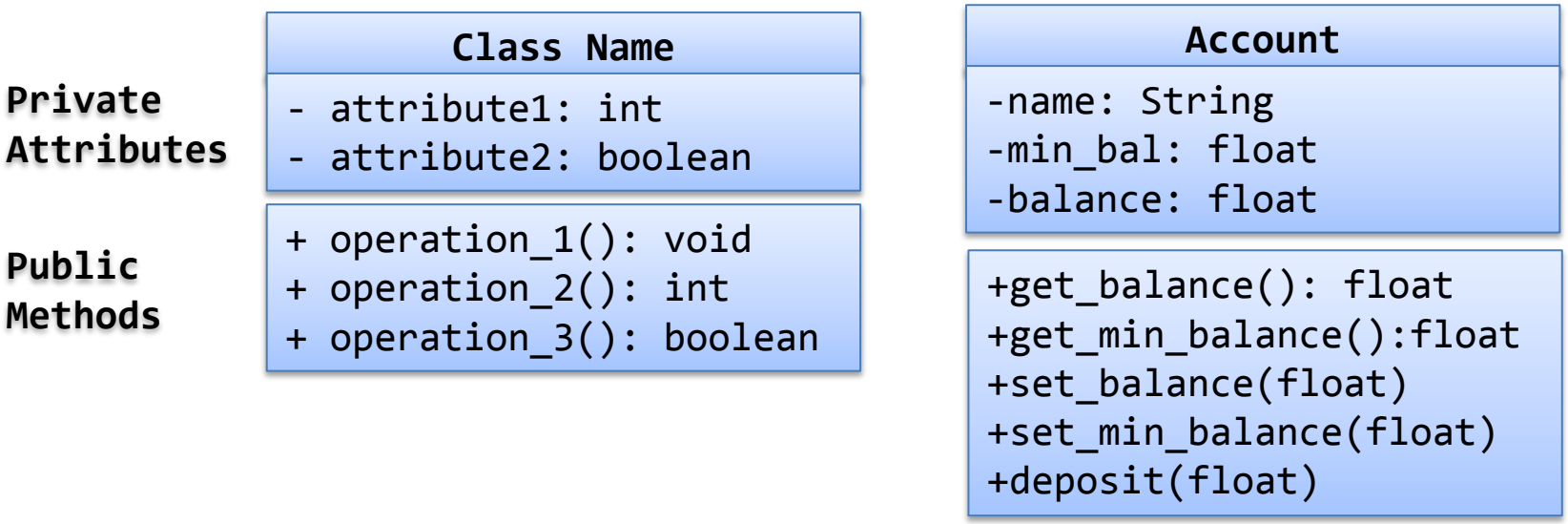
  - focus on the common essential qualities of the object
  - focus on the current application context

- What if a specific brand had a special property or behaviour? Later on….***inheritance***

Encapsulation

- Encapsulation implies ***hiding*** the object state (attributes)
- An object's attributes represent its individual characteristics or properties, so access to the object's data must be restricted
- **Methods** provide explicit access to the object

  e.g. use of *getter* and *setter* methods to access or modify the fields



UML notation for a Class

Why is encapsulation important?

1. Encapsulation ensures that an object's state is in a consistent state
2. Encapsulation increases usability
   - Keeping the data private and exposing the object only through its interface (public methods) provides a clear view of the role of the object and increases usability
   - Clear contract between the invoker and the provider, where the client agrees to invoke an object's method adhering to the method signature and provider guarantees consistent behaviour of the method invoked (if the client invoked the method correctly)
3. Encapsulation abstracts the implementation, reduces the dependencies so that a change to a class does not cause a rippling effect on the system

- So far,
  - we have defined classes and object instances
  - objects have attributes and responsibilities
  - learnt about abstraction and encapsulation
- let us now look at **relationships between objects** e.g.,
  - a dog **is-a** mammal
  - an instructor **teaches** a student
  - a university **enrols** students
- Relationships between objects can be broadly classified as:
  - Inheritance
  - Association

Relationships (1) – Inheritance

- So far, we have logically grouped objects with common characteristics into a class, but what if these objects had some special features?

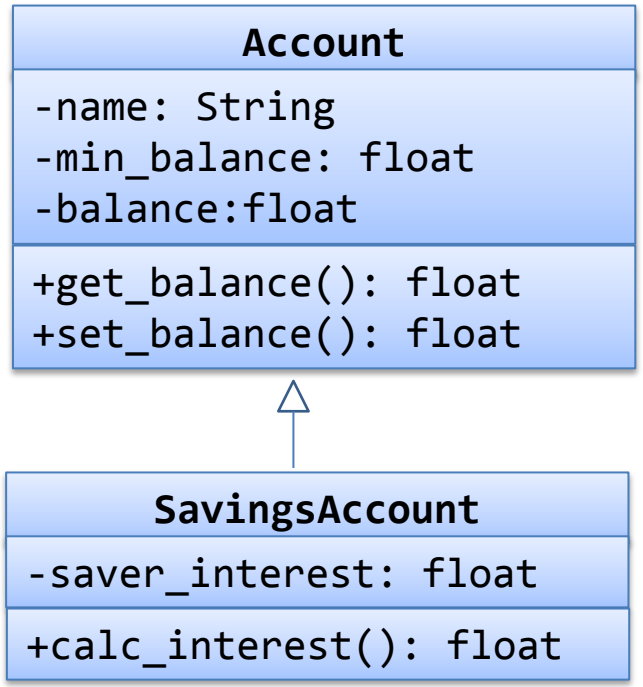
  e.g., if we wanted to store that sports car has spoilers

- Answer is inheritance - models a relationship between classes in which one class represents a more general concept (parent or base class) and another a more specialised class (sub-class)
- Inheritance models a "is-a" type of relationship e.g.,

  - a savings account is a type of bank account
  - a dog **is-a** type of pet
  - a manager **is-a** type of employee
  - a rectangle **is-a** type of 2D shape

Inheritance

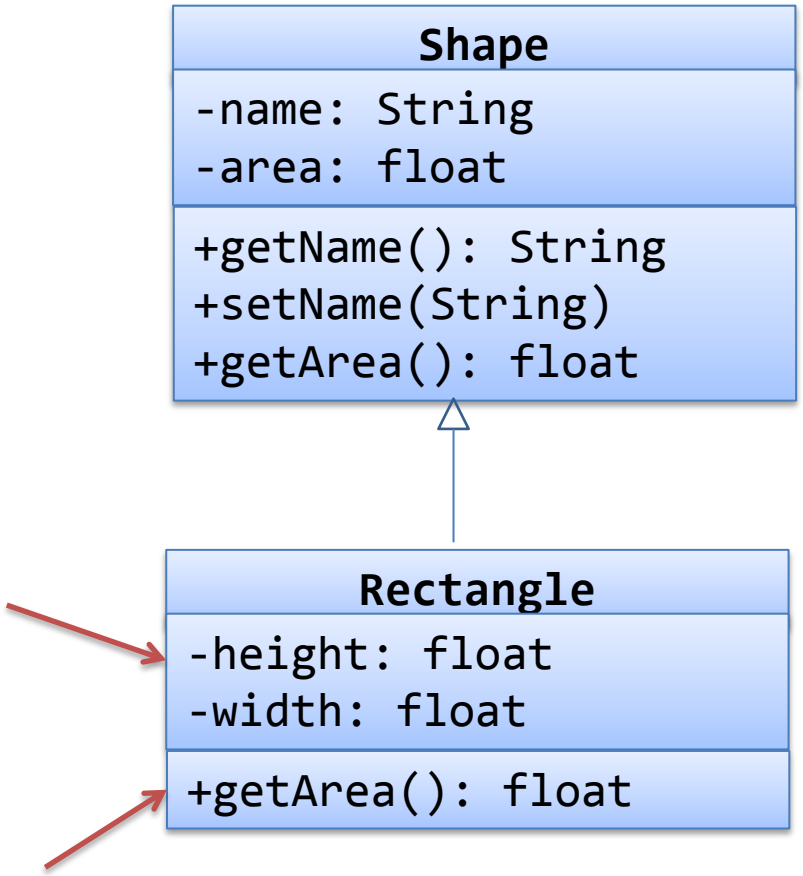
- To implement inheritance, we
  - create a new class (sub-class) , that inherits common properties and behaviour from a base class (parent-class or super-class)
    - We say the child class *inherits/ is-derived from* the parent class
  - sub-class can *extend* the parent class by defining additional properties and behaviour specific to the inherited group of objects
  - sub-class can *override* methods in the parent class with their own specialised behaviour



Parent class - Account
class Account defines *name, min_bal, balance*

Child class – SavingsAccount
extends Account class adding its own attributes and methods e.g., *saver_interest & calc_interest()*

Inheritance – another example



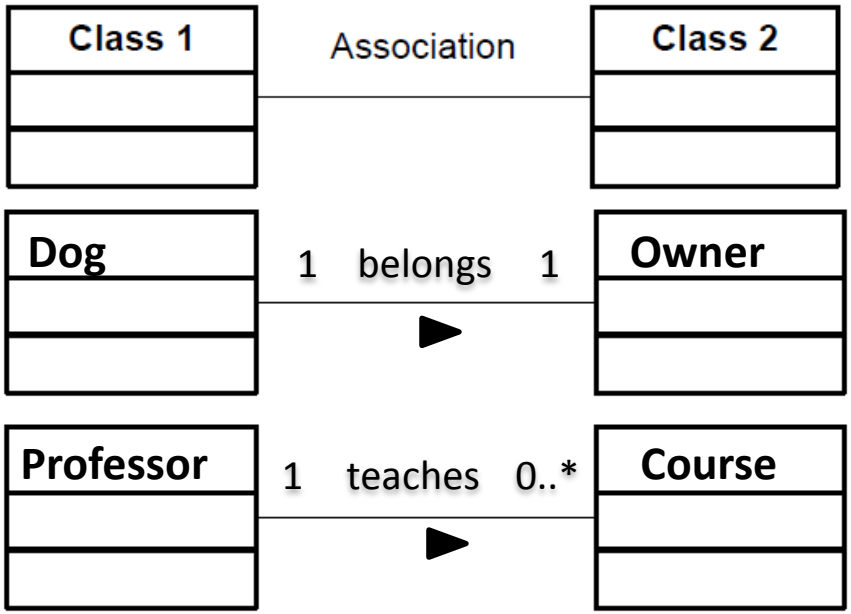
class Rectangle *extends* Shape adding attributes *height* , *width*

class Rectangle *overrides* method *getArea()* to provide its own implementation

Relationships (2) – Association

- Association is a special type of relationship between two classes, where one object instance shares a "uses" relationship with another object instance
  - One object instance "uses" the services or functionality provided by the other object instance

    e.g., a lecturer *teaches* a course-offering
- Association is modelled in UML as a line between two classes

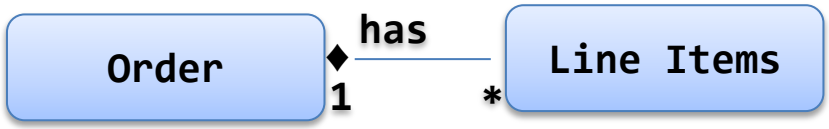


Relationships – Aggregation & Composition

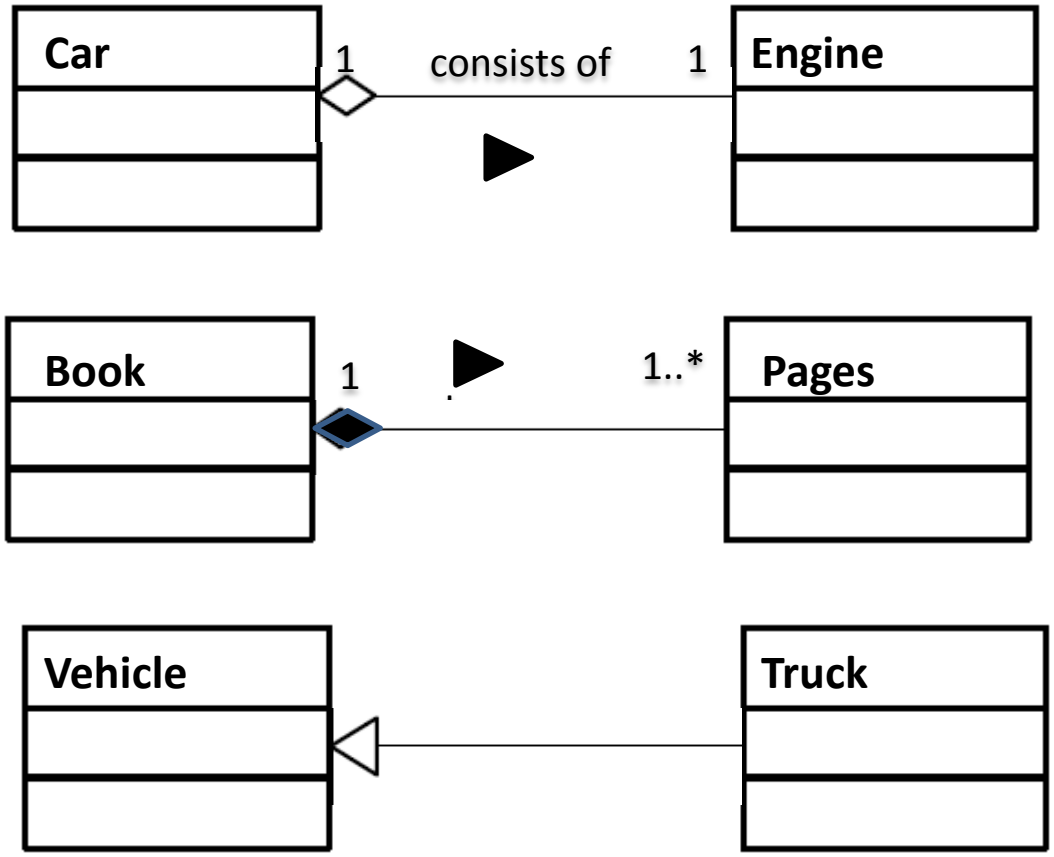
When associations model a **"has-a"** or "whole-part" relationship where one class (the whole) "contains" another class (the part), they can be further refined as:

Aggregation relationship (*denoted in UML by a line terminated with a hollow diamond symbol ◊ on the container side*): The contained item (part) is an element of a collection (whole) but it can also exist on its own, e.g., a lecturer in a university or a student at a university

Composition relationship (*denoted in UML by a line terminated with a filled diamond symbol ♦ on the container side*): The contained item is an integral part of the containing item, such as a leg in a desk, or engine in a car and the existence of the part is meaningless, when the container is destroyed



More examples of relationships



- Aggregation - “has-a” relationship where the part can exist without container
- Composition – “is-composed-of” relationship where part cannot live without container
- Inheritance – “is-a-kind-of” relationship

Re-visiting our earlier question:

How to create a domain model?

So far, requirements engineering helped us to

- Understand the problem domain
- Establish knowledge of how system-to-be is supposed to behave (from requirements analysis, e.g., use cases and sequence diagrams)
- We now apply OO design principles to build a domain model

Next,

- Noun/Verb analysis
- CRC cards
- Domain model

Domain Modelling Techniques (2) – Using CRC cards

- A simple tool for system analysis and design
- Part of the OO design paradigm and features prominently as a design technique in XP programming
- Written in 4 by 6 index cards, an individual CRC card use to represent a domain object
- The output of this tool – definition of classes and their relationships
- Benefits – Easy to use, forces you to be concise and clear

Domain Modelling Techniques (2) – Using CRC cards

CRC stands for **C**lass, **R**esponsibility and **C**ollaboration

Class:

- An object-oriented class name and represents a collection of similar objects

Responsibility

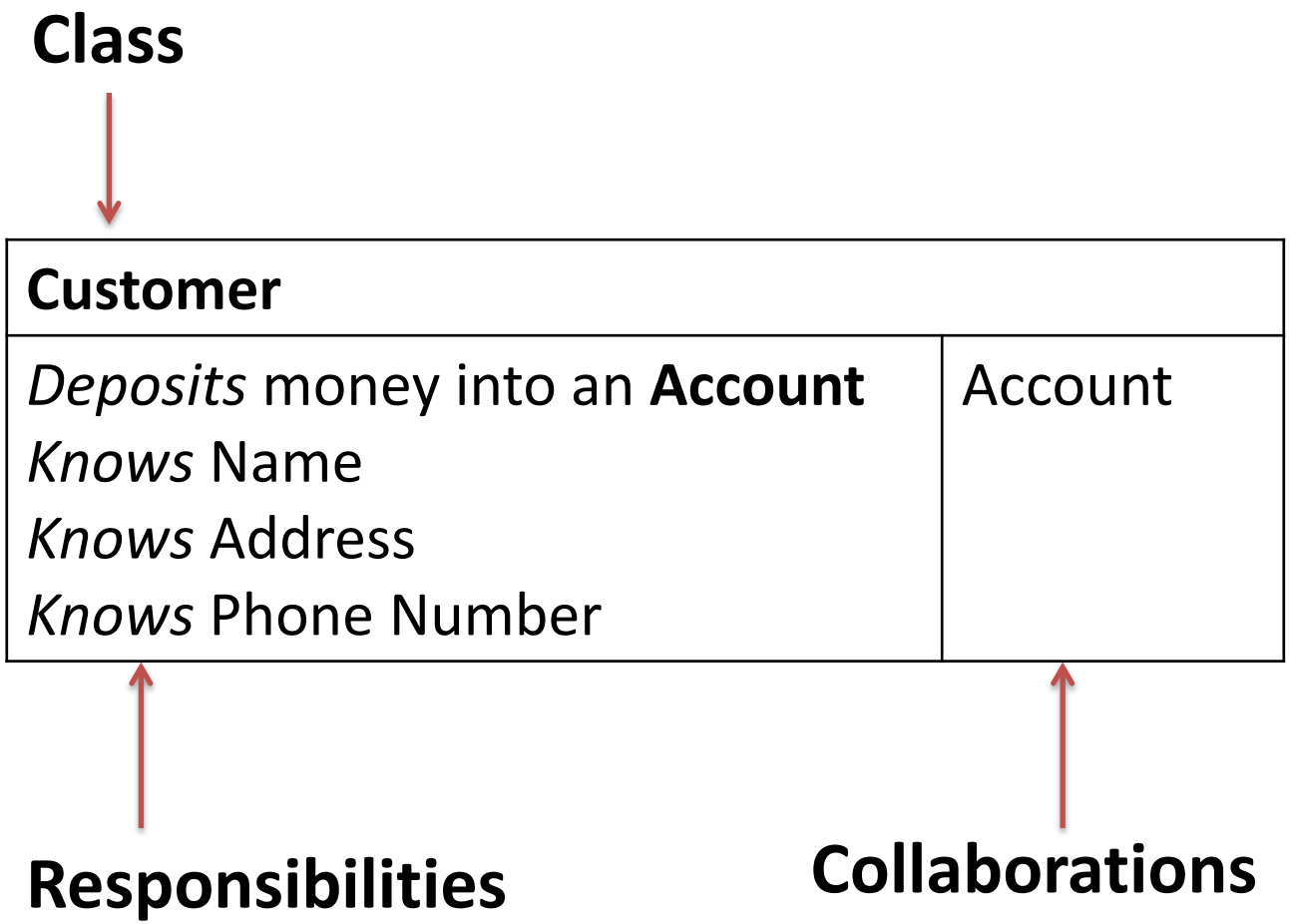
- What does this class know? (the **knowledge** that the object maintains)
- What does this class do? (the **actions** the object can perform)

Collaboration:

- Relationship to other classes (What classes does this class use?

Make sure all your classes are doing something different

Domain Modelling Techniques (2) – Using CRC cards



A **CRC Model** is a collection of CRC cards, that specifies the OO design of the system

How to create a CRC model?

You are given a description of the requirements of a software system

Gather around a table with a set of index cards and pen

As a team:

- Read the description
- Identify core classes (look for nouns)
- Create a card per class (start with just class names)
- Add responsibilities (look for verbs)
- Add collaborations
- Add more classes as you discover them
- Refine by identifying inheritance etc.

Key points to remember, when writing CRC cards (1)

Assigning Responsibilities

- Evenly distribute system intelligence
- Keep behaviour with related information (if any)
- Share responsibilities among shared objects

Look at **relationships between classes**, to spot examples of:

- is-kind-of
  - maybe superclass should have responsibility?
- is-analogous-to
  - if have similar responsibilities, perhaps should be common superclass with it?
- is-part-of
  - therefore clarify responsibilities between parts of an aggregate class

Key points to remember, when writing CRC cards (2)

Identifying Collaborations: Collaboration is **strongly** indicated by:

- has-knowledge-of e.g. a car needs to know the speed limit, which it gets from the sign on the side of the road or ...
- depends-on (changes-with) e.g. pressing accelerator increases speed of wheels and decreases petrol level (but beware INDIRECT collaborations)
- (composite) is-part-of e.g. to turn a car, the car will need to send messages to its steering wheel, wheels

How can you tell it works?

A neat technique: a **Scenario Walk-through**.

- Select a set of scenarios (use cases).
- Choose a plausible set of inputs for each scenario.
- Manually “execute” each scenario.
- Start with initial input for scenario and find a class that has responsibility for responding to that input.
- Trace through the collaborations of each class that participates in satisfying that responsibility.
- Make adjustments and refinements as necessary.

Techniques for Domain Modelling (2)

Evolve CRC domain models into UML class diagrams where:

- Concepts are represented as classes
- Collaborations between the classes established as relationships
- Depending on the kind of relationship, we can use the different notations that we've used for associations – non-hierarchical, part-of (aggregation), is-a (inheritance),

Lecture Demo

- Creating a CRC Model
- Creating a class diagram

Case Study -1

Problem Statement:

UNSW would like to build a library system that enables staff and students to *borrow* books and videos. Staff can borrow a maximum of 10 items and students can borrow a maximum of 5 items. For each item borrowed or returned, the borrower needs to go through a librarian, who will *record* the borrowings and returns. For each book, the title, author, if the book is available for borrowing needs to be stored (status).