# CS3342 Software Design - Review

## 1. Software Development Process

**Software Process** = a series of predictable steps (shown in **roadmap**) that help us to create a timely, high-quality software product.

**Process** defines **tasks and activities** within a schedule, and **roles and responsibilities** of the team members (who is responsible for what).

**Software Process Model** = an abstract representation of a process. It guides the development with a set of key activities.

All models have **phases**, each phase has 3 components:

- Set of **activities** - work to be done in the phase.
- Set of **deliverables** - documents, code, data, as a result of the activities.
- **Quality control** measures - what to use to assess the quality of the deliverables.

### Linear Process Models

#### Waterfall Model

5 phases: Requirements Definition → System and Software Design → Implementation and Unit Testing → Integration and System Testing → Operation and Maintenance.

Advantages: Easy, structured, provide a template to separate tasks in different phases.

Disadvantages:

- Sequential, does not reflect reality.
- No prototype, little user feedback.
    - Relies on the requirements being correct.
    - Feedback only in the last phase. Cannot adapt to changes.
    - Problems in the specification phase can only be found in coding/testing phase.
    - Take a long time before the first release.

Pure waterfall model: No overlap between phases.

When to use the waterfall model:

- Simple project.
- Limited amount of time.
- Well-understood requirements.
- Well-understood technology.

### Incremental Process Models

Goal: provide basic functionality early, then add more functionality in increments.

Pre-requisite: **Requirements are well defined**.

The software is completed **in small increments**.

### Incremental Model

**Iterative** (like prototype) and **controlled** (like waterfall).

(Requirements 1 → Design 1 → Implementation 1 → Testing 1) → (Requirements 2 → Design 2 → Implementation 2 → Testing 2) → … → Deployment and Maintenance

- 1st increment: Core product.
- later increments: Add more features.
- each increment follows the waterfall model.

When to use the incremental model:

- **Software can be broken down into increments**.

### RAD (Rapid Application Development)

Goal: Shorten the development time.

Communication → Planning → Parallel Modeling + Construction → Deployment

When to use RAD:

- Time frame is short.
- System can be modularized.
- Strong and skillful team members.

## Evolutionary Process Models

Pre-requisite: **Core requirements are well understood, additional requirements are changing fast**.

Specification, development and validation activities are carried out **concurrently** with **rapid feedback**.

Advantages:

- Do not require **full knowledge of requirements**.
- **Iterative**, divide project into **builds**.
- **Allows feedback** and deliver working software early.
- Provide steady and visible progress.

Disadvantages:

- Estimating time and cost is difficult.
- **Risk management** is crucial.

### Prototyping

Show the user a **prototype** of the software and help refine the requirements.

Oral requirements ↔ Code prototype → Prototype testing → Documentation (Design & Requirements) → Deployment → Maintenance

Advantages:

- Facilitate **identification of requirements**.
- Prototypes can be delivered quickly.

Disadvantages:

- Prototypes lack **performance and reliability** of the final product, but the customer may expect the quality of the prototype.

When to use prototyping:

- When the requirements are vague (high-level requirements, with no input/output/process details).
- When the developer is unsure of the design.

### Spiral Model

**Iterative** (like prototype) and **controlled** (like waterfall).

- Each **loop** in the spiral represents a phase.
- **No fixed phases** makes it **flexible**.
- **Risk-driven** - emphasis on risk analysis and management.

4 general activities in each loop:

- **Determine objectives, alternatives, and constraints**.
- **Evaluate alternatives, identify and resolve risks** - Risk analysis + Prototyping.
- **Develop and verify deliverables** - Coding + Testing.
- **Plan the next iteration**.

(Review → Risk Analysis → Prototyping → Engineering → Planning)

Advantages: Can be combined with other models; risk management provides **early warning**.

Disadvantages: **Complex** and requires more management.

When to use the spiral model:

- Very large, critical projects.
- Technical skills must be evaluated at each stage.

### Concurrent Engineering

**Divide and conquer** - divide the project into smaller parts and work on them concurrently. **Initial planning** and **periodic integration** are crucial.

Plan → Divide → Concurrent Work → Integrate

## Misc

### CBSE (Component-Based Software Engineering)

**Reuse** existing components to build new software.

Component principles:

- Components are **independent**.
- Component **implementation** is **hidden**.
- Communication through **well-defined interfaces**.
- Components on the same **platform** and reduce development costs/effort.

**Components** provide a service regardless in which system they are used.

Component characteristics:

- **Standardised** - interfaces, meta-data, documentation, composition and deployment.

- **Independent** - do not interfere with each other.
- **Composable** - can be assembled to form a new system.
- **Deployable** - self-contained and can be deployed independently.
- **Documented** - documentation is available.

Two steps in CBSE:

- **Domain engineering** - identify and develop (or acquire) reusable components.
- **Software engineering** - assemble components to build a new system.

4 activities in CBSE:

Component Qualification → Component Adaptation → Component Composition → Component Update

### Unified Process

**Phases**: Inception → Elaboration → Construction → Transition

**Iterations**: Each phase is divided into iterations.

**Workflow**: a sequence of activities that produce a result. Each workflow takes place in multiple iterations.

### Agile Development

Oral requirements ↔ Test cases → Code → Test → Deployment → back to Oral requirements

**Test-driven development** - write tests before writing code.

**Code is minimal** - only write necessary code to pass the test.

## Exercise

(a) In your own words, please explain what is “Software Engineering”.

> Software engineering is the study and application of engineering to design, develop, and maintain software systems. It involves requirements analysis, design, coding, testing, and maintenance of software systems.

(b) In your own words, explain what is “Software Development Process” (also known as Software Development Life Cycle or Software Process).

> Software Process is a series of predictable steps that help us to create a timely, high-quality software product. It defines tasks and activities within a schedule, and roles and responsibilities of the team members.

(c) In your own words, explain what is “Functional Requirement”, you also need to provide an example in reference to your own group project.

> Functional requirements are the requirements that describe what the software should do. They are the features and functions that the software should provide. They are usually presented in use-case diagrams. For example, in the account module of our group project, the software should allow residents to activate account, log in and retrieve password.

(d) In your own words, explain what is “Component Based Software Engineering” and what are the benefits of CBSE?

> CBSE is a software engineering approach that focuses on the reuse of existing components to build new software. The benefits of CBSE include reduced development costs and effort, and improved software quality and reliability.

(e) Explain why is Software Reuse important in Software Engineering?

> Software reuse is important in software engineering because it can reduce development costs and effort, improve software quality and reliability, and increase productivity and efficiency. It can also help to reduce time-to-market and improve customer satisfaction.

(f) What are the advantages and disadvantages of the Software Prototyping?

Advantages:

- Facilitate identification of requirements.
- Prototypes can be delivered quickly.

Disadvantages:

- Clients may assume the final product will have the same quality as the prototype, such as performance and reliability.
- Prototypes may not be suitable for all projects.

(g) What are the advantages and disadvantages of the Waterfall Model?

Advantages:

- Simple and structured.
- Provide a template so that methods in different phases can be separated.

Disadvantages:

- Sequential, does not reflect reality.
- The first release takes a long time.
- Little user feedback and unable to adapt to changes.
- Risk of problems in the specification phase can only be found in later phases.

# 2. Software Requirements

**Functional requirements** - what the software should do.

- Primary requirements: Core functions.
- Secondary requirements: Administrative, authentication, tracking, to support the primary requirements.

**Non-functional requirements** - how the software should do it.

- Achievements: Performance, scalability, environmental impact
- Safety: Security, reliability, recoverability
- User-friendly: Usability, accessibility

## Use-Case Diagram

**Requirements vs. Design**:

- Requirements: What the system should do.
- Design: How the system should do it.

**Use-case diagram** displays the relationship between actors and use-cases. **Use-case specification** provides detailed information about the use-case.

Elements of a use-case diagram:

- **Actor** - external objects that produce or consume information.
  - Example: human workers, sensors, machines, other systems.
  - Counter-example: database, file system, printer.
- **Use-case**
- **Association** - relationship between actors and use-cases.
- **System boundary** - box around the use-case diagram.

4 types of use-case relationships:

- **Association** - actor uses the use-case. (solid line)
- **Include** - `A <<Include>> B` means A calls B. (dashed line, arrow)
- **Extend** - `B <<Extend>> A` means A can call B if a condition is met. (dashed line, arrow)
  - **Extension point** - lists all extensions for A.
- **Generalization** - X generalizes X1 = X1 inherits from X. (X1 → X, solid line, hollow triangle)

## Exercise - Use-Case Diagram

(a) Study the following scenario. Draw a complete use case diagram of a Ticket Vending Machine (TVM) system for MetroTrain.

A Customer arrives at the train station ticket vending machine:

1. She has 3 options/use-cases allow her to: buy One-way ticket, buy Weekly-pass or buy Monthly-pass.
2. In each of these three options, IF any error occurs, THEN the system must be able to handle using ExceptionHandling, there are following different error exceptions (hint: use-case inheritance)
   a. TimeOut (i.e. the customer took too long to complete the transaction)
   b. TransactionAbort (i.e. the customer choose to cancel without completing the transaction)
   c. OutOfStock (i.e. the vending machine runs out of Tickets or Passes)
   d. OtherErrors (i.e. this is to handle any other errors not covered above)
3. She can choose multiple items. After all the selections of above are completed, she may proceed to the CheckOut function/use-case, which (1) will Calculate the total amount, and then (2) proceed to the Payment screen, where she will be given two options: (hint: use-case inheritance)
   a. Pay by Cash, the inserted cash note will be validated by a CashNoteValidationSystem, or
   b. Pay by Credit Card, the inserted credit card will be processed by a CardPaymentSystem
4. Only IF the payment is successfully completed, THEN it will issue the tickets.
5. The customer can now continue her journey with ticket(s) purchased.

Whenever possible, your use case diagram MUST use `<<Extend>>` or `<<Include>>` as well as inheritance techniques to provide a good use case diagram

(See attached image for the use-case diagram)

(b) Based on the same case study described above, complete the following table to describe the Checkout use case under typical course of events (assuming Customer pay by Cash) AND alternative course of events (assuming customer pay by Credit Card). The situation involves the Customer actor, as well as external payment processing systems such as CardPaymentGateway and CashNoteValidator Systems.

- Use Case Name: Checkout
- Actor(s): Customer, CashNoteValidatorSystem, CardPaymentGatewaySystem
- Description: This use case describes the process of a customer completing the checkout procedure for the tickets selected. On completion, tickets will be issued if the payment is successfully processed.
- Reference ID: METRO-TVM-1.0
- Typical course of events:
    - Step 1 [Actor Action]: This use case is initialized when a customer proceeds to checkout and pay for tickets.
    - Step 2 [System Response]: The system calculates the total amount of the tickets selected.
    - Step 3 [Actor Action]: The customer chooses to pay by cash.
    - Step 4 [System Response]: The system validates the cash note using the CashNoteValidatorSystem.
    - Step 5 [System Response]: The system issues the tickets to the customer.
    - Step 6 [Actor Action]: The customer collects tickets.
- Alternative course of events:
    - Step 3a: The customer chooses to pay by credit card.
    - Step 4a (following Step 3a): The system processes the credit card payment using the CardPaymentGatewaySystem.
    - Step 4b: [Extension point: if the customer cancels the transaction, invoke the use case TransactionAbort]
    - Step 4c: [Extension point: if the payment takes too long, invoke the use case TimeOut]
- Precondition: Checkout can only be made after at least one ticket is selected to purchase.
- Postcondition: The completed transactions will be recorded.

## Use-Case Specification

Elements of a use-case specification:

- Use case name
- Actors
- Description
- Reference
- **Typical course of events** - expected flow of events.
    - Divided into **actor action** and **system response**.
- **Alternate courses** - handled cases that are not in the typical course.
    - Format: `[Extension point + condition] response`.
    - Example: `Step 3: [Extension point: if the credit card requires 3D secure] system prompts the user to enter the verification code`.
- **Preconditions** - conditions that must be true before the use-case starts.

- **Postconditions** - outcome conditions.
    - **Minimal guarantees** - promises that hold even if the use-case fails. Example: The system will not charge the user unless the payment is successful.
    - **Success guarantees** - promises that hold if the use-case is successful. Example: The system will send a confirmation email to the user.

## Exercise - Use-Case Specification

In order to further define functional requirements for the Customer and Checkout scenario for the system, the following defines the requirement specifications for the Checkout use case.

After the Customer selected the goods and ready for Checkout (use case), which (1) it will Calculate Total Amount, and then (2) proceed to the Payment, where Customer will be given two options for Payment: a. Pay by Cash, the system will accept Cash. or b. Pay by Gift Card, the system will deduct the amount from the shopping gift card.

The system will verify and check only IF the payment is successfully completed, then a Receipt is Issued  to the customer, and the customer can take the receipt and the goods purchased. IF unsuccessful, an error  will be displayed to Customer.
Complete the following use case specification table for the Checkout use case under typical course of events (assuming customer pay by Cash) AND alternative course of events (assuming customer pay by Gift Card). This situation involves the Customer actor.

- Use Case Name: Checkout
- Actor(s): Customer
- Description: This use case describes the process of a customer completing the checkout for the goods selected. On successful completion, a receipt will be issued.
- Reference ID: SHOP-1.1
- Typical Course of Events:
    - Step 1 [Actor Action]: This use case is initialized when a customer proceeds to pay for goods selected.
    - Step 2 [System Response]: The system calculates the total amount of the goods selected.
    - Step 3 [Actor Action]: The customer chooses to pay by cash.
    - Step 4 [System Response]: The system accepts the cash payment.
    - Step 5 [System Response]: The system issues a receipt to the customer.
    - Step 6 [Actor Action]: The customer take the receipt and the goods purchased.
- Alternative Course of Events:
    - Step 3: The customer chooses to pay by gift card.
    - Step 4 (following the alternative Step 3): The system deducts the amount from the shopping gift card.
    - Step 4: [Extension point: if the payment is unsuccessful] The system displays an error to the customer.
- Preconditions: The customer has selected the goods and is ready for checkout.
- Postconditions: The completed transactions will be recorded.

# 3. Object Oriented Analysis

# OO Concepts

- **Object**: an entity that has state and operations on the state.
- **Object class**: a template for creating objects.
- **Class** is a set of related objects. Object is an **instance** of a class.

Advantages of OO:

- **Maintainability**: Easy to maintain and modify.
- **Reusability**: Reuse existing classes.

Details:

- **State**: Attributes of an object. Every object has exactly one state at any given time.
- **Method**: **Operations** (change state) and **queries** (return state) of an object.
- **Message passing**: Objects communicate by calling methods on each other.

**Inheritance**: is-kind-of relationship.

- X1 is a subclass of X, if X1 has all the attributes and methods of X, and possibly more.
- **X1 is the subclass**, **X is the superclass**.
- **X1 inherits from X**, **X generalizes X1**.
- X1 is kind of X.
- UML Notation: X1 → X (hollow triangle).
- Subclass inherits all the attributes, operations and relationships of the superclass.
- Subclass may add new attributes, operations and relationships. Subclass may override the operations of the superclass.

Advantages of inheritance:

- An **abstraction mechanism** to classify entities.
- Supports **reuse** both at the design and implementation levels.
- Provides a mechanism to extend the functionality of a class.
- Inheritance diagram provides **organizational knowledge about domains and systems**.

Disadvantages of inheritance:

- Classes are **not self-contained**. They are only meaningful in the context of the superclass.
- Inheritance diagram of analysis, design and implementation may be **different**. They need to be separately maintained.

**Encapsulation** / **Information Hiding**: Hiding the internal state of an object and requiring all interactions to be performed through an object's methods. Enhances **maintainability**.

With information hiding,

- The change in A does not affect B.
- Different classes communicate through **message passing**.

**Abstraction**: Abstract classes / interfaces. UML Notation: Italicized class name for abstract classes, `<<interface>>` for interfaces.

- Abstract class:
  - A special class that cannot be instantiated.
  - Can have default implementations. Subclasses can override the methods.
  - Does not support multiple inheritance.
- Interface:

- Not a class, just an entity.
- Contains only abstract methods. Subclasses must implement the methods.
- Supports multiple inheritance.

**Implementation**: Inheritance of interfaces. UML Notation: X1 → X (dashed line, hollow triangle).

**Use Object Line**: Y → X (dashed line, arrow) if Y uses X.

**Polymorphism**: The ability of an object to take on many forms.

- Allows a subclass to provide a specific implementation of a method that is already provided by its superclass.
- Realized by **method overriding**.

**Composition**: A has B. UML Notation: B → A (solid line, black diamond).

**Aggregation**: A has B. UML Notation: B → A (solid line, white diamond).

**Association**: A uses B during its lifetime. UML Notation: B —— A.

Difference between composition and aggregation:

- Composition means B is included as part of A, and is thus mandatory. A is responsible for the creation and destruction of B.
- Aggregation means B is only included during A's lifetime, and is thus optional.

**Multiplicity**: Number of objects that participate in a relationship. (`1`, `*`, `0..1`, `1..*`, `m..n`).

## Exercise - OO Concepts

```
class Button {}
class SquareButton extends Button {}
class RectangleButton extends Button {}
class Display {}
class Calculator {Display display; List<Button> buttons;}
class CalculatorApp {Calculator calc;}
```

(a) In your own words, explain what is “Inheritance” in Object-Oriented Programming Design.

> Inheritance allows a subclass to inherit and reuse the attributes and methods of a superclass. It requires the subclass to be a kind of the superclass, and therefore provides a mechanism to classify entities and extend the functionality of a class.

(b) Provide an example from the above Calculator design, that Inheritance could be used.

> In the Calculator design, `SquareButton` and `RectangleButton` can inherit from `Button`. Both `SquareButton` and `RectangleButton` are kinds of `Button`, and they can inherit the attributes and methods of `Button`.

(c) In your own words, explain what is “Polymorphism” in Object-Oriented Programming Design.

> Polymorphism is the ability of an object to take on many forms. It allows a subclass to provide a specific implementation of a method that is already provided by its superclass.

(d) Provide an example from the above Calculator design, that Polymorphism could be used.

> In the Calculator design, `SquareButton` and `RectangleButton` can override the `onClick` method of `Button`. Both `SquareButton` and `RectangleButton` can provide a specific implementation of the `onClick` method.

(e) There exist three main types of Class Linkages, namely Composition, Aggregation and Association. Which is the most appropriate relationship between class Calculator and Button (Figure 2), justify your choice.

> Calculator has a composition relationship with Button. This is because Button is part of Calculator, and is mandatory. Button cannot exist without Calculator.

(f) What is the most appropriate Multiplicity to describe the Class Association between Calculator and Button(Figure 2), justify your answer.

> One calculator can have multiple buttons, and one button can only belong to one calculator. Therefore, the most appropriate multiplicity is `1..*` on the Button side and `1` on the Calculator side.

(g) There is only one Display attached to the Calculator (Figure 2), which Design Pattern could be used?

> Singleton pattern can be used to ensure that there is only one instance of Display attached to the Calculator.

## Sequence Diagram

**Dynamic models** - specify the behavior of objects by describing the **sequentially control** of their methods.

Elements of a sequence diagram:

- **Objects or actors**
- **Life lines** - vertical dashed lines representing the object's existence.
- **Messages** - horizontal arrows representing the method calls between objects.
- **Execution occurrence** - vertical rectangles representing the time the object is executing a method.
- **Conditions** - guard conditions for messages.

4 types of messages:

- Synchronous message: The sender is blocked until the receiver returns.
- Reply message: The return message (if any). Dash line with an arrow.
- Asynchronous message: The sender does not wait for the receiver to return.
- Create message: Create a new object. Dash line with an arrow pointing to the object symbol.

**Conditions**: an expression that must be true for the message to be sent.

A guarded transition is a transition that is only taken if the guard condition is true. (Notation: `[condition] method()`).

**Black hole**: message to black hole is called **lost message**. message from black hole is called **found message**.

**InteractionUse** (ref): A reference to another sequence diagram.

**CombinedFragments**: Flow control.

- **alt**: if-elseif-else. Only one of the fragments is executed.

- **opt**: if. No else.
- **break**:
  - when break condition is true, `break` fragment is executed, and the rest is skipped.
  - when break condition is false, `break` fragment is skipped, and the rest is executed.
- **loop**:
  - loop as long as the condition is true.
  - `loop(min, max)` or `loop(min)` specifies the number of iterations.

Structure of a sequence diagram:

- **Fork diagram** (centralized control): one object controls the flow.
  - appropriate when operations can change order, or new operations can be added.
- **Stair diagram** (distributed control): each object controls its own flow.
  - appropriate when strong connection exists and operations will only be executed in a certain order.

# 4. Design Principles and Patterns

## Design Principles

- SRP (Single Responsibility Principle)
- OCP (Open-Closed Principle)
- LSP (Liskov Substitution Principle)
- ISP (Interface Segregation Principle)
- DIP (Dependency Inversion Principle)
- LoD (Law of Demeter)

### Cohesion and Coupling

Goal:

- **High cohesion** - elements within a module are strongly related. (SRP, ISP)
- **Low coupling** - modules are independent of each other. (OCP, DIP, LoD)

**Cohesion**: the degree to which an element contributes to a single purpose.

- **Functional cohesion**: elements perform a single well-defined task.
- **Sequential cohesion**: elements are related in a sequence. The output of one element is the input of the next.
- **Communicational cohesion**: elements operate on the same data.

**Coupling**: the degree of interdependence between modules. There is a coupling between X and Y if changes in X require changes in Y.

- **Data coupling**: modules communicate through parameters (primitive data types, or objects as a whole). (get/set an object reference)
- **Stamp coupling**: modules communicate through a composite data structure (but not all fields are used). (get/update state of an object)
- **Control coupling**: modules communicate through control flags.
- **Common coupling**: modules share global variables.
- **Content coupling**: modules share internal working mechanisms. (e.g. copying code from one module to another)

## OCP (Open-Closed Principle)

A class should be **open for extension**, but **closed for modification**.

- Open for extension: new behavior can be added.
- Closed for modification: (1) attributes are private, (2) new behavior can be added without changing the existing code.
- Implemented by **abstraction** and **polymorphism**.

```
abstract class Shape {
    abstract void draw();
}
class Circle extends Shape {
    void draw() { /* ... */ }
}
class Rectangle extends Shape {
    void draw() { /* ... */ }
}
class GraphicEditor {
    void drawShape(Shape s) { s.draw(); }
}
```

In this example, new `Shape` classes can be added without changing the `GraphicEditor` class.

## DIP (Dependency Inversion Principle)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

- High-level modules: modules that contain the application logic.
- Low-level modules: modules that contain the implementation details.

```
interface Appliance {
    void turnOn();
    void turnOff();
    boolean isOn();
}
class Light implements Appliance {
    void turnOn() { /* ... */ }
    void turnOff() { /* ... */ }
    boolean isOn() { /* ... */ }
}
class Spotlight implements Appliance {
    void turnOn() { /* ... */ }
    void turnOff() { /* ... */ }
    boolean isOn() { /* ... */ }
}
class Switch {
    private Appliance appliance;
    void setAppliance(Appliance a) { appliance = a; }
    void push() {
        if (appliance.isOn()) appliance.turnOff();
        else appliance.turnOn();
    }
}
```

In this example, `Light` and `Spotlight` are low-level modules, and `Switch` is a high-level module. Both depend on the `Appliance` interface, so `Switch` does not directly depend on specific implementations.

### LoD (Law of Demeter)

A module should only have limited knowledge of other modules; only access its immediate friends.

Degree of separation: at most 1. (e.g. `a.b.c` has a degree of separation of 2)

```
class NotificationService {
    ArrayList<Observer> observers;
    void notifyObservers() {
        for (Observer o : observers) o.update();
    }
    void addObserver(Observer o) { observers.add(o); }
    void removeObserver(Observer o) { observers.remove(o); }
}
class RSSFeed {
    NotificationService ns;
    void update() { ns.notifyObservers(); }
}
```

In this example, `RSSFeed` should not instruct `NotificationService` on how to notify observers. Instead, `RSSFeed` should only call `update()`.

### SRP (Single Responsibility Principle)

A class should have only one reason to change.

```
class GraphicEditor {
    void drawShape(Shape s) { s.draw(); }
}
class MathUtils {
    double getArea(Shape s) { return s.getArea(); }
}
```

In this example, `GraphicEditor` is responsible for user interaction, and `MathUtils` is responsible for background calculations. They have different responsibilities.

### ISP (Interface Segregation Principle)

A client should not be forced to implement an interface that it does not use.

```
interface IReadWrite {
    void read();
    void write(String s);
}
interface IReadGrade {
    void read();
    void grade(int score);
}
class Coursework implements IReadWrite, IReadGrade {
```

```
    void read() { /* ... */ }
    void write(String s) { /* ... */ }
    void grade(int score) { /* ... */ }
}
class Student {
    void read(IReadWrite r) { r.read(); }
    void write(IReadWrite w, String s) { w.write(s); }
}
class Tutor {
    void read(IReadGrade r) { r.read(); }
    void grade(IReadGrade g, int score) { g.grade(score); }
}
```

In this example, `Coursework` implements both `IReadWrite` and `IReadGrade`, but `Student` and `Tutor` only use the methods they need.

### LSP (Liskov Substitution Principle)

Subtypes must be substitutable for their base types.

```
class Rectangle {
    int width, height;
    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
    int getArea() { return width * height; }
}
class Square extends Rectangle {
    void setWidth(int w) { width = w; height = w; }
    void setHeight(int h) { width = h; height = h; }
}
```

This is a counter-example of LSP. A `Square` is not a subtype of `Rectangle` because it does not behave like a `Rectangle`.

Despite the implementation enforces the `Square` to be a square, the client may not use it in a semantically correct way. For example:

```
public void test(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assert r.getArea() == 20;
}
test(new Square());
```

Additionally, subclasses should not inherit methods and properties that are not used. For example, a `Bird` class should not inherit `swim` method from `Animal` class. (Mitigation: create interfaces and `Bird extends Animal implements IFly`)

# Design Patterns

## Creational - Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate.

```
abstract class ShapeFactory {
    abstract Shape createShape(double[] args);
    void createAndDraw(double[] args) {
        Shape s = createShape(args);
        s.draw();
    }
}
class CircleFactory extends ShapeFactory {
    Shape createShape(double[] args) { return new Circle(args); }
}
class RectangleFactory extends ShapeFactory {
    Shape createShape(double[] args) { return new Rectangle(args); }
}

// main
ShapeFactory f = new CircleFactory();
f.createAndDraw(args);
```

In the above example,

- `Shape` is the abstract product.
- `Circle` and `Rectangle` are concrete products.
- `ShapeFactory` is the abstract factory. It delegates the creation of the product to the concrete factories, but also provides a common method to create and draw the product.
- `CircleFactory` and `RectangleFactory` are concrete factories. They create the concrete products.

Another example:

```
interface Document {
    void open();
    void close();
    void save(String filename);
}
class HTMLDocument implements Document { /* ... */ }
class PDFDocument implements Document { /* ... */ }
abstract class AbstractApplication {
    abstract Document createDocument(String type);
    void newDocument(String type) {
        Document doc = createDocument(type);
        doc.open();
    }
    void openDocument(String filename) { /* ... */ }
}
class DocumentFactory extends AbstractApplication {
    Document createDocument(String type) {
        if (type.equals("html")) return new HTMLDocument();
        else if (type.equals("pdf")) return new PDFDocument();
```

```
        return null;
    }
}
```

### Creational - Singleton

- Ensure a class has only one instance.
- Provide a **global point of access** to the instance.

```
class Singleton {
    private static Singleton instance; // private static instance
    private Singleton() {} // private constructor
    public static Singleton getInstance() { // public static method
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

### Structural - Facade

Facade serves as an interface to a complex subsystem.

```
class Facade_FrontEnd {
    private Shape circle, rectangle;
    public Facade_FrontEnd() {
        circle = new Circle();
        rectangle = new Rectangle();
    }
    public void drawCircle() { circle.draw(); }
    public void drawRectangle() { rectangle.draw(); }
}
class Facade_Client {
    public static void main(String[] args) {
        Facade_FrontEnd f = new Facade_FrontEnd();
        f.drawCircle();
        f.drawRectangle();
    }
}
```

In the above example,

- `Shape`, `Circle`, and `Rectangle` are the subsystem classes.
- `Facade_FrontEnd` is the facade class, which provides external access to the subsystem classes.
- `Facade_Client` is an external client.

### Behavioral - Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

When to use:

- When a system has two parts with a **one-to-many relationship**.
- When a change in one object requires a change in possibly many other objects.

- When an object should be able to notify others without knowing who they are.

Supports **broadcast communication**.

```
abstract class RSSFeed {
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    void addObserver(Observer o) { observers.add(o); o.update(this); }
    void removeObserver(Observer o) { observers.remove(o); }
    void notifyObservers() {
        for (Observer o : observers) o.update(this);
    }
}
class Blog extends RSSFeed {
    void onDeploy() { notifyObservers(); }
}

interface Observer {
    void update(RSSFeed f);
}
class MobileReader implements Observer {
    void subscribe(RSSFeed f) { f.addObserver(this); }
    void update(RSSFeed f) { /* ... */ }
    void unsubscribe(RSSFeed f) { f.removeObserver(this); }
}
class WebReader implements Observer {
    void update(RSSFeed f) { /* ... */ }
}
```

Note: in the `Observer` class, only `update` method is part of the observer pattern.

## Behavioral - Command

Encapsulate a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of requests.

```
interface Command {
    void execute();
}
class LightOnCommand implements Command {
    private Light light;
    LightOnCommand(Light l) { light = l; }
    void execute() { light.turnOn(); }
}
class LightOffCommand implements Command {
    private Light light;
    LightOffCommand(Light l) { light = l; }
    void execute() { light.turnOff(); }
}
class Switch {
    private ArrayList<Command> history = new ArrayList<Command>();
    public void storeAndExecute(Command c) { history.add(c); c.execute(); }
    public void redo() { history.get(history.size() - 1).execute(); }
}
```

Note: In this case `Switch` has an `ArrayList<Command>`. This is condiered `Switch` aggregates `Command`. (Not composition because `Command` can exist independently of `Switch`.)

Of course a fully functional undo/redo system would require two stacks, one for undo and one for redo.

```
class CommandCenter {
    private ArrayList<Command> undoStack = new ArrayList<Command>();
    private ArrayList<Command> redoStack = new ArrayList<Command>();
    private static CommandCenter instance;
    private CommandCenter() {}
    public CommandCenter getInstance() {
        if (instance == null) instance = new CommandCenter();
        return instance;
    }

    public void execute(Command c, String[] args) {
        c.execute(args);
        undoStack.add(c);
        redoStack.clear();
    }
    public void undo() {
        Command c = undoStack.remove(undoStack.size() - 1);
        c.undo();
        redoStack.add(c);
    }
    public void redo() {
        Command c = redoStack.remove(redoStack.size() - 1);
        c.execute();
        undoStack.add(c);
    }
}
```

## Behavioral - State/Strategy

Allow an object to alter its behavior when its internal state changes.

```
interface State {
    void updateState(Context c);
}
class Acknowledged implements State {
    void updateState(Context c) { c.setState(new Shipped()); }
}
class Shipped implements State {
    void updateState(Context c) { c.setState(new Delivered()); }
}
class Delivered implements State {
    void updateState(Context c) { /* ... */ }
}
```

```
interface SortStrategy {
    void sort(int[] arr);
}
class BubbleSort implements SortStrategy {
    void sort(int[] arr) { /* ... */ }
}
class QuickSort implements SortStrategy {
    void sort(int[] arr) { /* ... */ }
}

// main
SortStrategy s = new BubbleSort();
s.sort(arr);
```

Advantages:

- **Consolidate** the state-specific behavior into one class.
- **Consistent** behavior across states.
- **Allow state changes**

Disadvantages:

- Too many instances of the state classes. (can use singleton pattern)

### When to Use Design Patterns

- State: Allow an object to alter its behavior when its internal state changes.
- Singleton: (1) Ensure a class has only one instance. (2) Save the number of duplicate objects.
- Factory Method: Creation of objects is to be determined at runtime.
- Facade: Provide a simple interface to a complex subsystem and hide the backend complexity.
- Observer: When a system has two parts with a one-to-many relationship. (1) Need to notify all dependents when one object changes state. (2) Need to separate present layer (views) from the data layer (model).
- Command: Encapsulate a request as an object, issue commands without knowing underlying operations, allow for saving requests in a queue.

## Exercise - Design Principles and Patterns

```
public class BusTicket
{
    private String BusZone;
    private boolean isReturnTicket;
    private String Title;
    private String Name;
    private int birthDay;
    private int birthMonth;
    private int birthYear;
    BusTicket(boolean returnTicket, String _title, String _name,
        int _birthDay, int _birthMonth, int _birthYear, String _busZone) {
        isReturnTicket = returnTicket;
        Title = _title;
        Name = _name;
        birthDay = _birthDay;
```

```
        birthMonth = _birthMonth;
        birthYear = _birthYear;
        BusZone = _busZone;
    }
    public int calculateFare () {
    double fare = 0;
        if (BusZone == "CITY") {
            if (isReturnTicket)
                fare = 10;
            else
                fare = 8;
        } else if (BusZone == "INTER_CITY") {
            if (isReturnTicket)
                fare = 20;
            else
                fare = 16;
        } else if (BusZone == "INTER_STATE") {
            if (isReturnTicket)
                fare = 50;
            else
                fare = 30;
        }
        return fare;
    }
    public void changeBusZone(String _busZone) {
        BusZone = _busZone;
    }
}
```

(a) The class constructor BusTicket(...) contains a long list of parameters describing the attributes of a Customer buying the ticket, and it would be difficult to maintain the code and design if would place them in a single class. This is in violation of _____ Design Principle.

> Single Responsibility Principle

(b) The class function calculateFare() is designed to correctly calculate the Bus ticket fares for three different Bus Zones, namely CITY, INTER_CITY and INTER_STATE, and the fare is also calculated according to the Boolean value isReturnTicket. According to Figure 3. BusTicket.java, unfortunately this design do not allow its behaviour to be further extended without modifying the source code of BusTicket.java. In Software Design Principle, this could be described as potential violation of _____ Design Principle, making addition of new Bus Zones a difficult task in the future.

> Open-Closed Principle

(c) Redesign the BusTicket class to mitigate the above design issues. Add `Customer` class to store customer information.

```
class Customer {
    private String Title;
    private String Name;
    private int birthDay;
    private int birthMonth;
    private int birthYear;
```

```
    Customer(String _title, String _name, int _birthDay, int _birthMonth, int
_birthYear) {
        Title = _title;
        Name = _name;
        birthDay = _birthDay;
        birthMonth = _birthMonth;
        birthYear = _birthYear;
    }
    public String getName() { return Name; }
}

abstract class BusZone {
    private boolean isReturnTicket;
    public BusZone(boolean _isReturnTicket) { isReturnTicket = _isReturnTicket;
}
    public abstract double calculateFare();
}

class CityBusZone extends BusZone {
    public CityBusZone(boolean _isReturnTicket) { super(_isReturnTicket); }
    public double calculateFare() { return isReturnTicket ? 10 : 8; }
}

class InterCityBusZone extends BusZone {
    public InterCityBusZone(boolean _isReturnTicket) { super(_isReturnTicket); }
    public double calculateFare() { return isReturnTicket ? 20 : 16; }
}

class InterStateBusZone extends BusZone {
    public InterStateBusZone(boolean _isReturnTicket) { super(_isReturnTicket);
}
    public double calculateFare() { return isReturnTicket ? 50 : 30; }
}

class BusTicket {
    private BusZone busZone;
    private Customer customer;
    BusTicket(BusZone _busZone, Customer _customer) {
        busZone = _busZone;
        customer = _customer;
    }
    public double calculateFare() { return busZone.calculateFare(); }
    public void changeBusZone(BusZone _busZone) {
        busZone = _busZone;
    }
}
```

(a) Please describe the motivation and when to use Singleton Pattern in software design, in addition to your explanation, please draw a simple class diagram to show an example of Singleton Pattern including its essential attributes and operations.

> Motivation: Need to ensure that a class has only one instance. When to use: (1) When a class should have only one instance, and clients should be able to access it. (2) When we need to save the number of duplicate objects, like in State pattern.

```
class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

(b) To facilitate the needs of information dissemination of banks, the City's Metropolitan Banking & Monetary Authority needs to design a new system to allow all news agencies to receive the latest updates about different bank's financial updates such as their latest interest rates. You are required to use the Observer Pattern to create a design and using a complete class diagram to show your solution.

Specifically, the system needs to handle at least three different news agencies namely, NewsPaper, Internet, and TVnews, and your design should ensure its compliance to OCP design principle.

According to the Observer Pattern, the subject of the information update should include the latest information of its Loan interest rate and TermDeposit interest rate of banks. The system should be able to register and unregister concerning news agencies to such a subject for information of a bank, as well as being able to notify all news agencies about the latest updates of the concerning subject registered.

Please show your solution using Observer Pattern in a complete UML class diagram. You are required to use ArrayList to store all the registered observers.

```
abstract class Subject {
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    void addObserver(Observer o) { observers.add(o); }
    void removeObserver(Observer o) { observers.remove(o); }
    void notifyObservers() {
        for (Observer o : observers) o.update(this);
    }
}
class Bank extends Subject {
    private double loanInterestRate;
    private double termDepositInterestRate;
    public double getLoanInterestRate() { return loanInterestRate; }
    public double getTermDepositInterestRate() { return termDepositInterestRate;
}
    public void setLoanInterestRate(double rate) { loanInterestRate = rate;
notifyObservers(); }
    public void setTermDepositInterestRate(double rate) {
termDepositInterestRate = rate; notifyObservers(); }
}
abstract class Observer {
    protected double loanInterestRate;
    protected double termDepositInterestRate;
    abstract void update(Subject s);
}
class NewsPaper extends Observer {
    void update(Subject s) {
```

```
        loanInterestRate = ((Bank)s).getLoanInterestRate();
        termDepositInterestRate = ((Bank)s).getTermDepositInterestRate();
    }
}
class Internet extends Observer {
    void update(Subject s) {
        loanInterestRate = ((Bank)s).getLoanInterestRate();
        termDepositInterestRate = ((Bank)s).getTermDepositInterestRate();
    }
}
class TVnews extends Observer {
    void update(Subject s) {
        loanInterestRate = ((Bank)s).getLoanInterestRate();
        termDepositInterestRate = ((Bank)s).getTermDepositInterestRate();
    }
}
```

(a) The Metro Library needs to implement a new system to keep track of the borrowings of books by their library members. Each member is able to borrow many books from the Library, and the library also needs to record the due date of each book being borrowed by a member.

For recording information about their books, the Library need to record attributes:
- Book_ID: int
- title: String
- author: String
- As well as all operations required accessing these attributes.

For recording information about their members, the Library need to record attributes:
- Member_ID: int
- name: String
- age : int
- As well as all operations required accessing these attributes.

The library needs to record information about the book being borrowed by a specific member, as well as its dueDate (book return date). Given that many library members and many books are in the system, how do you correctly model such a relationship using UML class diagram to design the system?
Please show your full solution using an UML class diagram, including all attributes and operations required. (The class constructor is optional).

```
class Book {
    private int Book_ID;
    private String title;
    private String author;
    Book(int _Book_ID, String _title, String _author) {
        Book_ID = _Book_ID;
        title = _title;
        author = _author;
    }
    public int getBook_ID() { return Book_ID; }
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
}
class Member {
```

```
    private int Member_ID;
    private String name;
    private int age;
    Member(int _Member_ID, String _name, int _age) {
        Member_ID = _Member_ID;
        name = _name;
        age = _age;
    }
    public int getMember_ID() { return Member_ID; }
    public String getName() { return name; }
    public int getAge() { return age; }
}
class BorrowRecord {
    private Book book;
    private Member member;
    private Date dueDate;
    BorrowRecord(Book _book, Member _member, Date _dueDate) {
        book = _book;
        member = _member;
        dueDate = _dueDate;
    }
    public Book getBook() { return book; }
    public Member getMember() { return member; }
    public Date getDueDate() { return dueDate; }
}
```

(b) In addition to (a), the Metro Library needs to implement three different membership classifications (Student, Adult and Senior) to facilitate the different membership fee charges according to their membership types. i.e. $50/year for Student, $200/year for Adult and $20/year for Senior members, the system should be able to facilitate the changes of membership classifications in a long term. Please provide a solution to extend your class diagram given in (a), you can provide a full or only the extended class diagram to show your solution.

```
interface Membership {
    double getFee();
}
class StudentMembership implements Membership {
    public double getFee() { return 50; }
}
class AdultMembership implements Membership {
    public double getFee() { return 200; }
}
class SeniorMembership implements Membership {
    public double getFee() { return 20; }
}
class Member {
    private int Member_ID;
    private String name;
    private int age;
    private Membership membership;
    Member(int _Member_ID, String _name, int _age, Membership _membership) {
        Member_ID = _Member_ID;
        name = _name;
        age = _age;
```

```
        membership = _membership;
    }
    public int getMember_ID() { return Member_ID; }
    public String getName() { return name; }
    public int getAge() { return age; }
    public double getMembershipFee() { return membership.getFee(); }
}
```

(c) Please explain and justify in full details whether your solution given in (b) satisfies the OCP Design Principle?

> The solution satisfies the OCP Design Principle. Reason: The solution uses the State pattern to encapsulate the behavior of different membership classifications. Therefore, the system is open for extension but closed for modification: new membership classifications can be added without changing the existing code. Additionally, use of private attributes ensures all changes are made through public methods. (Additional notes: state pattern allows switching between different membership classifications, and provide consistent behavior across states.)

```
// IN PHONE.JAVA
public interface Phone {
    public void makeCall(int phoneNum);
    public void sendMessage(int phoneNum);
    public void takeHighQualityImage();
    public void accessWebviaWifi();
}
// IN MOTO2100.JAVA
public class Moto2100 implements Phone {
    public void makeCall(int phoneNum) { System.out.println("Calling to " +
phoneNum + " with a Moto2100"); }
    public void sendMessage(int phoneNum) { System.out.println("Sending message
to " + phoneNum + " with a Moto2100"); }
    public void takeHighQualityImage() { System.out.println("This operation is
not supported in this device"); }
    public void accessWebviaWifi() { System.out.println("This operation is not
supported in this device"); }
}
// IN GALAXY.JAVA
public class Galaxy implements Phone {
 public void makeCall(int phoneNum) { System.out.println("Calling to " +
phoneNum + " with a Galaxy"); }
 public void sendMessage(int phoneNum) { System.out.println("Sending message to
" + phoneNum + " with a Galaxy"); }
 public void takeHighQualityImage() { System.out.println("Take highquality image
with the Galaxy technology"); }
 public void accessWebviaWifi() { System.out.println("Access web via Galaxy Wifi
technology"); }
}
```

The class Phone models the general functions of various phone devices. On top of Phone, the classes of Moto2100 and Galaxy implement their specific behavior. Study the java code above and point out the bad design within that violates the ISP design principle. Then you should propose your improvement on the current (undesirable) design. Specifically, your answer should meet the

following two requirements:
(1) Point out and explain why these codes represent a violation of the ISP design principle.

> ISP design principle states that a client should not be forced to implement an interface that it does not use. In this case, the Moto2100 class implements the takeHighQualityImage() and accessWebviaWifi() methods, which are not supported by the Moto2100 device. This violates the ISP design principle because the Moto2100 class is forced to implement methods that it does not use.

(2) Describe your idea of design improvement and sketch a new full class diagram to illustrate your design that complies with the ISP design principle.

> Design improvement: Segregate the Phone interface into smaller interfaces, each representing a specific set of functionalities. This includes PhoneCommunication, PhoneCamera, and PhoneInternet. The Moto2100 and Galaxy classes will then implement only the interfaces that they need.

```
public interface PhoneCommunication {
    public void makeCall(int phoneNum);
    public void sendMessage(int phoneNum);
}
public interface PhoneCamera {
    public void takeHighQualityImage();
}
public interface PhoneInternet {
    public void accessWebviaWifi();
}

public class Moto2100 implements PhoneCommunication {
    public void makeCall(int phoneNum) { System.out.println("Calling to " +
phoneNum + " with a Moto2100"); }
    public void sendMessage(int phoneNum) { System.out.println("Sending message
to " + phoneNum + " with a Moto2100"); }
}

public class Galaxy implements PhoneCommunication, PhoneCamera, PhoneInternet {
    public void makeCall(int phoneNum) { System.out.println("Calling to " +
phoneNum + " with a Galaxy"); }
    public void sendMessage(int phoneNum) { System.out.println("Sending message
to " + phoneNum + " with a Galaxy"); }
    public void takeHighQualityImage() { System.out.println("Take highquality
image with the Galaxy technology"); }
    public void accessWebviaWifi() { System.out.println("Access web via Galaxy
Wifi technology"); }
}
```

## Roles of Variables

| Role | Description |
| --- | --- |
| Constant | Fixed value, cannot be changed |
| Stepper | |

| Role | Description |
|---|---|
| Most-recent holder | From input only (*) |
| Gatherer | Accumulate values |
| Transformation | Computed from other variables |
| One-way flag | True or false |
| Temporary | Contextually short-lived |
| Organizer | Group of variables |

(*) Last value encountered in a loop is a Temporary variable.

## Exercise - Roles of Variables

(from assignment)

```
public class Grades {
    public static void main(String[] args) {
        int student_id[] = { 1031, 1022, 3021, 2023, 2062, 3027, 4022}; //
student_id: Organizer
        int student_sc[] = { 80, 84, 89, 79, 55, 92, 73}; // student_sc:
Organizer
        int fail = 0;
        int total = 0;
        int max_sc = 0;
        double average = 0;
        Boolean neg = false;
        final int student_num = student_id.length; // student_num: Constant
        for (int i = 0; i < student_num; i++) { // i: Stepper
            int sc = student_sc[i]; // sc: Temporary
            int id = student_id[i]; // id: Temporary
            total += sc; // total: Gatherer
            if (sc > max_sc) {
                max_sc = sc; // max_sc: Most-recent holder
            }
            if (sc < 60) {
                fail++; // fail: Stepper
                neg = true; // neg: One-way flag
            }
        }

        average = total / student_num; // average: Transformation
        system.out.println();
        system.out.println("Number of student: " + student_num);
        system.out.println("Average score: " + average);
        system.out.println("Number of fail: " + fail);
        system.out.println("Maximum score: " + max_sc);
    }
}
```

(from sample exam)

```
public class Seller {
    private List<Goods> goods_list = new ArrayList<Goods>(); // goods_list:
Organizer
    private final double discount_rate = 0.8; // discount_rate: Constant
    public void addGoods(int num) { // num: Constant
        Scanner scanner = new Scanner(System.in);
        for (int i = 1; i <= num; i++) { // i: Stepper
            String goods_name = scanner.next(); // goods_name: Most-recent
holder
            Goods goods = new Goods(goods_name); // goods: Temporary
            goods_list.add(goods);
        }
    }
    public double calculateRevenue() {
        double sum_revenue = 0.0;
        for (Goods goods : goods_list) { // goods: Temporary
            sum_revenue = sum_revenue + goods.getRevenue(); // sum_revenue:
Gatherer
        }
        double final_revenue = sum_revenue * discount_rate; // final_revenue:
Transformation
        return final_revenue;
    }
    public double getMaxRevenue() {
        double max_revenue = 0.0;
        double current = 0.0;
        boolean neg_revenue = false;
        final int m = goods_list.size(); // m: Constant
        for (int i = 0; i < m; i ++) { // i: Stepper
            current = goods_list.get(i).getRevenue(); // current: Temporary
            if (current < 0) {
                neg_revenue = true; // neg_revenue: One-way flag
                System.out.println("Error exists.");
            }
            if (current > max_revenue) {
                max_revenue = current; // max_revenue: Most-recent holder
            }
        }
        return max_revenue;
    }
}
public class Goods {
    private String goods_name;
    private int sales;
    private int price;
    public Goods(String name) {this.goods_name = name;}
    public double getRevenue() {return price * sales;}
}
```

Notes:

- In `public void addGoods(int num)`, `num` is a function parameter, which is a constant.
- In `String goods_name = scanner.next();`, `goods_name` is a most-recent holder for the input.

- In `for (Goods goods : goods_list)`, `goods` is not a stepper because it does not iterate through a predictable sequence.

(from midterm)

```
String[] tmp = scanner.nextLine().split(" ");
```

In this case `tmp` is a temporary variable. It is not treated as an organizer because organizer focus on rearranging the data in a structured way, but for `tmp` it is just a temporary storage for the split result.

(from sample midterm)

```
public class AccountController {
    private final List<Account> account_list = new ArrayList<Account>(); //
account_list: Organizer
    public void registerAccount(String name, String address, double balance) {
// name, address, balance: Constant
        Account account = new Account(); // account: Temporary
        account.setName(name);
        account.setAddress(address);
        account.setBalance(balance);
        account_list.add(account);
    }
    public Account searchAccount(String acc_name) {
        for (int i = 0; i < account_list.size(); i++) { // i: Stepper
        if (account_list.get(i).getName() == acc_name)
            return account_list.get(i);
        }
        return null;
    }
    public double calculateBalance_USD() {
        double sum_HKD = 0;
        double sum_USD = 0;
        for (Account acc : account_list) {
            sum_HKD = sum_HKD + acc.getBalance(); // sum_HKD: Gatherer
        }
        sum_USD = sum_HKD / 7.8; // sum_USD: Transformation
        return sum_USD;
    }
}
public class Account {
    private String _name;
    private String _address;
    private double _balance;
    public void setName(String name) { _name = name; }
    public void setAddress(String address) { _address = address; }
    public void setBalance(double balance) { _balance = balance; }
    public String getName() { return _name; }
    public String getAddress() { return _address; }
    public double getBalance() { return _balance; }
}
```

(from sample exam)

```
public class SortPercentage {
    public static void main(String[] args) {
        int intArray[] = new int[]{5,90,35,45,95,3,45,19,62,73}; // intArray:
Organizer
        double intArrayPercent [] = new double[intArray.length];  //
intArrayPercent: Organizer
        double sum=0, max=0, min=0, range=0;
        double percent=100; // percent: Constant

        Sort(intArray);

        for (int i=0 ; i< intArray.length; i++) { // i: Stepper
            double percent_conv = (double)intArray[i] / percent; //
percent_conv: Transformation
            intArrayPercent[i] = percent_conv;
            sum = sum + percent_conv; // sum: Gatherer
        }
        max = intArrayPercent[intArrayPercent.length-1]; // max: Most-recent
holder
        min = intArrayPercent[0]; // min: Most-recent holder
        range = max - min; // range: Transformation

        System.out.println("Size of Array is: \t" + intArrayPercent.length);
        System.out.println("Sum is: \t\t"+ sum);
        System.out.println("Max is: \t\t"+ max);
        System.out.println("Max is: \t\t"+ min);
        System.out.println("Range is: \t\t"+ range);
    }
    private static void Sort(int[] intArray) { // intArray: Organizer
        int n = intArray.length; // n: Constant
        int temp = 0;

        for(int i=0; i < n; i++){
            for(int j=1; j < (n-i); j++){
                if(intArray[j-1] > intArray[j]){
                    temp = intArray[j-1]; // temp: Temporary
                    intArray[j-1] = intArray[j];
                    intArray[j] = temp;
                }
            }
        }
    }
}
```

Note:

- In `private static void Sort(int[] intArray)`, `intArray` is not a constant even though it is a parameter.
    - It is an organizer because it is here to be sorted. In fact, its content is changed in the method.
    - "Organizer is a data structure, which is only used for rearranging its data and object elements after initialization."

(from tutorial)

```
class Growth {
    public static void main(int amount) { // amount: Constant
        float loanAmount, interest; int i;
        int n = 10; // n: Constant
        loanAmount = amount;
        for (i = 1; i <= n; i++) { // i: Stepper
            interest = 0.05 * loanAmount; // interest: Transformation
            loanAmount = loanAmount +interest; // loanAmount: Gatherer
            System.out.println("After " + i + " years, loan is " + loanAmount);
        }
    }
}

class Student {
    int student_id;
    double total_credits;
    Vector course_list;

    Student (int s_id) { // s_id: Constant
        student_id = s_id; // student_id: Constant
        total_credits = 0;
        course_list = new Vector(); // course_list: Organizer
    }

    public void passCourse (Course c) { // c: Constant
        total_credits = total_credits + c.getCredits(); // total_credits:
Gatherer
        course_list.add(c);
    }
}

boolean isThreeOfaKind(const String cards[], int n) { // cards: Constant, n:
Constant
    for (int i=0; i<n-2; i++) { // i: Stepper
        if (cards[i][1] == cards[i+1][1] && cards[i+1][1] == cards[i+2][1])
            return true;
    }
    return false;
}
```

Note: in the last example, `cards[]` is explicitly marked as a constant. Otherwise, we always assume that container variables are organizers.

# 5. Software Ethics

## Code of Ethics in Software Engineering

- **Public interest** - Act consistently with the public interest.
- **Client and employer** - Act in the best interests of the client and employer.
- **Product** - Deliver and maintain the product with the highest standards possible.
- **Judgment** - Maintain integrity and independence of oneself.
- **Management** - Promote an ethical approach in management of subordinates.
- **Profession** - Advance the integrity and reputation of the profession as software engineers.
- **Colleagues** - Be fair to and supportive to colleagues.

- **Self** - Participate in lifelong learning, given the fast pace of technology.

Why is Software Engineering Code of Ethics important to Software Engineers?

- Code of ethics can be **powerful instruments** in the drive for professionalism and in establishing safeguards for society.
- Code of ethics **educate and inspire** professional members to adopt and follow the code.
- Code of ethics **inform the public** about the responsibilities that are important to the profession.
- Code of ethics instruct practitioners about **the standards the society expects** them to meet and the peers expect each other to uphold.
- Code of ethics offer **practical advise about issues** that matters to the profession and their clients.

## Exercise - Code of Ethics in Software Engineering

Case study (from lecture):

> A team of software engineering students try hard to finish their group project that develops a software application to meet the assignment requirement. However, Student X (a member of the same team) contributes virtually nothing to the course project.
> The team hides the case from the course instructor, and collectively claim that Student X contributes equally to the project.
> Course instructor finds out the case but still gives the same grade to Student X as the other team members.

| Code | Student X | Team | Course Instructor |
|---|---|---|---|
| Public interest | - | - | - |
| Client and employer | No | No | No |
| Product | - | - | - |
| Judgment | - | No | No |
| Management | - | - | No |
| Profession | - | - | - |
| Colleagues | No | No | No |
| Self | - | - | - |

Explanation:

- Client and employer: The course instructor is the client. X and the team are not acting in the best interest of the client. Course instructor is against the University's interest.
- Judgment: X does not have to endorse a document. However, the team should have reported the issue and the course instructor should have taken action.
- Management: The course instructor gives preferential treatment to X, which is unfair to the whole class.
- Colleagues: The team is not fair to the other team members who have contributed. Course instructor is not fair to other instructors who follow the rules.

Case study (from tutorial):

> Mandy is a technical writer (who writes documentation in the software development project).
> John is a programmer in the same project.
> Normally, John briefs Mandy in detail. He sketches about the design of his program, and lets Mandy play the software product version that uses his code for a while in front of him (so that John knows whether Mandy gets his concept right). Mandy then writes the design document for John's program based on the briefing and her actual experience on the software product version.
> However, now, John is very busy, and Mandy is going to leave the company. John only drafts a very rough sketch on the whiteboard with many technical terms that Mandy does not comprehend. Mandy wants to produce the document before her departure, but John is too busy to squeeze time out to clarify the terms to Mandy. At the same time, for this particular case, no usable software product version can be available for Mandy to play with. Mandy then uses her experience to guess the design and writes up the design document.
> Luke is the supervisor of both Mandy and John. He oversees the project. Luke finds that the design document seems wrong. He requests John to review the document. Nonetheless, due to his busy schedule, John simply tells Luke that the document is "good enough" without checking the content in detail. Luke feels uncomfortable and considers that this deliverable is risky. He thus schedules another technical writer to review and modify this particular document again.

| **Code** | **Mandy** | **John** | **Luke** |
|---|---|---|---|
| Public interest | - | - | - |
| Client and employer | No | No | Yes |
| Product | No | No | Yes |
| Judgment | No | No | Yes |
| Management | - | - | Yes |
| Profession | No | No | - |
| Colleagues | No | No | Yes |
| Self | - | - | - |

Case study (from sample exam):

> During the current coronavirus pandemic, CityU has decided to run all final examination online, and trust that their students are capable of supervising themselves and respecting the system while carrying out online examination without having any academic misconduct, such as cheating by seeking for solutions from any other available materials including the Internet, Books, and any related Teaching materials.
> A student X studying in CS3342 Software Engineering Design, fully aware of the examination rules and arrangements and its closed-book nature. X is gaining unfair advantage by searching Google and other Internet online resources, and obtained solutions required for the exam questions. Another student Y also studying in CS3342 Software Engineering Design, pre-arranged a study room on campus and teaming up with student X above to working on the examination together, discussion and working towards pre-divided

> examination questions with student X.
> Student Z observed the above situation but did not report to the course instructor.

Based on the ACM Code of Ethics, evaluate the actions of Student X, Student Y, and Student Z, from 8 perspectives: Public interest, Client and employer, Product, Judgment, Management, Profession, Colleagues, Self. (Answer with Yes/No/Irrelevant)

| Code | Student X | Student Y | Student Z |
|---|---|---|---|
| Public interest | - | - | - |
| Client and employer | No | No | No |
| Product | No | No | - |
| Judgment | No | No | No |
| Management | - | - | - |
| Profession | - | - | - |
| Colleagues | No | No | No |
| Self | - | - | - |

Explanation:

- Client and employer: CityU is the client. X, Y, and Z are working against the academic integrity of CityU.
- Product: The product is the coursework (examination). For X and Y, the examination is compromised. Z is not directly involved.
- Judgment: X fail to work with integrity. Y and Z should have reported the misconduct.
- Colleagues: Working together to cheat is unfair to other students.

Case study (from revision):

> In a software development project, Mary is the programmer, and Angela supervises Mary. They are good friends of one another. Sam is the user of the software.
> Mary presents a user interface (UI) of the software to Sam, but Sam considers the UI very bad in design. Because Mary does not want to make changes, she tells Sam that he has to follow the current way to input data.
> Mary reports to Angela that Sam is happy with the current UI.
> Angela is surprised by Mary's report because Angela tried the UI herself, but did not consider the current UI easy to use. She thus contacts Sam directly and finds out that Sam also finds the UI is not usable.
> Angela therefore coaches Mary about professionalism, and suggests Mary to consult Sam again and improve the UI accordingly.

| Code | Mary | Angela |
|---|---|---|
| Public interest | - | - |
| Client and employer | No | Yes |
| Product | No | Yes |

| Code | Mary | Angela |
|---|---|---|
| Judgment | No | Yes |
| Management | - | Yes |
| Profession | No | - |
| Colleagues | No | Yes |
| Self | - | - |

## Five P's

- **Purpose** - The objective for doing something.
    - Feel comfortable with the purpose.
    - Purpose should be able to hold up to scrutiny.
- **Pride** - Take pride in the solution you have developed.
    - No false pride or self-doubt.
- **Patience** - Think about all implications of the solution.
- **Persistence** - Do not be easily discouraged by failure.
    - Keep trying to find the best solution.
- **Perspective** - Think about the big picture.
    - Think about the impact of the solution on the world.

Relationship between the Five P's: Purpose, Pride, Patience, Persistence → Perspective

(Perspective is the inner guidance gained from the other P's that allow us to see the issue more clearly.)