Module 2: Object-Oriented Modelling

4 major design principles of abstraction, encapsulation, decomposition, and generalization.

Creating Models in Design

* Object-oriented approach helps make the design process easier.
* Description of concepts as objects. It can be understood by both user and developer

O-O design concepts consist of:

* Conceptual design
  + O-O analysis to identify key objects in the problem and break it down to smaller pieces.
* Technical design
  + O-O design to further refine the details of the objects (incl. attributes & behaviours)
* Goal during SD is to construct and refine models of all the objects.
  + Entity Objects: initial focus during the design is placed in the problem space
  + Control Objects: receive events and co-ordinate actions as the process moves to the solution space
  + Boundary Objects: Connect outside services to your system.

Models should always be critiqued to ensure the original problem is addressed and qualities like reusability, flexibility, maintainability are met.

SM are often expressed in a visual notation, called Unified Modelling Language (UML)

O-O Modelling has different kinds of models for different types of software issues.

Evolution of Programming Languages

Solve problems more efferently with a different paradigm

|  |  |  |  |
| --- | --- | --- | --- |
| Programming Language | Time Period | Solution | Unresolved issues |
| Cobal & Fortran | 1960 | Imperative paradigm  Global data & Subroutines  Subroutines would access vars in global Data | Global data vars changes. Would affect those subroutines |
| Algo 68 Pascal | Early 1970s | Idea of scopes and local vars,  subroutines could have its own vars | Software was quickly growing, having one file to maintain was hard. |
| C Modula-2 | Mid-1970s | New languages, provided a means to organize programs into separate files | Many data types can be created, though one type cannot be declared an extension of another. |
| OOP (Java, C++, C#, etc) | 1980s to present | O-O design seeks to   * Abstract data type easier to write * Structure a system around abstract date types called classes * Inheritance | Predominant programming paradigm now |

For Design Principles

Abstraction:

* Breaks a concept down into a simplified description
* Abstraction should follow the **rule of least astonishment**. Component of a system should behave in a way the user will expect it to behave with no surprises.
* Essential characteristics of an abstraction can be understood in 2 ways:
  + Basic attributes
  + Basic behaviours

Encapsulation:

* Second major design principle. Involves a concept that allows something to be contained in a capsule.
* 3 ideas behind encapsulation:
  + Bundle attribute values and behaviours that manipulate those values, into self-contained object.
  + Expose certain **data** and **functions** of that objects, which can be accessed from other objects, usually through an **interface**
  + Ability to restrict access to certain data and functions to only within the object.

Integrity and Security

* Data integrity and the security of sensitive information is linked to encapsulation.
* Prevents queries from accessing sensitive data.

Black Box

* Encapsulation is also tied to **black box** thinking.
* A class is like a black box. What happens in the box doesn’t matter, as long it accepts inputs and provides outputs.
* Encapsulation achieves an **abstraction barrier** through the **black box**

Decomposition

* 3rd major design principles.
* Consists of taking a whole thing and dividing it into different parts.
* General Rule is to look at the different responsibilities of a whole and evaluate how the whole can be separated into parts that each have specific responsibility

The Nature of Parts

A whole might have a fixed or dynamic number of a certain type of part.

Generalization

* Final of the 4 major design principles.
* Helps reduce redundancy when solving problems.
* Achieves generalization by classes through **inheritance**.
* Take repeated common or shared characteristic between two or more classes and factor them out into another class.
* A parent class is referred to as a **superclass**
* Child class is called a **subclass**.
* Superclass forms a generalization, and its subclasses are more specialized.
* Methods and inheritance allow developers to reuse code, resulting in less code and repetition.
  + D.R.Y – Don’t Repeat Yourself

Design Structure in Java and UML Class Diagrams

Design process consists of both conceptual design and the technical design.

* Conceptual Design
  + Incl prototyping & simulating higher-level designs.
  + Can be visualized through **Class, Responsibility, Collaborator (CRC) cards**
* Technical Design
  + One technique used for technical design is the **UML class diagram** or **class diagram** for short
  + They provide more detail then CRC cards and simple transition to code.

This lesson will look at how the design principles of **abstraction**, **encapsulation**, **decomposition** and **generalization** work with **Java** and **UML class diagrams**.

**AB|EN|DE|GE**

Abstraction

* Abstraction can be applied using UML.
* CRC cards capture components. Which they’re eventually refined into functions or classes, or collections of other components.

CRC Example

Table

Description automatically generated

Class diagram

Text

Description automatically generated

Every class in a class diagram is represented as a box like above.

|  |
| --- |
| Class Name |
| Properties |
| Operations |

* Class name: Same as Java class
* Properties: Java’s member variables.
  + This **section defines** the **attributes** of the **abstraction** by using a standard template for variable name and variable type.
  + **e.g.** **<var name> : <var type>**
* Operations: Java’s Methods
  + Defines behaviour of the abstraction.
  + Using standard template for the operation name, parameter list, return type
  + **e.g. <name>(parameter list) : <return type>**

You will notice that class diagrams distinguish a difference between responsibilities that become properties and responsibilities that become operations, Whereas CRC cards list them together.

Distinguish this ambiguity makes class diagrams easier to translate into code.

* Class name turns into class in Java
* Properties turn into member variables
* Operations become methods.

Encapsulation

The design principle of ENCAP involves 3 ideas:

1. Data & functions that manipulate data are bundled into a self-contained object.
2. Data and functions of the object can be exposed or made accessible from other objects.
3. Data & functions of the object can be restricted to only within the object.

UML Class Diagram Encapsulation is expressed as:

Example:

Table

Description automatically generated with medium confidence

* The class diagram itself bundles data and functions in a self-contained object.
* Access and Restriction (two aspects of visibility) can be represented using symbols - & +
* +: public access modifier
* -: private access modifier

In the example above:

* Public methods can be used to manipulate the students GPA.
* Prevents student GPA attribute from being directly manipulated.
* Encapsulation in UML class diagrams helps you determine the gate to controlling data, by using only public methods to access data attributes of the class.
* For every piece of essential data, the use of public methods to access private data creates protection from unexpected direct change of that data. Preserving data integrity.

2 different kinds of methods used to preserve data integrity:

* Getter methods: To retrieve data.
  + Often **retrieves** **private data**
* Setter Methods: To change data.
  + Often **set** a **private attribute** in a safe way.

Decomposition

* Takes whole thing and divides into parts.
* Takes random parts and makes it whole.

3 different types or relationships – define the interaction between the whole & the parts

* Association
* Aggregation
* Composition

Association

* Loose relationship between 2 objects.
* Not dependent on each other.
  + If one object is destroyed the other one can continue to exist

e.g.

Relationship could be a person & hotel

Person can interact with hotel and not own one

Hotel can interact with many people

* Association represented in UML diagrams as

Person

0..\*

0..\*

Hotel

* 0..\* -> Shows that the person object is associated 0 or more times with the hotel object

Aggregation

* Has a relationship where a whole has parts that belong to it.
* Parts may be shared among wholes in this relationship.
* Typically weak, though they can exist independently.

**Aggregation** is represented the same way as **association** in the UML diagram.

Composition

* Dependent on decomposition relationships.
* Strong as a relationship
* A whole cannot exist without its parts.
  + House (whole), rooms (parts) cannot exist without the house.
* You can access the parts through the whole.

e.g. Composition can be represented with UML class diagrams as below:

House

1..\*

Room

Generalization

* Takes common/ shared characteristics between classes and puts it in a super class which are then inherited in a subclass.
* Generalization & Inheritance can be represented in UML using solid-lined arow

Diagram

Description automatically generatedA picture containing diagram

Description automatically generatedLion is a subclass of Animal.

Superclass attributes and behaviour do not need to be rewritten in the subclass.

Superclass is the generalized classes Subclasses are the specialized classes

The # symbolizes **protected** attributes.

Protected attributes in Java can only be accessed by:

* The encapsulated class itself
* All subclasses
* All classes within the same **package**

**Classes are data types**

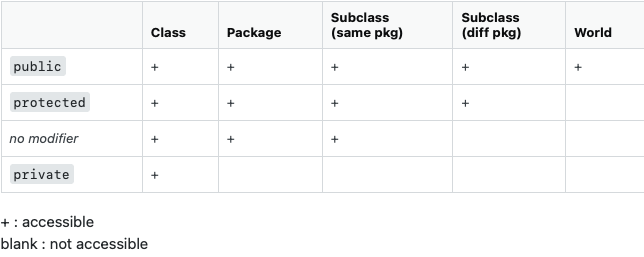
**Java package groups related classes**

**Package is a container that groups related types together**

Text

Description automatically generated with medium confidenceText

Description automatically generatedSince Animal class is a generalization, it should not create an instance of itself. The keyword **abstract** indicates that the class cannot be instantiated.



Inheritance is declared using the **extends** keyword

Classes can have **implicit constructors** or **explicit constructors**

Text

Description automatically generatedText

Description automatically generated

The subclass uses the keyword **super** when its inheriting from the superclass.

The subclass can override methods in the **super** class

Types of Inheritance

Java can support several different types of inheritance.

In Java, only single implementation inheritance is allowed.

* Superclass can have multiple subclasses.
* Subclass can only inherit from a single superclass.

Subclasses can also be a superclass to another class

C++ supports **multiple inheritance** (Subclass can have two or more superclasses)

Java addresses the restriction of **single implementation inheritance** by offering **interface inheritance**

Java interface -> Abstract class that’s used to group related methods with empty bodies.

In Java, keyword **interface** used to indicate that one is being defined. The letter “I” is sometimes placed before an actual name to indicate an interface.

Text

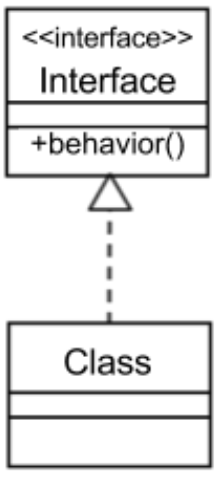
Description automatically generatedThe interface doesn’t encapsulate any attributes of the superclass – Attributes are not behaviours.

To use an interface – You need to use the **implements** keyword.

Text, letter

Description automatically generated

Interfaces represented using UML

**Abstract class** – Cannot be instantiated.

Interfaces are a means of achieving **polymorphism**.

* 2 classes have the same behaviour, but the implementation of said behaviour can be different.

Text

Description automatically generated

Interfaces can inherit from other interfaces – They should not be extended if you’re trying to create a large interface.

Example of Inheritance

Text

Description automatically generated with medium confidence

DATA AMBIGUITY

**Interfaces Advantage** -> MII

Subclass inherits from 2 or more superclasses that have the same attributes or behaviours with the same method signature - not possible to distinguish between them.

Java cannot tell which one or referenced – So doesn’t allow multiple inheritance to prevent data ambiguity

In Java, class can implement as many interfaces as desired.

* Interfaces are only contracts, so overlapping method signatures are not a problem

Abbreviations

O-O: Object-Oriented

SD: Software Design

SM: Software Models

CRC: Class, Responsibility, Collaborator Cards

UML: Unified Modelling Language

ENCAP: Encapsulation

MII: Multiple implementation inheritance