Domain Modelling using Object-Oriented Design Techniques

COMP 1531, 17s2

Aarthi Natarajan

Week 4

Domain model

- **Wikipedia Definition**
  - Is a system of abstractions that describes selected aspects of a sphere of knowledge, influence or activity and is used to solve problems related to that domain.
  - Is a representation of meaningful real-world concepts pertinent to the domain that need to be modelled in software where the concepts include the data involved in the business and rules the business uses in relation to that data.
  - Uses the vocabulary of the domain so that a representation of the model can be used to communicate with non-technical stakeholders.

- Is created to provide 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

- Various names – conceptual model, domain object model.

Why do we create a Domain model?

- ❖ The goal of domain modeling is to understand how system-to-be will work
  - – Requirements analysis determined how users will interact with system-to-be (external behavior)
  - – Domain modeling determines how elements of system-to-be interact (internal behavior) to produce the external behavior

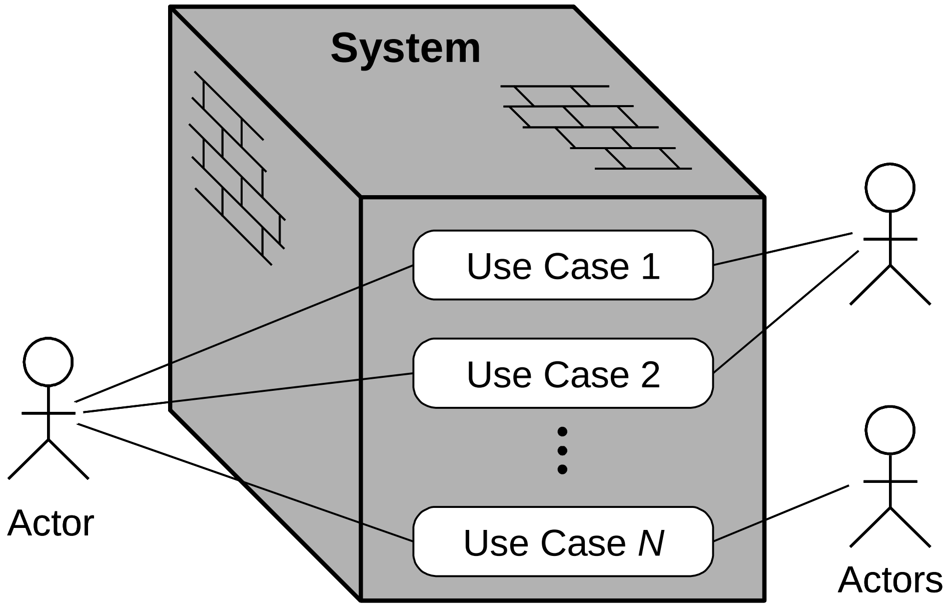
- ❖ Requirements and domain modelling are mutually dependent
  - ⁻ Domain modelling supports clarification of requirements, whereas requirements help building up and clarifying the model.

Use Cases vs. Domain Model

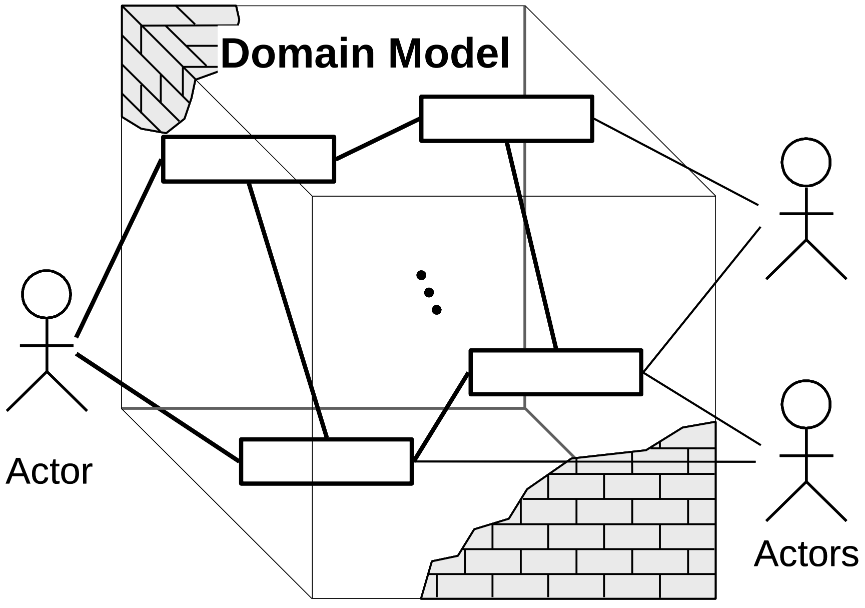
In **use case analysis**, we consider the system as a “**black box**”

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

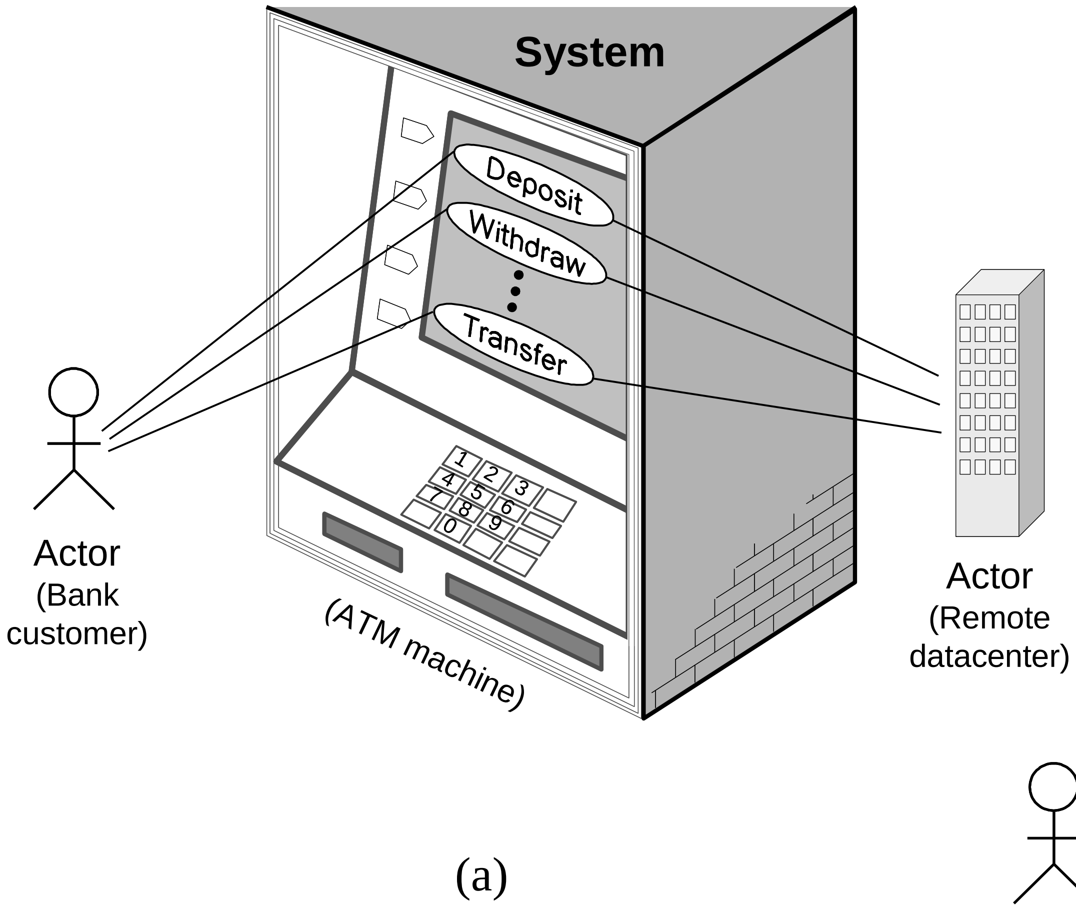
(a)

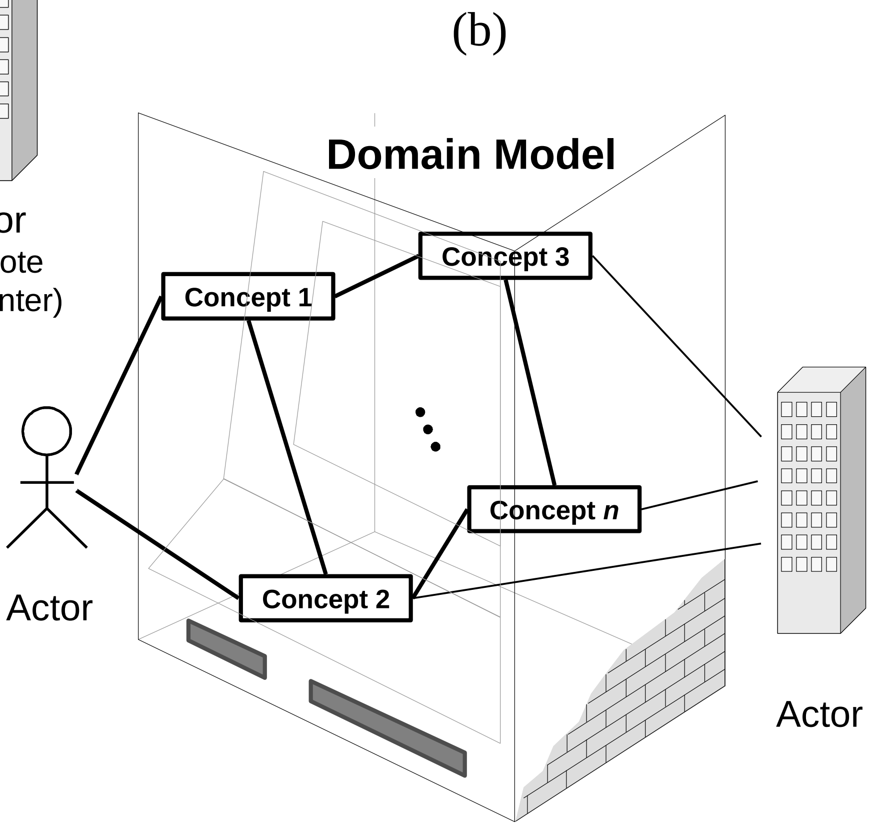


(b)



Example: ATM Machine



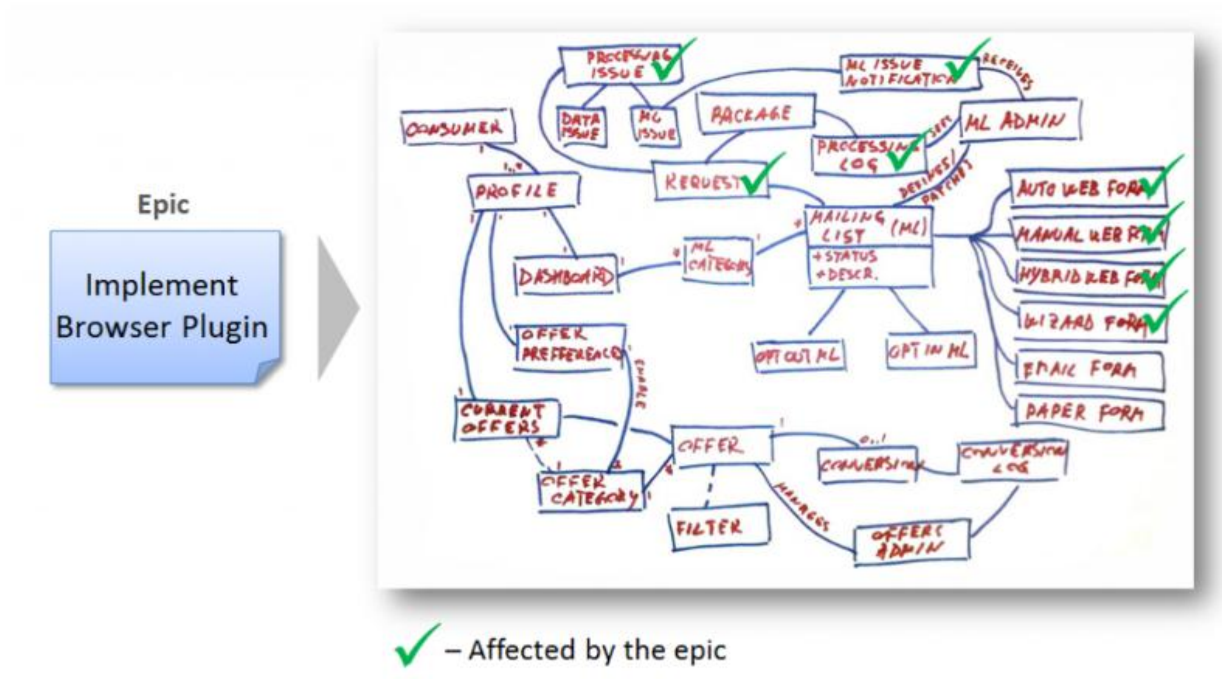


Benefits of a Domain model

- **Benefits of Domain Modelling**
  - 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

Domain Modelling in Agile Development

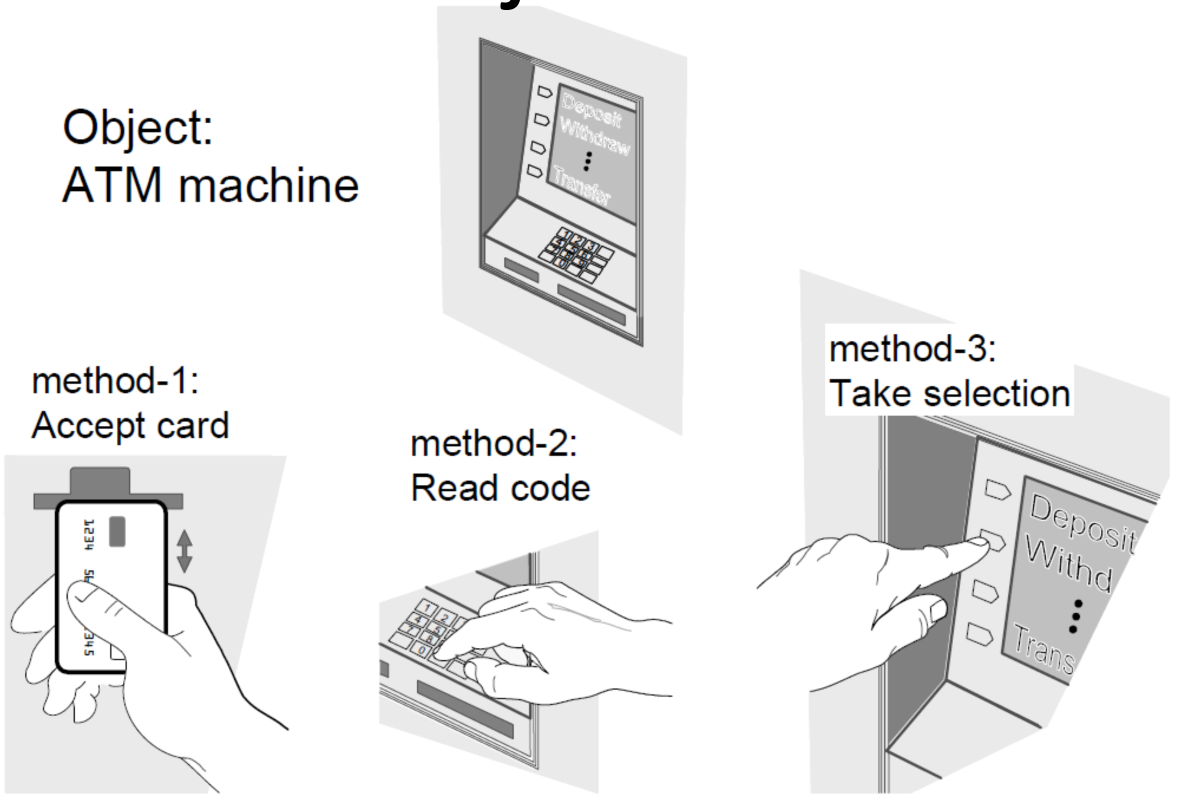
- ❖ **Domain Modelling is** a primary design technique used in Agile software development as it helps to bridge the gap between understanding the problem domain and the interpretation of requirements for the system-to-be
- ❖ A domain model -
  - Helps in understanding the scope of an epic
  - Is developed and continuously refined by the team in collaboration with the stake-holders to understand the impact of epics and features on the system

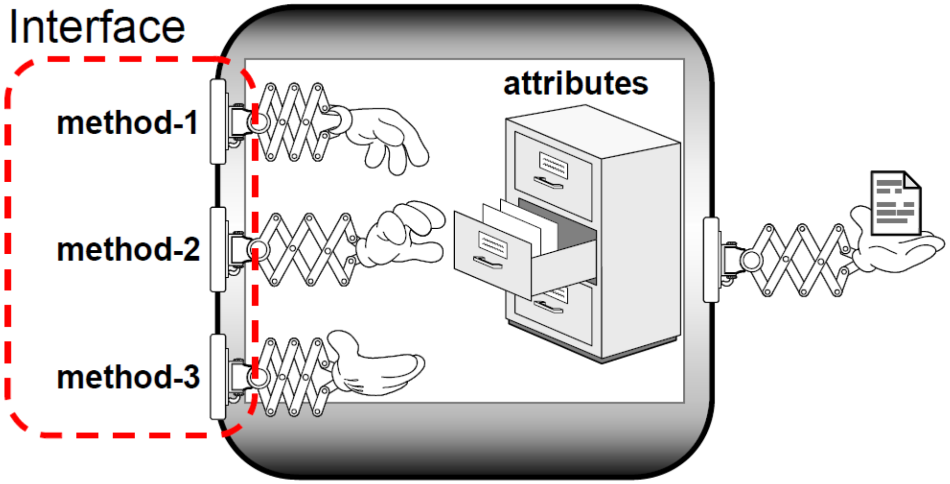


Object Oriented Design

- ***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
- Each object has its own:
  - ***attributes*** – characteristics and properties of the object e.g., model number, colour, registration of a car or colour, weight of a duck, age, breed of a dog
  - ***behaviour*** – what the object can do (methods) e.g., a duck can *fly*, you can *withdraw* or *deposit* into an account, a dog can *bark*
- Each object encapsulates some state (the currently assigned *values* for its attributes) and the state of one object is independent of another
- Each object has its own object identity
- The methods on the object define the object's ***interface***

Object Oriented Design

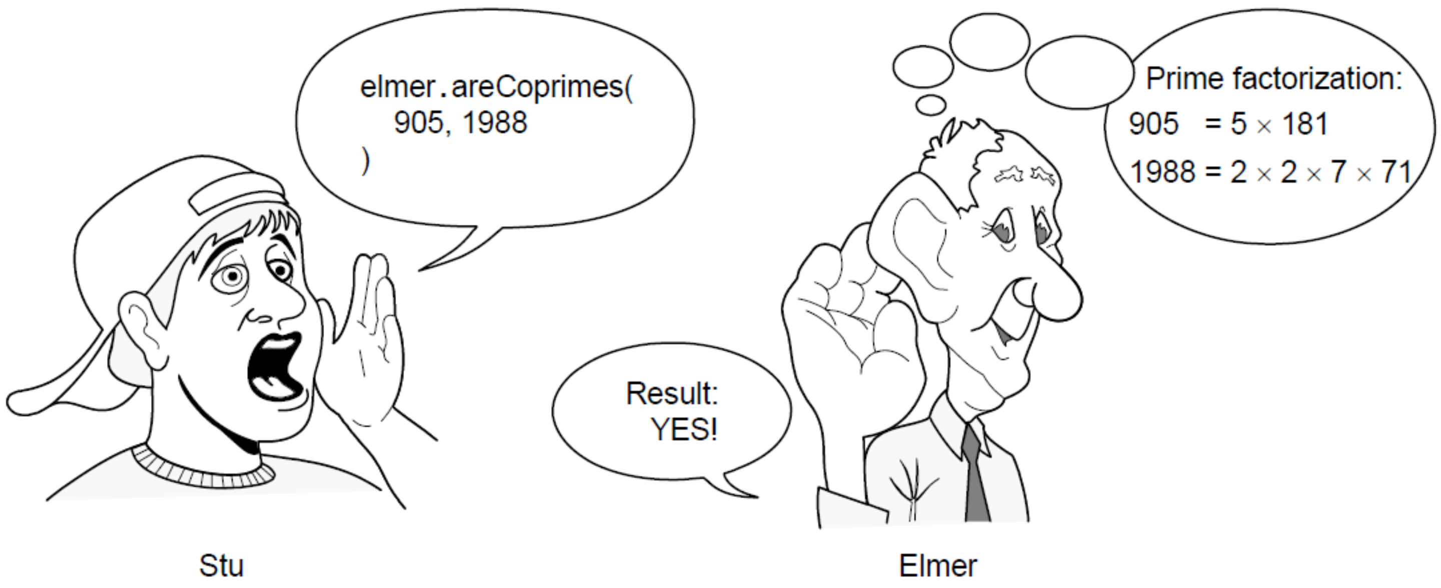




- An object's *interface* is the set of the object's methods that can be invoked on the object
- The interface is the fundamental means of communication between objects

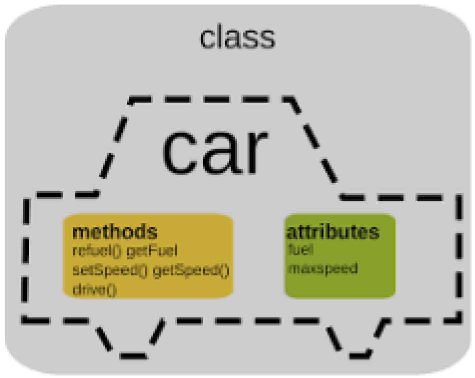
Object Oriented Design

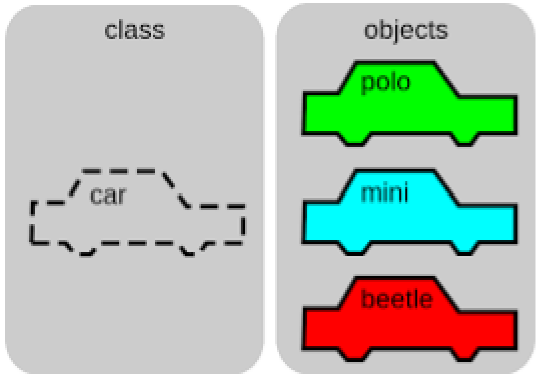
- Objects communicate with each other through sending ***messages*** i.e. invoking methods
  - when an object *A* calls a method on an object *B* we say, “*A* sends a message to *B*.”
  - objects play a *client* and *server* role and could be located in the same memory space or on different computers
  - here, object A is the client and object B is the server



Object Oriented Design

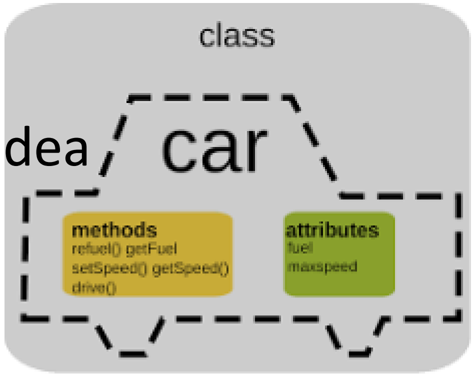
- A ***class*** is a *blue-print* to describe many objects that share the same semantics, properties and behaviour
- A class defines the attributes and methods (behaviour) of the object
- A class is sometimes referred to as an ***object's type***.
- An object is ***instantiated*** from a class (an ***instance*** of a class)
- An object has state but a class doesn't
- Two object instances from the same class share the same properties and behaviours, but have their own object identity and are independent of each other





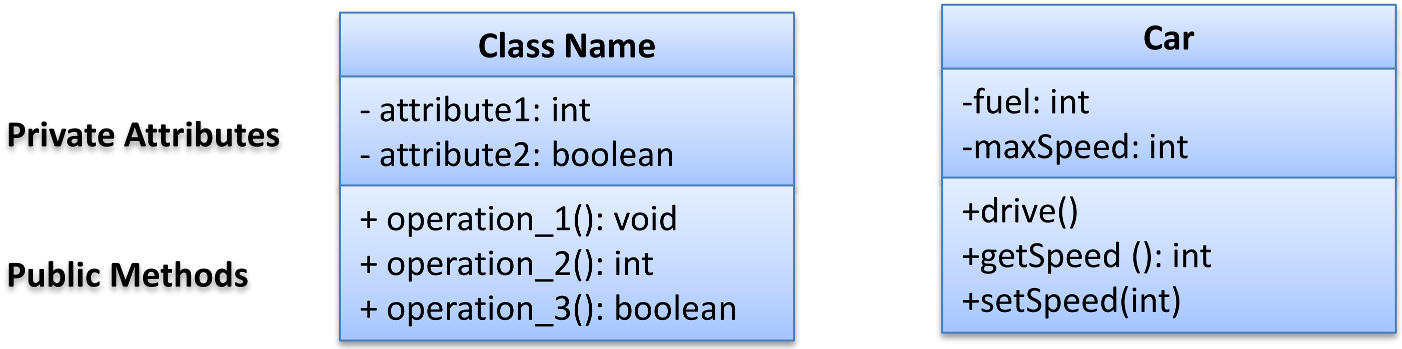
Abstraction

- ***Abstraction*** is at the heart of OO design and key to defining a class
- 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…) ?
    - ***we don't*** create a class for each brand,
  - instead, ***abstract***;
    - focus on the common essential qualities of the idea
    - focus on the current application context
    - and write ***one*** class called Car.
  - What if a specific brand had a special property or behaviour? Later on….***inheritance***



Encapsulation

- **Encapsulation:**
  - A fundamental property of object orientation
  - Means ***hiding*** the object state so that it can only be observed or modified through the object's methods
- An object's attributes represent its individual characteristics or properties, so access to the object's data must be restricted
- How do we achieve encapsulation?
  - Define your blue-print or class with attributes and methods
  - Restrict access to the inner workings of that class by hiding the attributes so that they're only accessible from inside the class scope.
  - Enforced in different ways in different languages (e.g., Java applies stricter enforcement than Python)



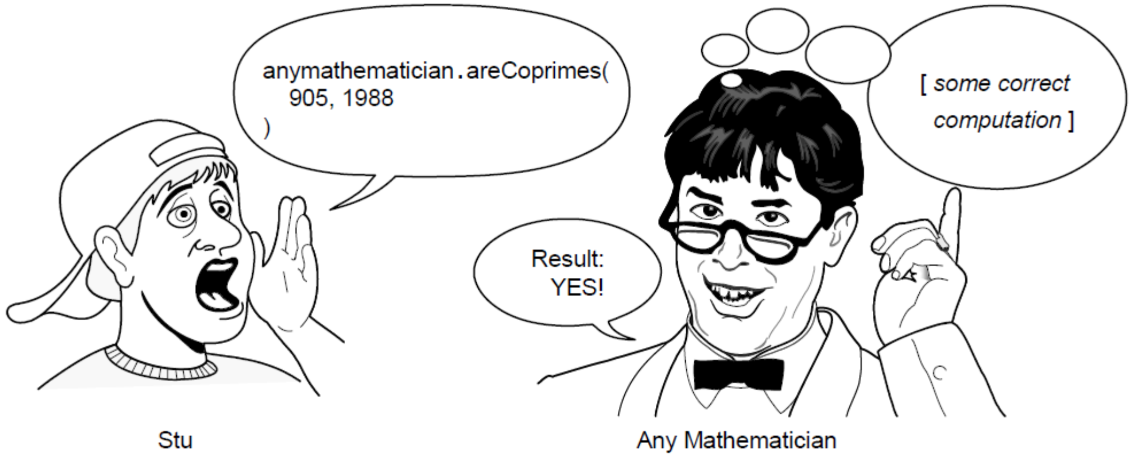
UML notation for a Class

Encapsulation

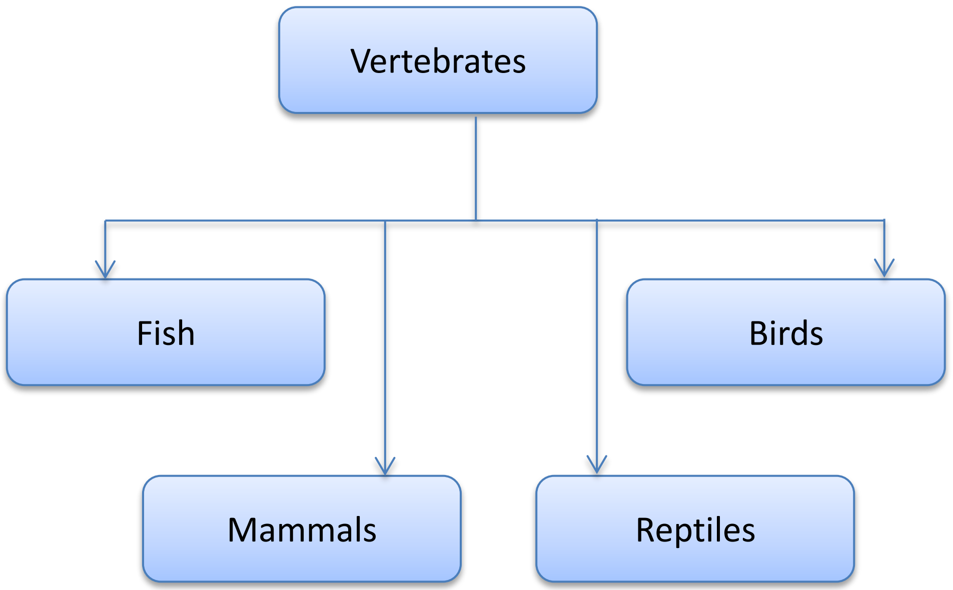
- Why do we need encapsulation?
  - An object’s attributes represent its individual characteristics or properties, so access to the object’s data must be restricted
  - Exposing the object through only its interface ensures that the current object attributes cannot be directly accessed
    - e.g., *balance* can only be changed through *deposit* and *withdraw* methods and not directly
  - Restricted access also means that you only have to worry about a single class
    - *e.g., if you have a bank account object, and you want to change how the balance is stored. Then, the other parts of the application don’t have to worry about this change, Why? Because the balance attribute of the account object is already hidden; can’t be accessed directly.*
  - So, encapsulation *abstracts* away the implementation, reduces dependencies between different parts of the application and ensures that a *change* will not cause a rippling effect on the system

Encapsulation

- OO encourages black-box approach of focusing on an interface through encapsulation
  - we are only interested in *what* an object does,
  - not *how* it does it. The “how” part is considered in implementation
  - encapsulation *abstracts* away the implementation and ensures that a *change* will not cause a rippling effect on the system
- So, we separate a class design into three parts:
  - ***Public interface:*** The services provided by the object
  - ***Contracts:*** The terms and conditions of use
  - ***Implementation:*** Details of how the object conducts its business


anymathematician.areCoprimes(905, 1988)
[some correct computation]
Result: YES!
Stu
Any Mathematician

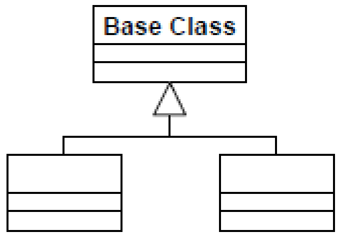
Taxonomy – Class Hierarchy

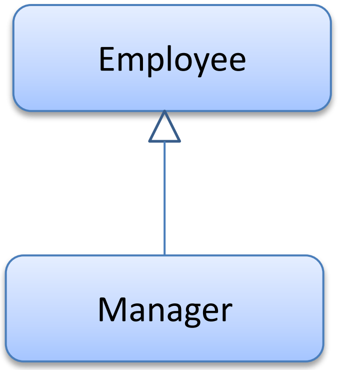


Taxonomy – Class Hierarchy

Class Relationships – Inheritance (1)

- Abstraction says "create one generic class that has the common properties and behaviour of the objects"
- Inheritance says
  - "create a new class (sub-class) , that inherits common properties and behaviour from a base class (super-class) (sub-class *inherits/ extends/ is-derived from* a parent class
    - e.g., a dog **is-a** mammal
  - new class *overrides* parts with specialised behaviour and *extends* parent class with property and behaviour specific to the inherited class of objects
- Inheritance defines a "is-a" type of relationship





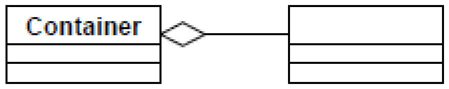
Parent class - Employee
class employee defines *employeeId, name, salary*

Child class – Manager
inherits from parent and defines its own attributes e.g., *bonus*

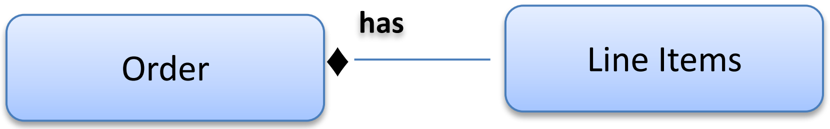
- Multiple inheritance allows you to inherit from more than one super class
  - Supported by languages such as C++, Python
  - Java, C# does not support multiple inheritance

Class Relationships – Composition (2)

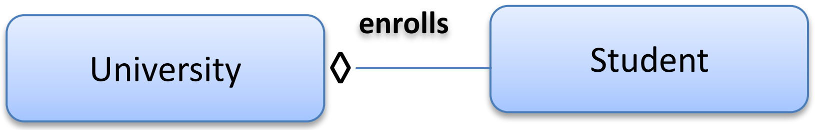
- A special type of relationship between two classes is a ***"has-a"*** relationship between two classes where one class "contains" another class (a class references another class)
  - e.g., a course-offering *has* students



- This relationship can be further refined as:
  - *Composition* relationship (filled diamond symbol ♦ in UML diagrams): The contained item is an integral part of the containing item, such as a leg in a desk, or engine in a car



  - *Aggregation* relationship (hollow diamond symbol ◊): The contained item is an element of a collection but it can also exist on its own, such as a lecturer in a university or a student at a university



How to create a domain model ?

- Domain modelling done from several sources:
  - Studying the problem statement and using our knowledge of how system-to-be is supposed to behave (from requirements analysis, e.g., use cases and sequence diagrams)
  - Knowledge base of software designs and developer’s past experience with software design
- Use sources to identify:
  - Conceptual classes
  - Key attributes of these concepts
  - Key responsibilities of these concepts
  - Determine the collaborations between these concepts

Domain Modelling Techniques (1)

- Noun and Verb Phrase Identification
  - Analyze textual description of the domain to identify **noun** phrases
  - Caveats: Textual descriptions in natural languages are ambiguous (different nouns can refer to the same thing and the same noun can mean multiple things

Consider this text about an ATM machine:

A customer arrives at an ATM machine to withdraw money. The customer enters the card into the ATM machine. Customer enters the PIN. The ATM verifies whether the customer's card number and PIN are correct. Customer withdraws money from the account. The ATM machine records and updates the transaction.

Candidate conceptual classes: ATM, Customer, Account, Card

Object Design using CRC cards

- CRC stands for:
  - ***Class*** : Represents a collection of similar objects
  - ***Responsibility*** : Something that the class *knows* or *does*
  - ***Collaborator*** : Another class that a class must interact with to fulfil its responsibilities
- CRC techniques help people to break away from the procedural mode of thought
- Written in 4 by 6 index cards, an individual CRC card use to represent a domain object
- Featured prominently as a design technique in XP programming

Student	
Enrols in a Seminar *Knows* Name *Knows* Address *Knows* Phone Number	Seminar

Techniques for Domain Modelling (2)

- Domain modelling based on ***object oriented design*** approaches envisions a software application as a set or community of domain objects interacting (collaborating) to fulfil system requirements

  - Each *object* embodies one or more roles, a *role* being defined by a set of related *responsibilities*. Roles, i.e., objects, *collaborate* to carry out their responsibilities.

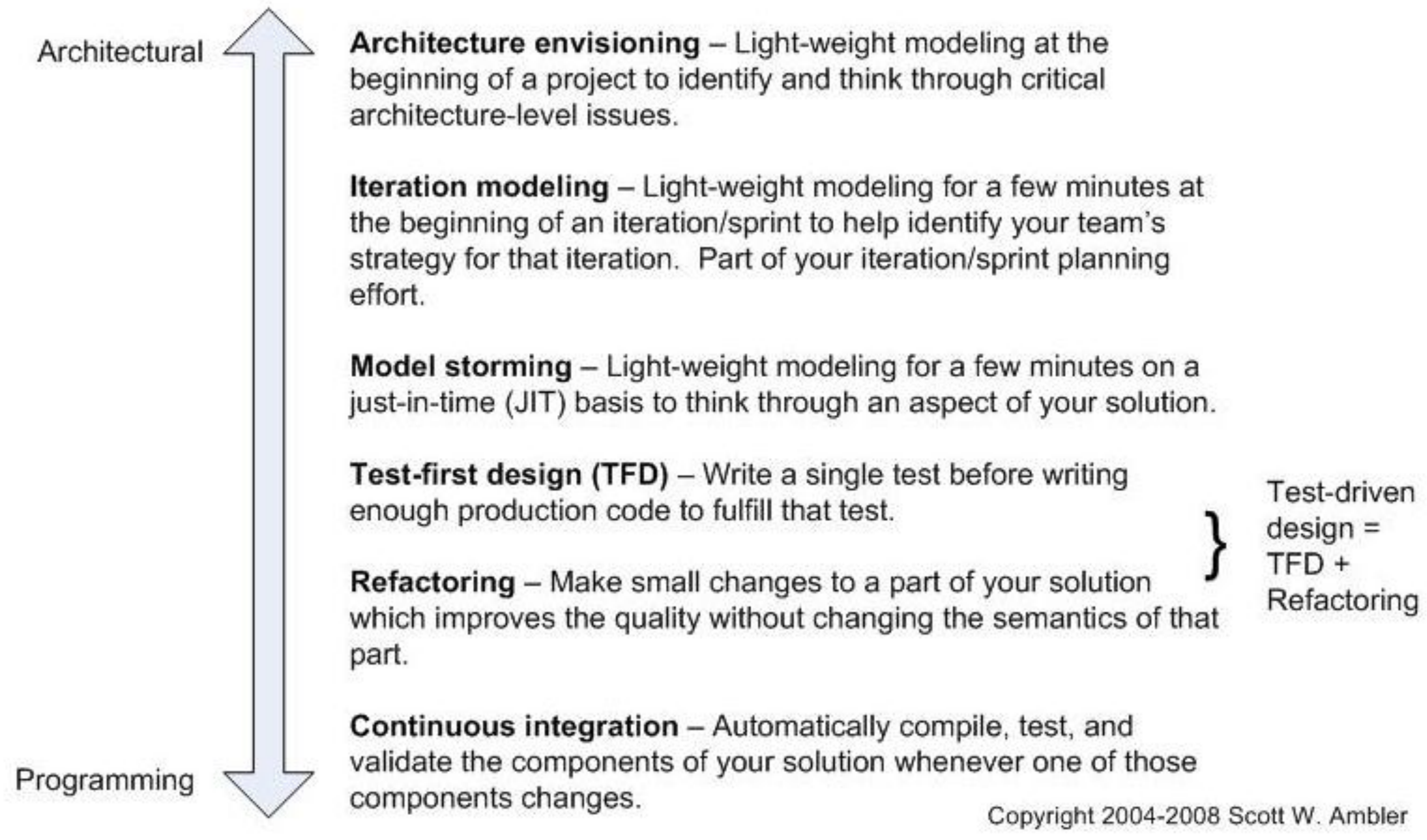
Techniques for Domain Modelling (2)

- Use CRC modelling techniques to:
  - Identify concept “classes and objects”.
  - Identify key attributes of these concepts.
  - Identify concept responsibilities and attributes.
  - Determine associations and generalizations between concepts

- Evolve CRC domain models into UML class diagrams where:
  - Concepts are represented as classes
  - Relationships are represented as associations
  - 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),

Agile Design Practices

To be effective at Agile Design, there is a range of agile design practices from high-level architectural practices to low-level programming practices



Architecture envisioning – Light-weight modeling at the beginning of a project to identify and think through critical architecture-level issues.

Iteration modeling – Light-weight modeling for a few minutes at the beginning of an iteration/sprint to help identify your team's strategy for that iteration. Part of your iteration/sprint planning effort.

Model storming – Light-weight modeling for a few minutes on a just-in-time (JIT) basis to think through an aspect of your solution.

Test-first design (TFD) – Write a single test before writing enough production code to fulfill that test.

Refactoring – Make small changes to a part of your solution which improves the quality without changing the semantics of that part.

Continuous integration – Automatically compile, test, and validate the components of your solution whenever one of those components changes.