Object Oriented Analysis and Design using Java

UE21CS352B

Prof K V N M Ramesh

Department of Computer Science and Engineering

PES UNIVERSITY
CELEBRATING 50 YEARS

UE21CS352B:
Object Oriented Analysis and Design using Java

GRASP: Designing objects with responsibilities

Prof K V N M Ramesh

Department of Computer Science and Engineering

Agenda

- Introduction
- Responsibility
- Types of Responsibility
- References
- GRASP Design Principles

Introduction

- Consists of **guidelines for assigning responsibility to classes and objects in object-oriented design**.
- Stands for **General Responsibility Assignment Software Patterns**
- First published by **Craig Larman in 1977**
- A set of "**nine fundamental principles in object design and responsibility**"

- As a tool for software developers, provides a **means to solve organizational problems and offers a common way to speak about abstract concepts.**

Responsibility

- **A contract / obligation that a class / module / component** must accomplish
- Responsibility can be:
  - Accomplished by a single object
  - A group of object collaboratively accomplish a responsibility.

- Identify the objects and responsibilities from the problem domain, and also identify how objects interact with each other. Helps us in deciding which responsibility should be assigned to which object/class.
- **Define blueprint for those objects** – i.e., class with methods implementing those responsibilities

Responsibility contd..

- Design of software objects and larger scale components is in terms of **responsibilities, roles, and collaborations.** - RDD

RDD – **An abstraction of what they do**

UML - **A contract or obligation of a classifier**

- Related to the obligations or behavior of an object in terms of its role
- **Assigned to classes of objects during object design**.

Example: I may declare that "a Sale is responsible for creating SalesLineItems"

- The **translation of responsibilities into classes and methods** is influenced by **the granularity of the responsibility**

Types of Responsibility

- **Doing:**

  Doing something itself, such as creating an object or doing a calculation

  Initiating action in other objects

  Controlling and coordinating activities in other objects

- **Knowing:**

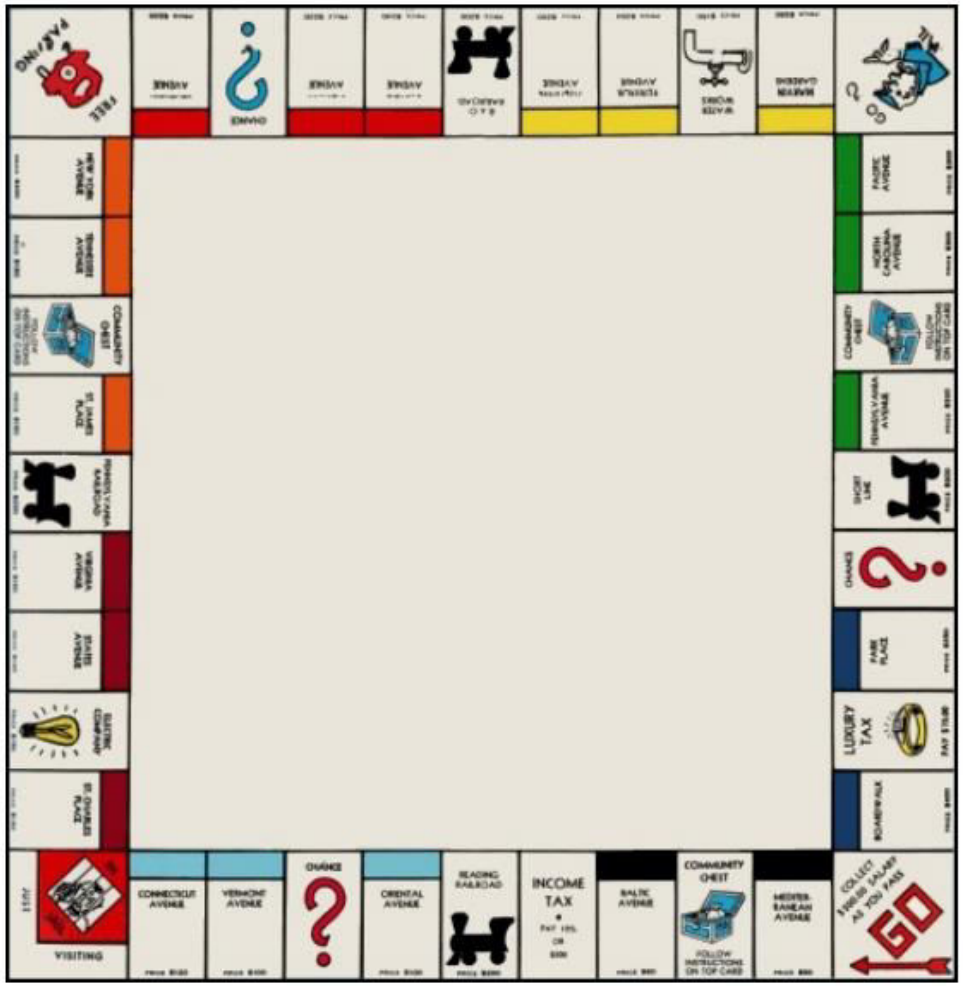
  Knowing about private encapsulated data

  Knowing about related objects

  Knowing about things it can derive or calculate

Object Oriented Analysis and Design using Java

Monopoly game



GRASP Design principles

- Different patterns/principles used are
  - **Creator**
  - **Information expert**
  - **Low coupling**
  - **Controller**
  - **High cohesion**
  - **Polymorphism**
  - **Indirection**
  - **Protected variations**
  - **Pure fabrication**

Design principle (Larman: "pattern") – An attempt at a methodical way to do object design

Creator

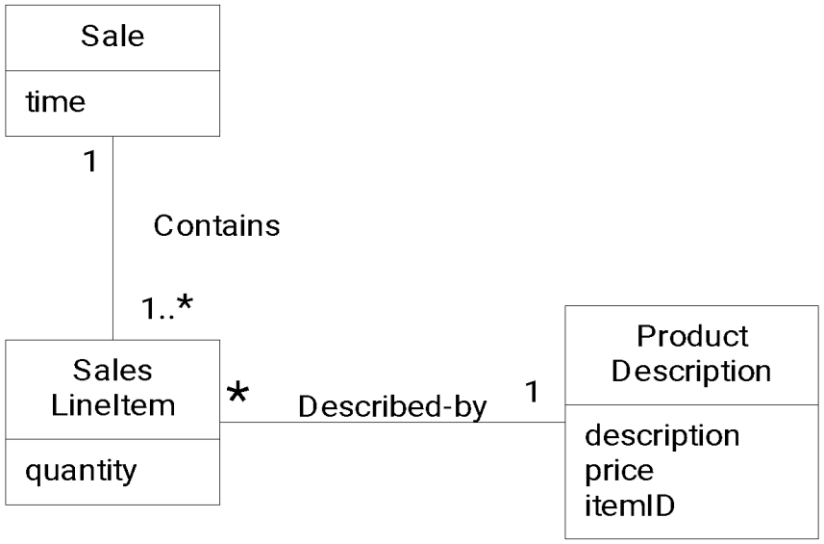
- Who creates an Object? Or who should create a new instance of some class?
- "Container" object creates "contained" objects.
- Decide who can be creator based on the object's association and their interaction

Name:	**Creator**
Problem:	Who creates an A?
Solution: (this can be viewed as advice)	Assign class B the responsibility to create an instance of class A if one of these is true (the more the better): ■ B "contains" or compositely aggregates A. ■ B records A. ■ B closely uses A. ■ B has the initializing data for A.

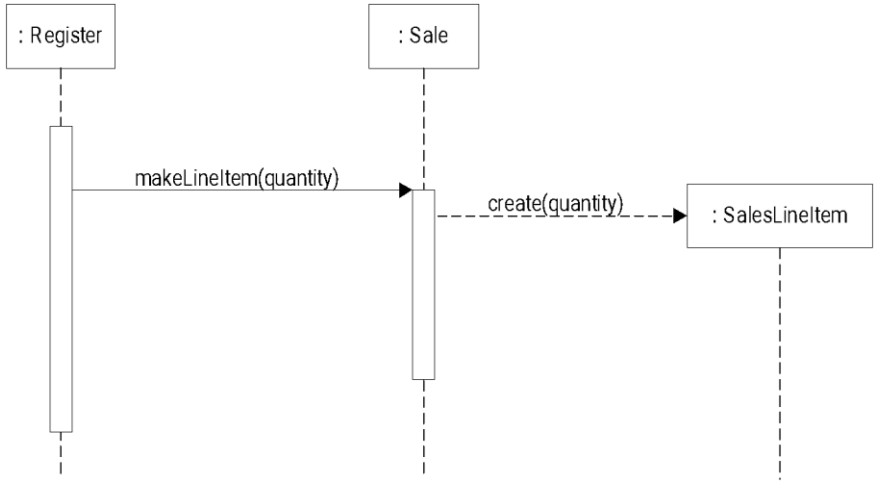
- **Example:** Who creates the square object in monopoly game?

Creator

- Start with domain model.

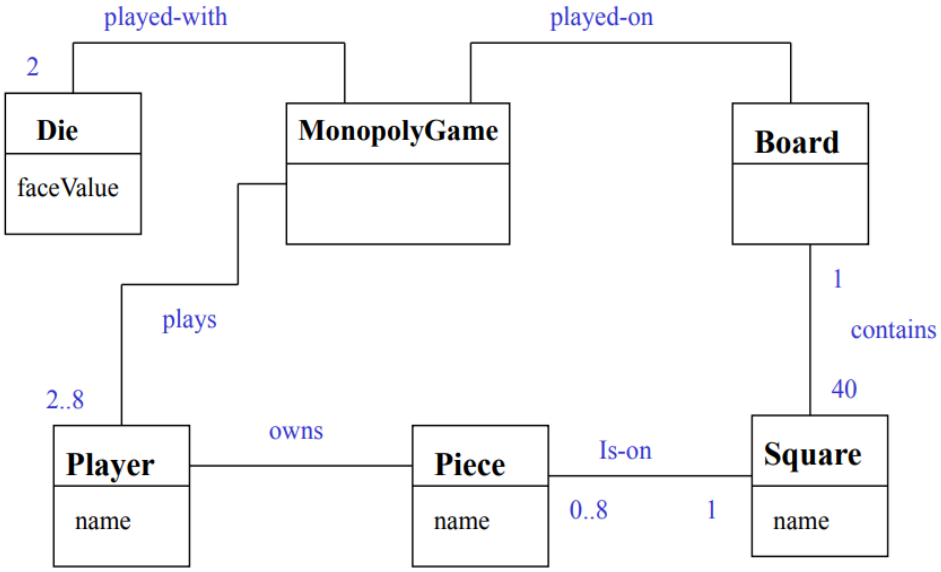


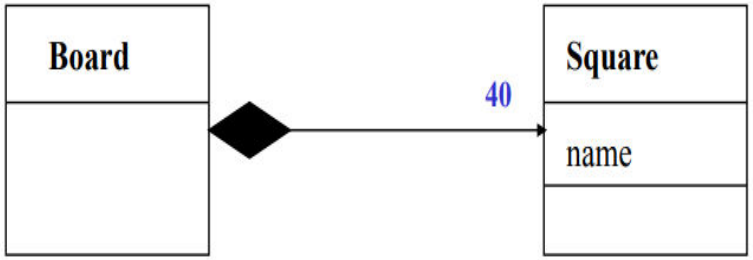
- Since a Sale contains SalesLineItem objects, it should be responsible according to the Creator pattern



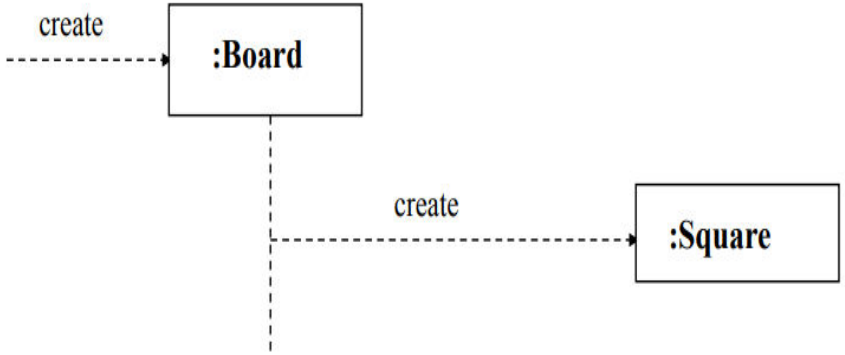
- Note: There is a related design pattern called **Factory for more complex creation situations**

Creator – Monopoly example





Applying the Creator pattern in Static Model



Dynamic Model – illustrates Creator Pattern

Information Expert

- "**objects do things related to the information they have**"
- Information necessary may be spread across several classes => **objects interact via messages**
- **Definition:**

  Name: Information Expert

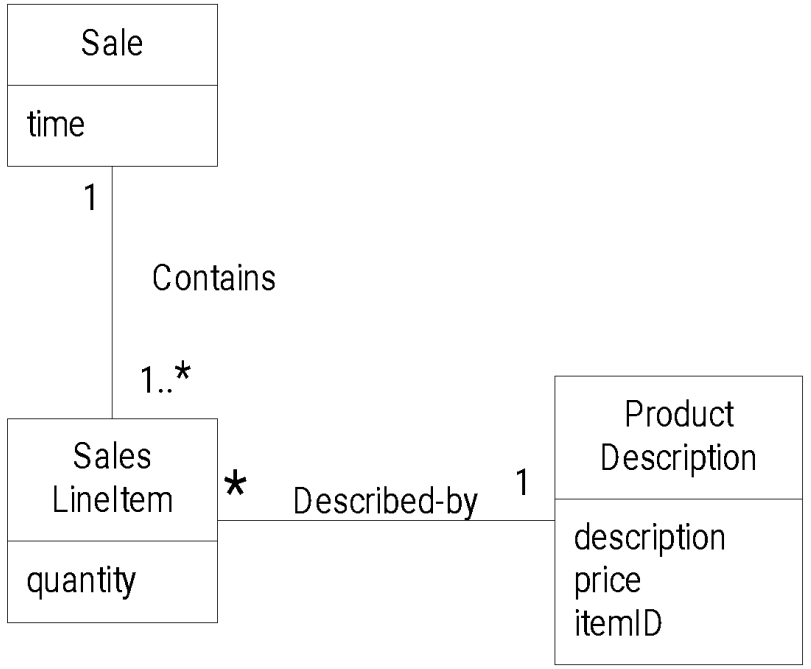
  Problem: What is the basic principle by which to assign responsibilities to object

  Solution: Assign a responsibility to the class that has the information needed to respond to it.

- Just because an object has information necessary doesn't mean it will have responsibility for action related to the information

Information Expert

- They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.
- **Example:** (from POS system) Who should be responsible for knowing the grand total of a sale?



- Start with domain model and look for associations with Sale
- What information is necessary to calculate grand total and which objects have the information?

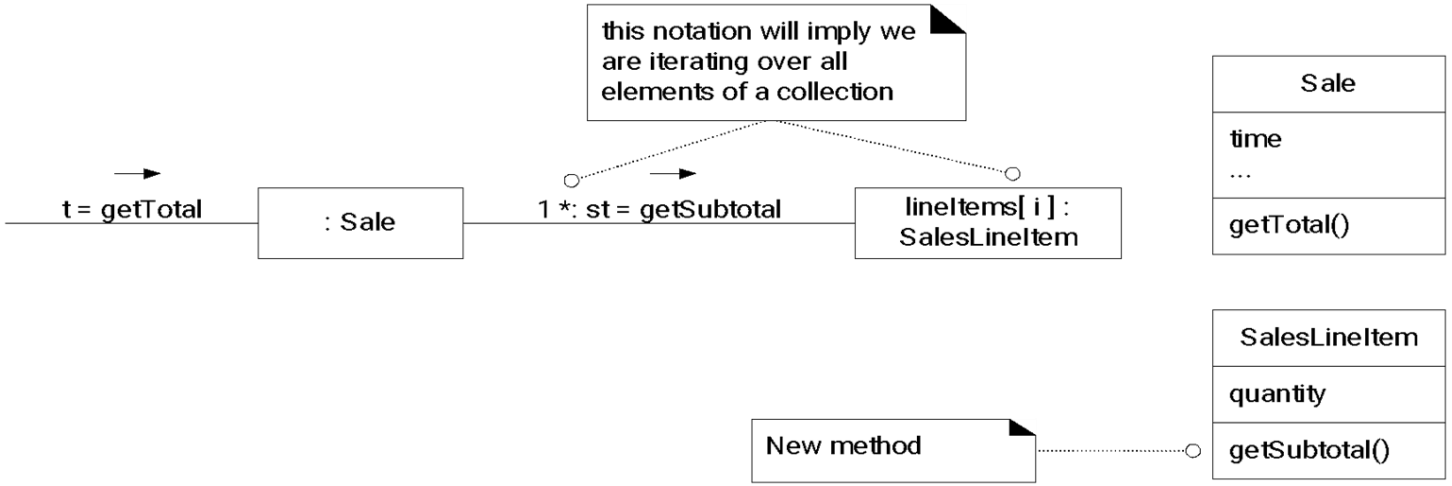
Information Expert

- From domain model: “Sale” Contains SalesLineItems. So, it has the information necessary for total

- How is line item subtotal determined?

  quantity – an attribute of SalesLineItem

  price – stored in ProductDescription

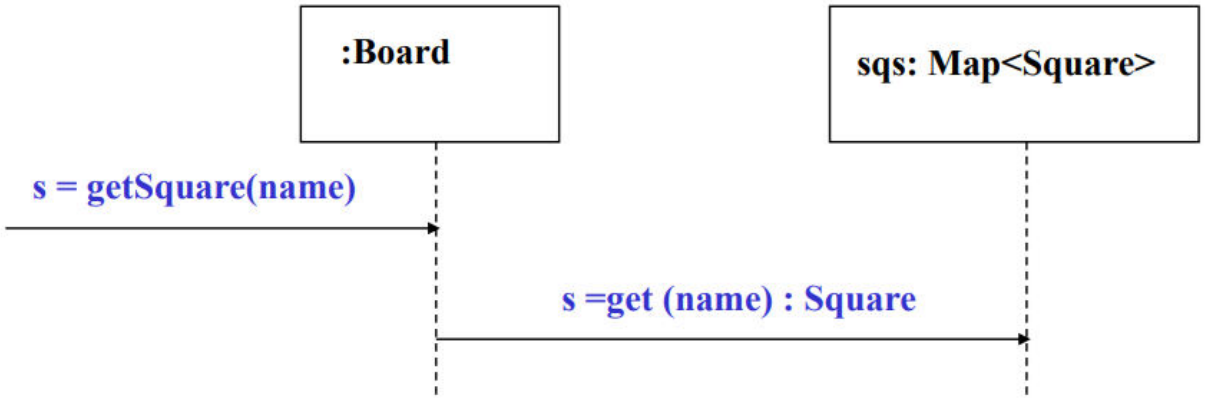


Information Expert - Summary

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductDescription	knows product price

Information Expert –Monopoly game

- The player Marker needs to find the square to which it is to move and the options pertaining to that square.
- The Board aggregates all of the Squares, so the Board has the Information needed to fulfill this responsibility.



Make board information expert

Low Coupling

- **Coupling** – A measure of how strongly one element is connected to, has knowledge of, or relies on other elements . Deals with how much information the component knows about other components
- **Definition:**

  Name: Low Coupling
  Problem: How to support low dependency, low change impact, increased reuse?
  Solution: Assign a responsibility so coupling is low.

- **Minimizes the dependency** hence making system **maintainable, efficient and code reusable**
- **Example: Inheritance, Composition, aggregation, association**

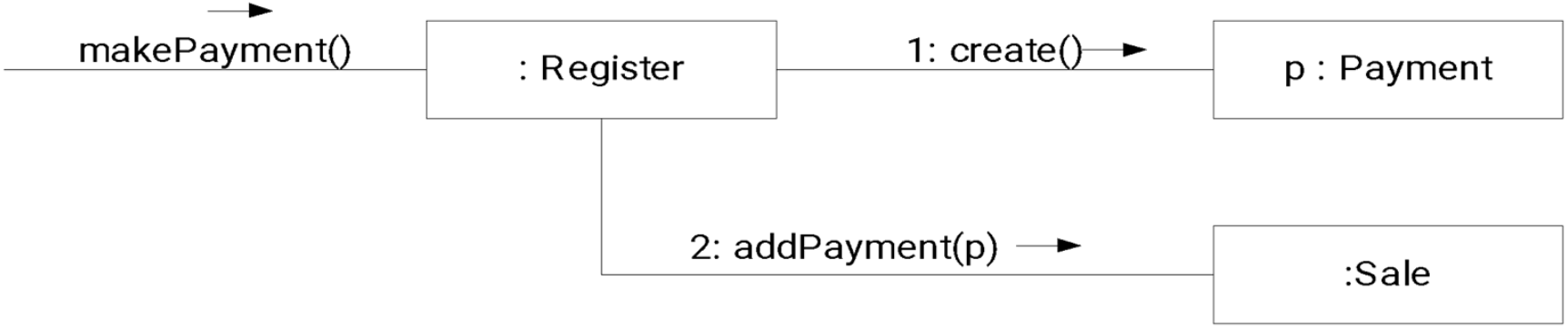
Low Coupling contd..

- A class with high coupling leads to problems:
  - Changes in related classes force local changes
  - Harder to understand in isolation
  - Harder to reuse
- Low Coupling – How can we reduce the impact of change in depended elements on dependant elements? **Assign responsibilities so that coupling remains low.**
- Two **elements are coupled**, if – One element has aggregation/composition association with another element. – One element implements/extends other element

Low Coupling - Example

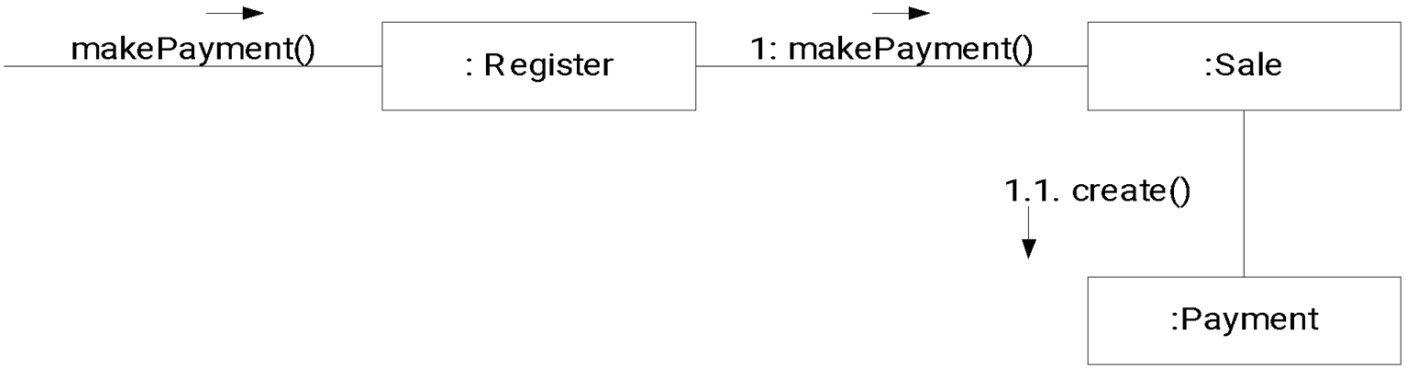
- **Example** (from POS system): Consider Payment, Register, Sale
- To create a Payment and associate it with a Sale, who is responsible?
- Register "records" a Payment. So, **Register is the creator**
- Register creates Payment *p* then sends *p* to a Sale => **coupling of**

Register class to Payment class



Low Coupling - Example

- **Alternate approach:** Register requests Sale to create the Payment



Note:

- In both cases, Sale needs to know about a Payment
- However, a Register needs to know about a Payment in first but not in second
- **Second approach has lower coupling**

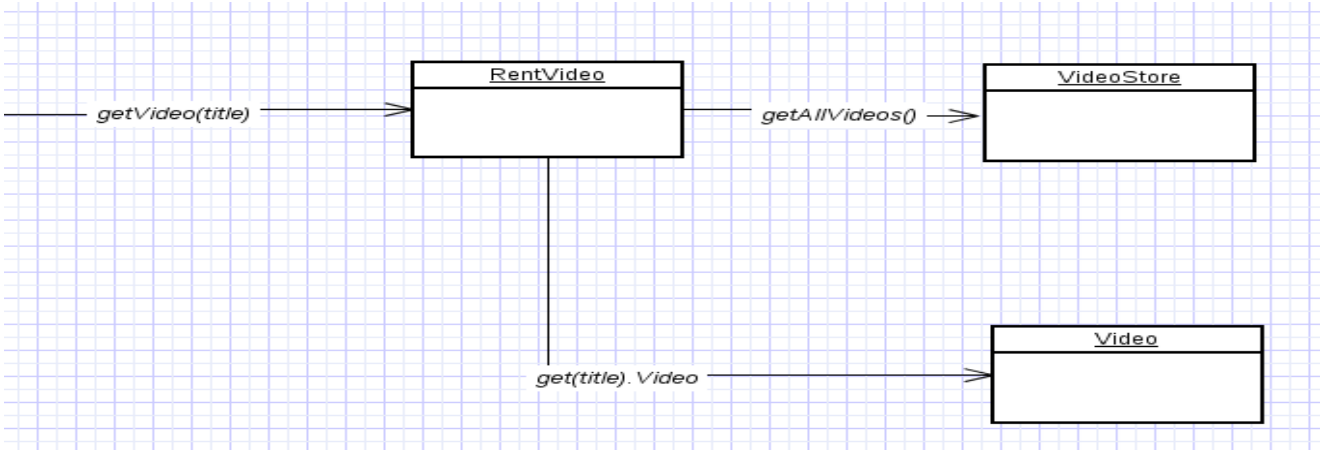
Low Coupling – More in terms of Java

- Low coupling is an evaluative principle, i.e., keep it in mind when evaluating designs
- Examples of coupling in Java (think “dependencies”):
  - TypeX has an attribute of TypeY
  - TypeX calls on services of a TypeY object
  - TypeX has a method that references an instance of TypeY
  - TypeX is a subclass of TypeY
  - TypeY is an interface and TypeX implements the interface

Note: subclassing => high coupling

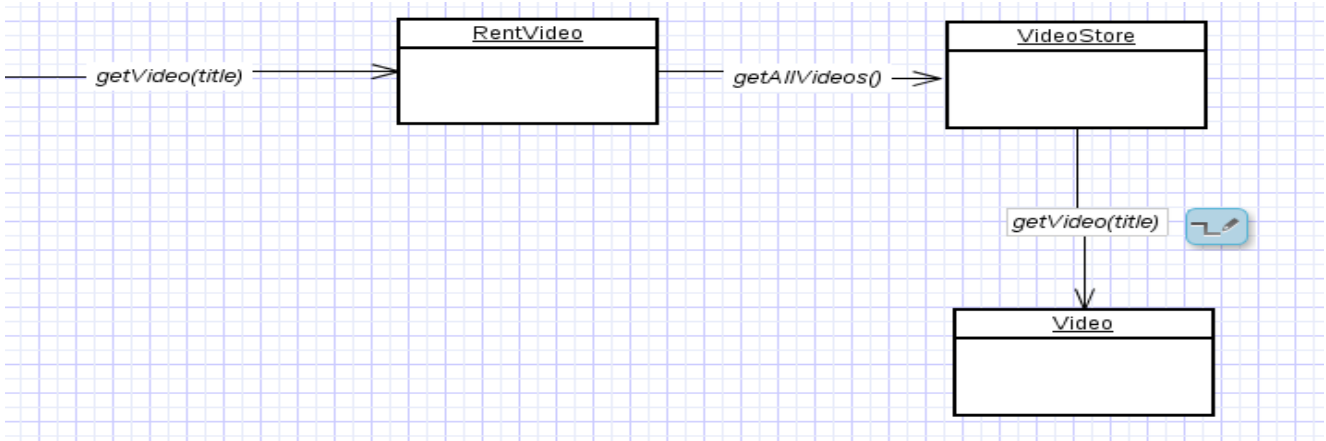
Example: Low Coupling

High Coupling



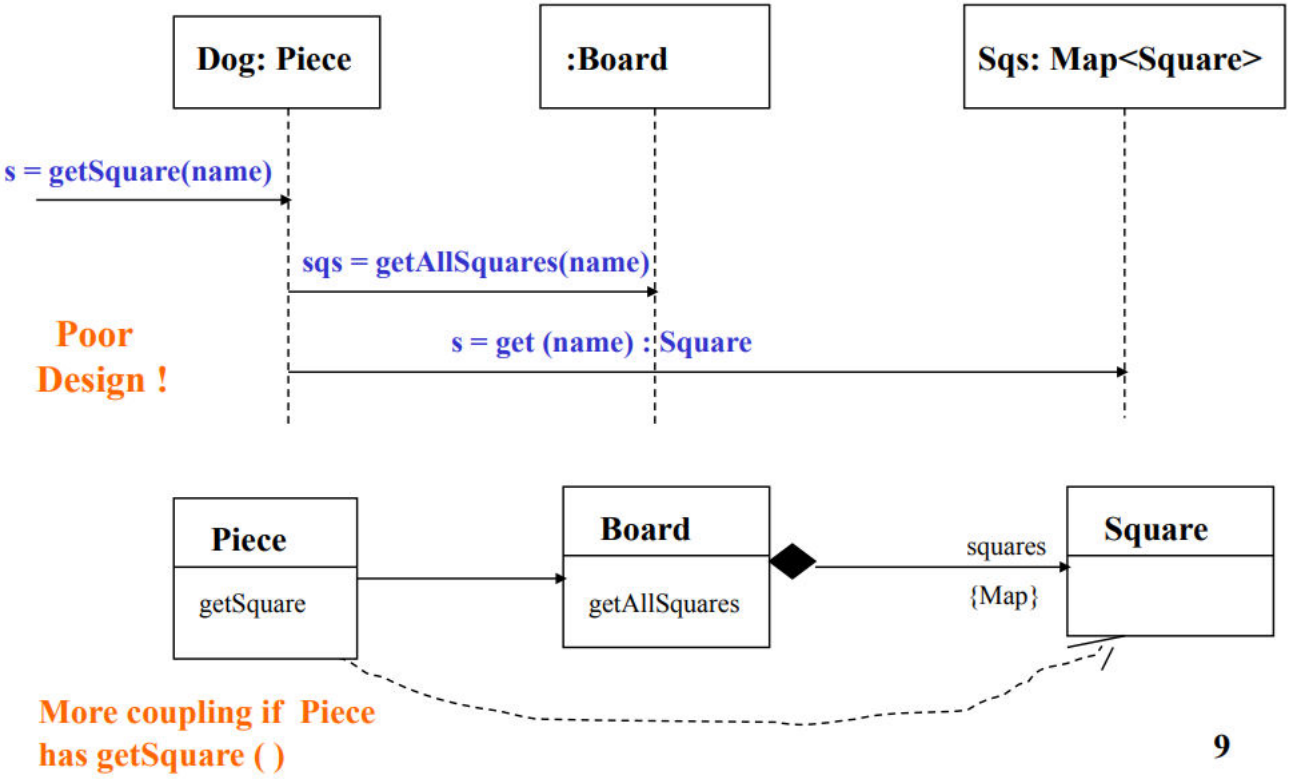
here class RentVideo knows about both VideoStore and Video objects. Rent is depending on both the classes.

Low Coupling



VideoStore and Video class are coupled, and Rent is coupled with VideoStore. **Thus, providing low coupling.**

Low Coupling – Monopoly example



Alternate Design: Bad solution

Controller

- Deals with **how to delegate the request from the UI layer objects to domain layer objects.**
- Helps in **minimizing the dependency** between GUI components and the system operation classes
- When a request comes from UI layer object, Controller as a pattern helps us in determining which is the first object that should receive the message from the UI layer objects. – **Controller Object**
- Delegates the work to other class and coordinates all activities.
- **Benefits** - Can reuse the controller class. Can use to maintain the state of the use case. Can control the sequence of the activities

Controller contd..

- **Definition**

Problem: Who should be responsible for handling a system event? (Or, what object receives and coordinates a system operation?)

Solution: Assign the responsibility for receiving and/or handling a system event to one of following choices:

Object that represents overall system, device or subsystem (façade controller)

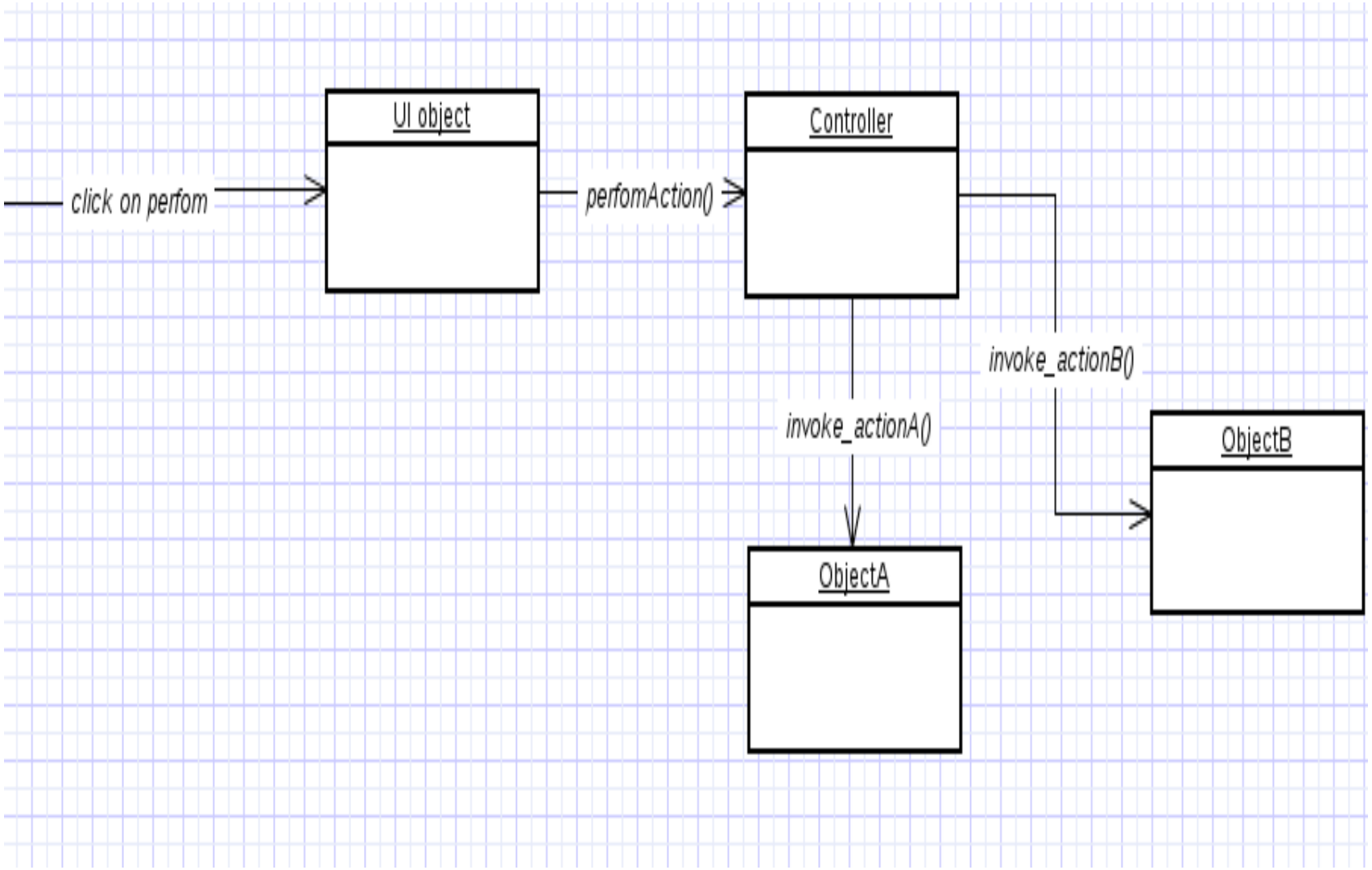
Object that represents a use case scenario within which the system event occurs (a <UseCase>Handler)

- **Input system event** – Event generated by an external actor associated with a system operation

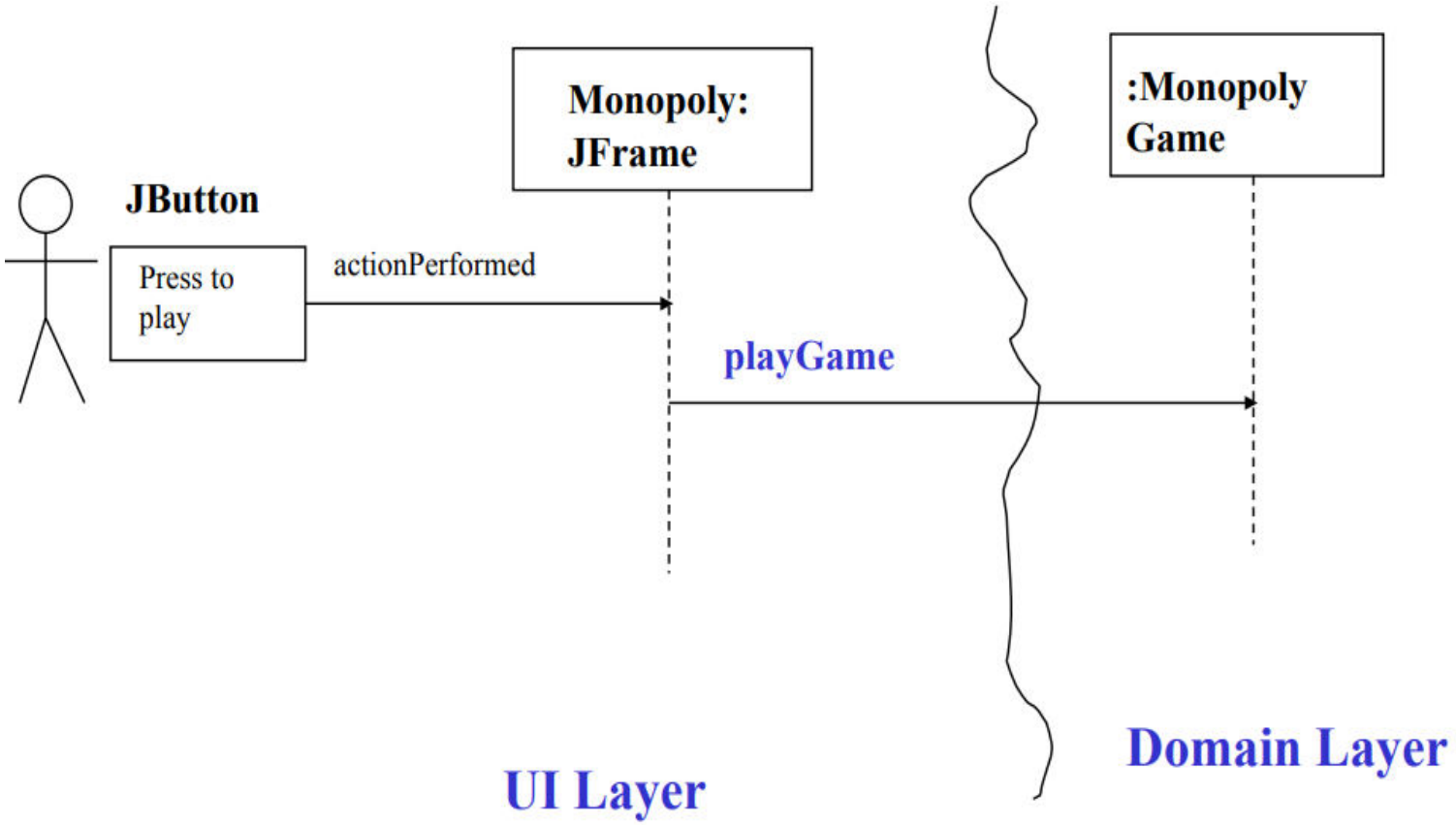
Controller contd..

- Normally controller coordinates activity but delegates work to other objects rather than doing work itself
- **Façade Controller :** A class that represents the overall system, device subsystems
- **Use case Controller:** A class that represent a use case, whereby performs handling particular system operations Often uses same controller class for all system events of one-use case so that one can maintain state information, e.g., events must occur in a certain order
- **Bloated Controller**
  - Single class receiving all system events and there are many of them
  - Controller performs many tasks rather than delegating them
  - Controller has many attributes and maintains significant information about system which should have been distributed among other objects

Example: Controller



Example: Monopoly example



High cohesion

- **Cohesion** - A measure of how strongly related and focused the responsibilities of an element (class, subsystem, etc.)
- **Definition:**

  Name: High Cohesion

  Problem: How to keep complexity manageable, understandable, manageable, and support low coupling?

  Solution: Assign the responsibility so that cohesion remains high.

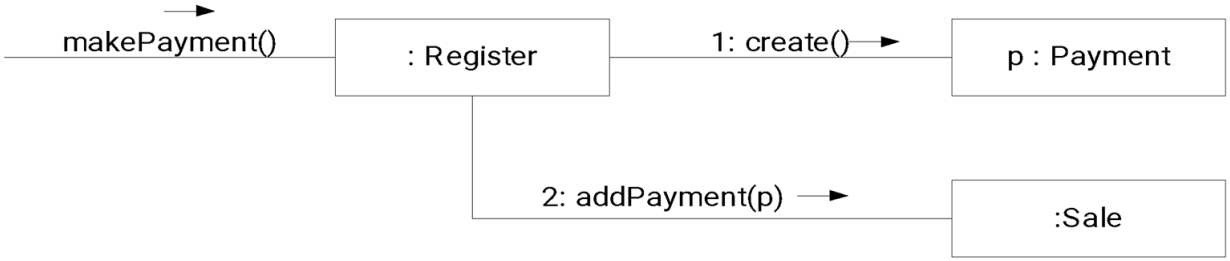
- Problems from low cohesion (does many unrelated things or does too much work):

  Hard to understand/comprehend
  Hard to reuse
  Hard to maintain
  Brittle – easily affected by change

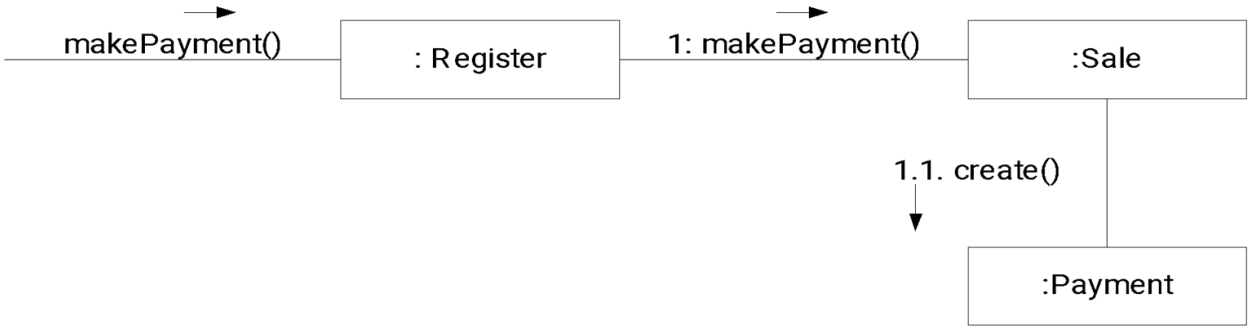
Example: High Cohesion

Example: Who should be responsible for creating a Payment instance and associate it with a Sale?

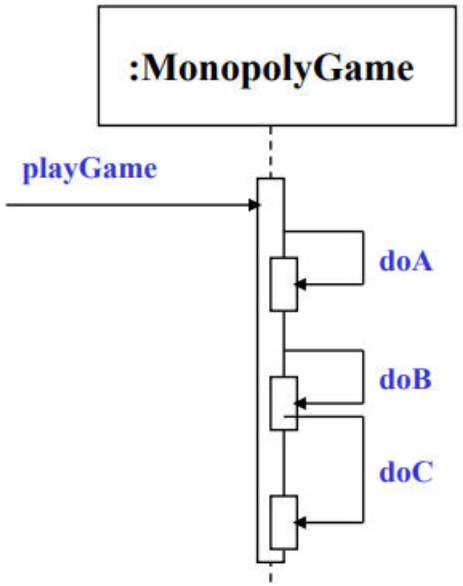
1. Register creates a Payment *p* then sends addPayment(*p*) message to the Sale



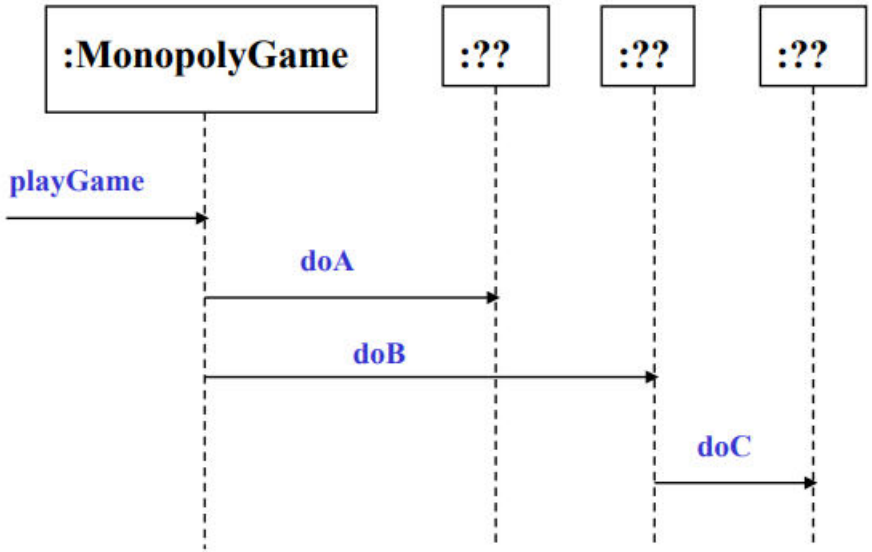
1. Register delegates Payment creation to the Sale



High Cohesion - Monopoly example



Poor (low) Cohesion in the MonopolyGame object



Better Design

Polymorphism

- Deals with How to act different depending on object's class
- How to handle related but varying elements based on type?
- Guides us in deciding which object is responsible for handling those varying elements.
- Types:
  - Adhoc - Overloading
  - Parametric – Early binding
  - Inclusion – Sub typing
  - Coercion - Casting

Polymorphism contd..

- **Definition:**

Name: Polymorphism

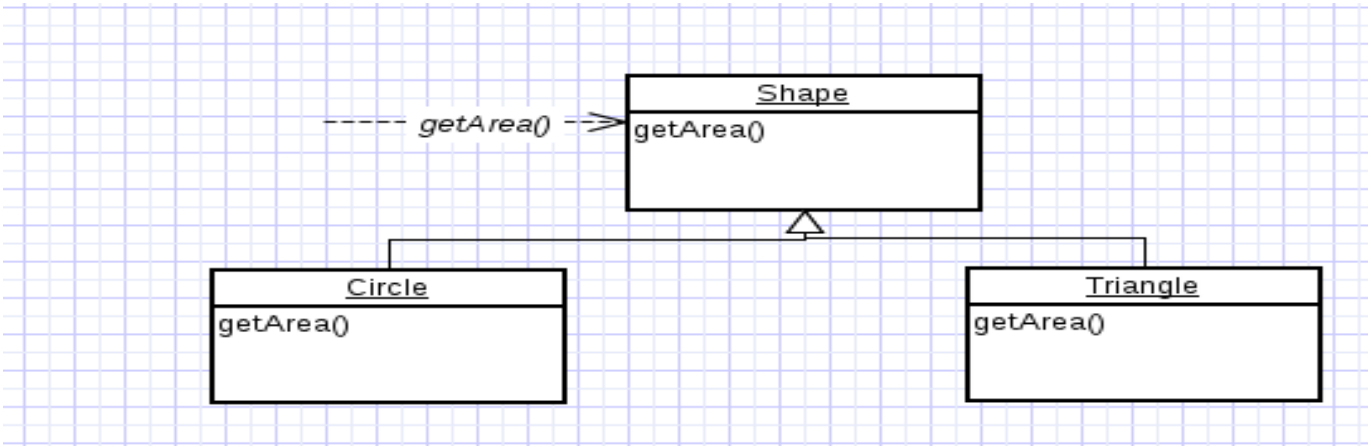
Problem: How to handle alternatives based on type.

Pluggable software components -- how can you replace one server component with another without affecting the client?

Solution: When related alternatives or behaviors vary by type (class), assign responsibility using polymorphic operations – to the types for which the behavior varies. In this context, polymorphism means giving the same name to similar or related services

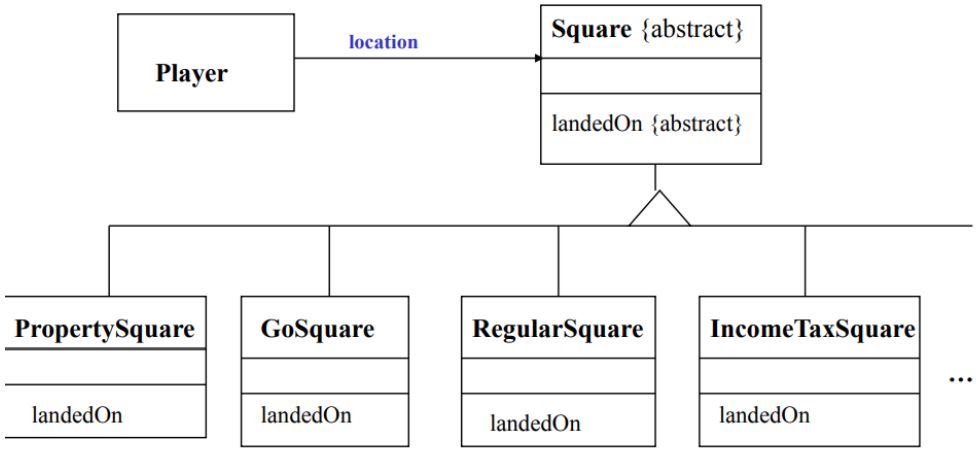
Polymorphism : Example

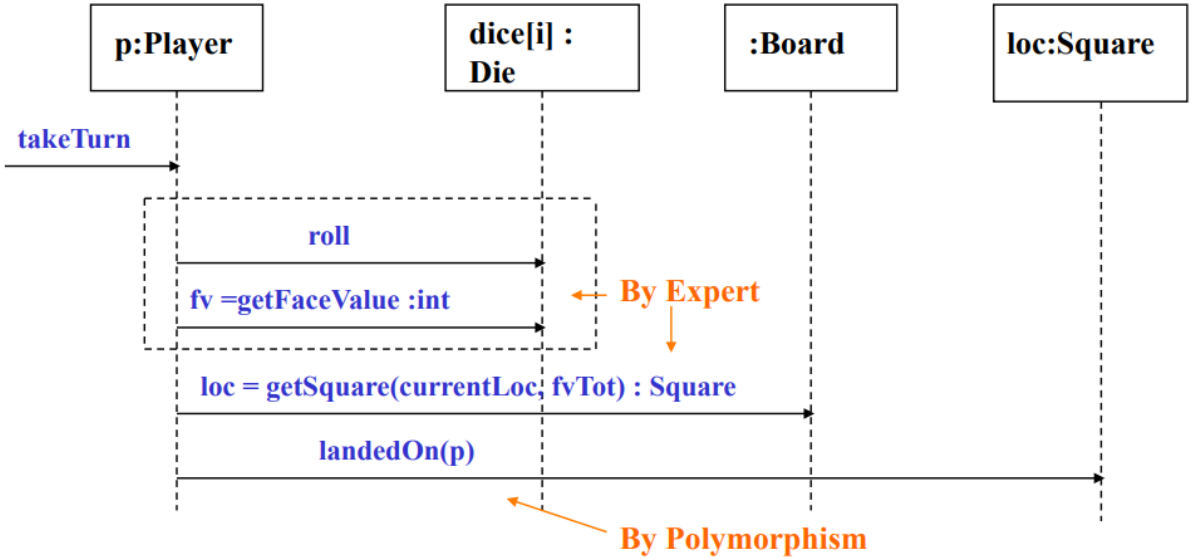
- The getArea() varies by the type of shape, so we assign that responsibility to the subclasses
- By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle



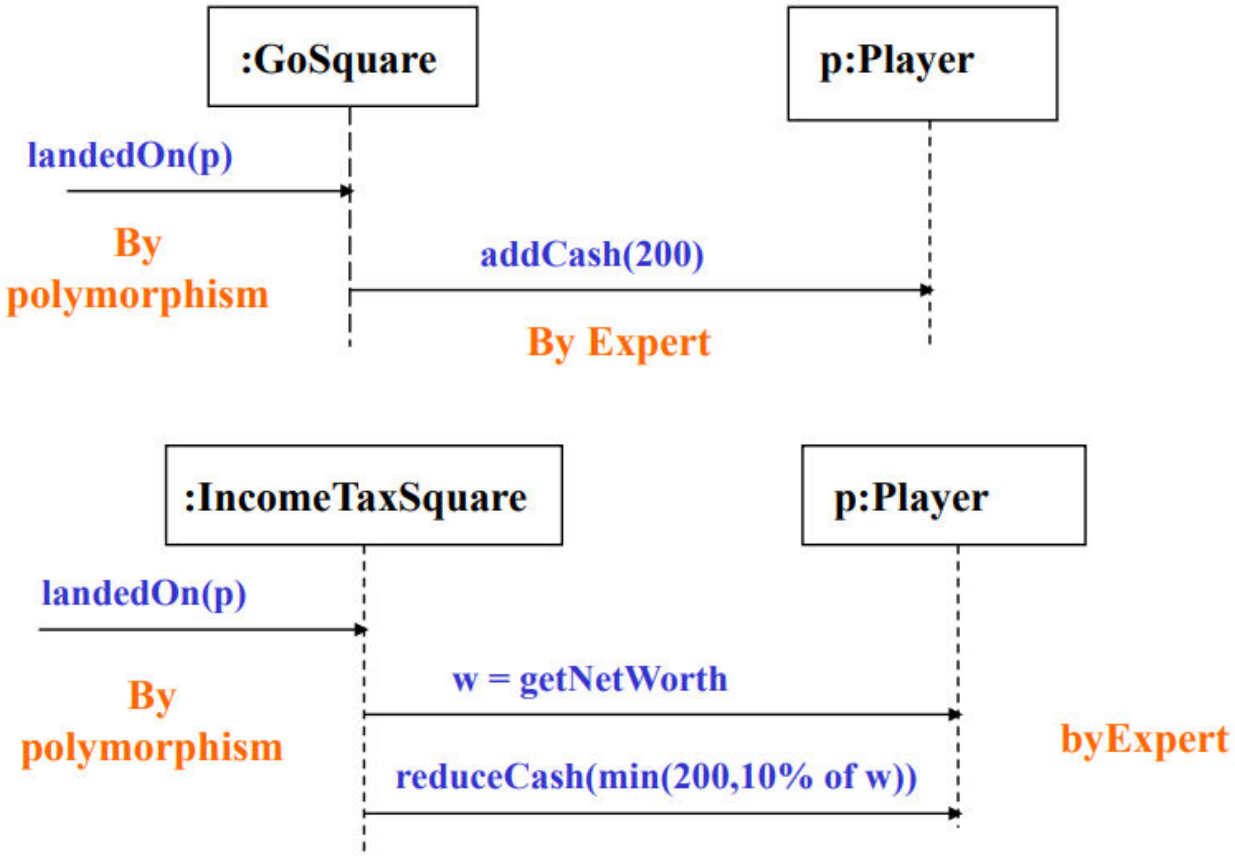
Object Oriented Analysis and Design using Java

Polymorphism : Monopoly game





Polymorphism : Monopoly game



Polymorphic cases

Indirection

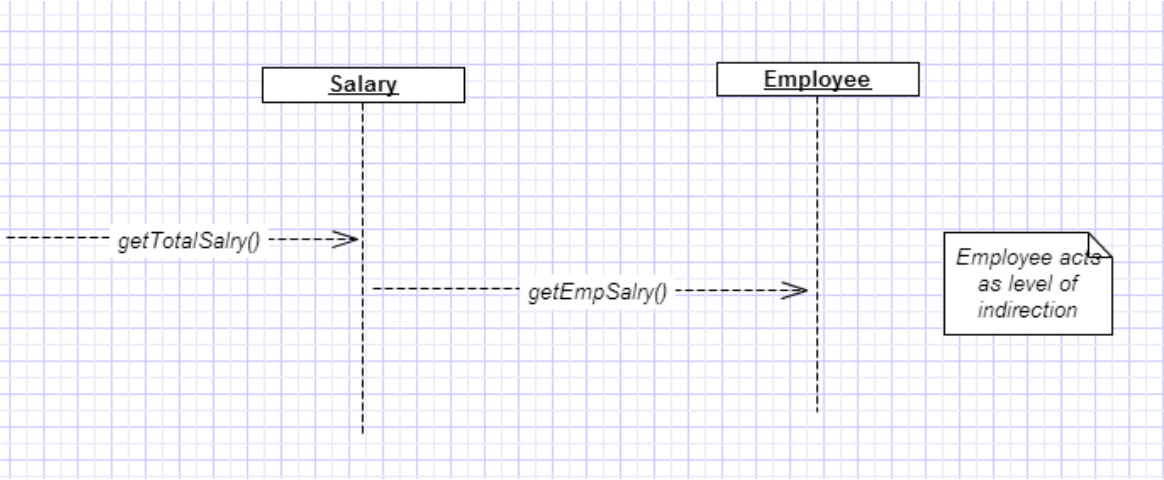
- **Definition:**

  Name: Indirection

  Problem**:** How to assign responsibilities in order to avoid direct coupling between two components and keep ability for reuse.

  Solution: Assign responsibility to intermediate class for providing linking between objects not linking directly

- Related Design patterns: Adapter, Bridge, Mediator.



Pure Fabrication

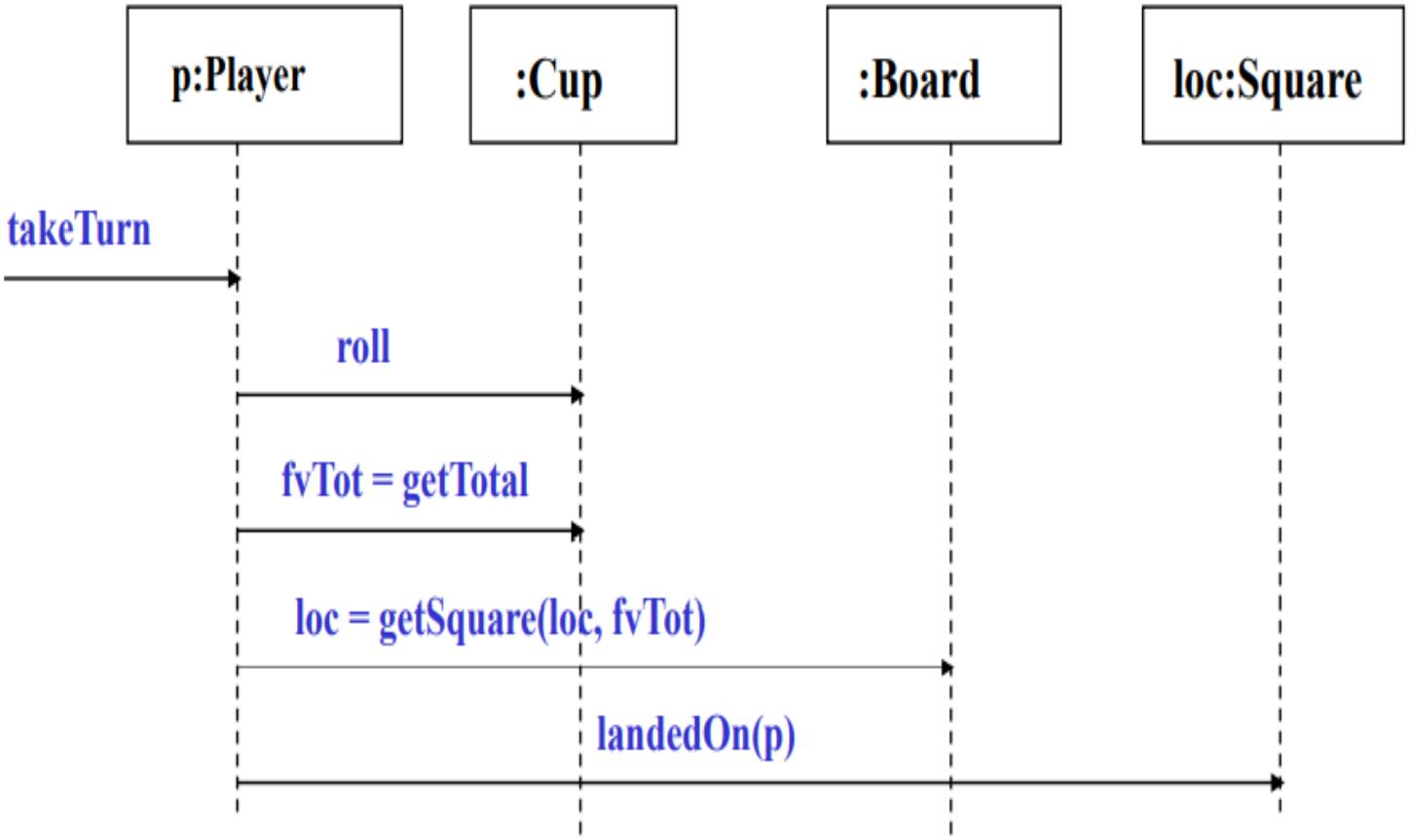
- **Definition:**

Name: Pure Fabrication

Problem: What object should have responsibility when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate? Sometimes assigning responsibilities only to domain layer software classes leads to problems like poor cohesion or coupling, or low reuse potential.

Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a domain concept

Pure Fabrication – Monopoly game



Use a Cup to hold the dice, roll them, and know their total. It can be reused in many different applications where dice are involved.

Controlled /Protected variation

- **Definition:**

Name: Protected Variation

Problem: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?.

Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

- Example:- Open-Closed Principle similar to CV/PV pattern

- Protected variation pattern applying for both **variation and Evolution points**

References

- Source Code Examples
- Ad-hoc, Inclusion, Parametric & Coercion Polymorphisms – GeeksforGeeks
- Larman - Monopoly Game.pdf - THE MONOPOLY GAME SYSTEM • The software version of the game will run as a simulation • One person will start the game and | Course Hero
- http://www.kamilgrzybek.com/design/grasp-explained/

THANK YOU

Prof K V N M Ramesh
kvnmramesh@pes.edu
Department of Computer Science and Engineering

Object Oriented Analysis and Design using Java

Dr. L. Kamatchi Priya

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

PES UNIVERSITY
CELEBRATING 50 YEARS

UE20CS352:
Object Oriented Analysis and Design using Java

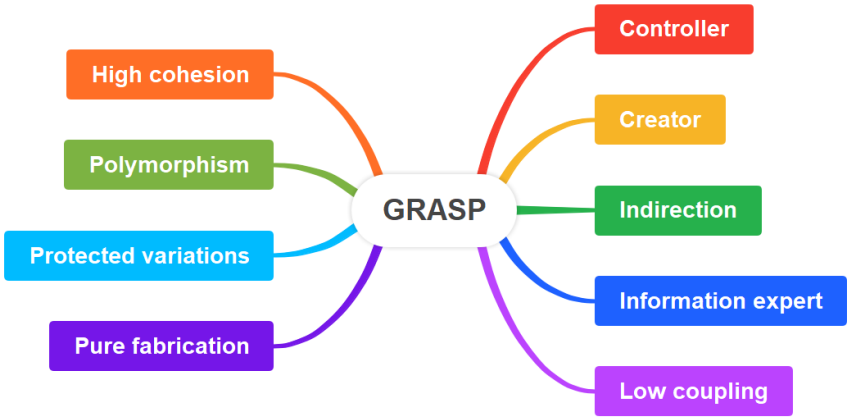
Design Exercise: Creating Simple Systems using GRASP

Dr. L. Kamatchi Priya

Department of Computer Science and Engineering

Recap - GRASP

- GRASP (General Responsibility Assignment Software Patterns) is a set of guidelines that helps designers make informed decisions during the object-oriented design process.
- It help guide object-oriented design by clearly outlining who does what: which object or class is responsible for what action or role. GRASP also helps us define how classes work with one another.
- The nine principals of GRASP are



Object Oriented Analysis and Design using Java
Creator

Analyze the provided Java code, which defines classes for Course, Student, and Department. The Course class has attributes courseCode and courseName, the Student class has attributes studentId, studentName, and an associated Course instance, and the Department class has attributes departmentName and an associated Course instance.

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenges might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of Creator. Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle. Include the modified Department class, ensuring that the creation of Course instances is centralized.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

Object Oriented Analysis and Design using Java
Creator

Before Implementation: The responsibility of creating instances of the Course class is scattered across different classes, leading to code duplication.

01

Code organization and modularity in Course class.

02

Efficient process for instantiating Course class.

03

Reducing code duplication through centralized instantiation.

Object Oriented Analysis and Design using Java
Creator

```java
public class Course {
    private String courseCode;
    private String courseName;

    public Course(String courseCode, String courseName) {
        this.courseCode = courseCode;
        this.courseName = courseName;
    }

    // Getters and setters for courseCode and courseName
}
```

```java
public class Department {
    private String departmentName;
    private Course course;

    public Department(String departmentName) {
        this.departmentName = departmentName;
        // Creating a Course instance within the Department class
        this.course = new Course("ENG201", "Advanced English Writing");
    }

    // Getters and setters for departmentName and course
}
```

```java
public class Student {
    private int studentId;
    private String studentName;
    private Course course;

    public Student(int studentId, String studentName) {
        this.studentId = studentId;
        this.studentName = studentName;
        // Creating a Course instance within the Student class
        this.course = new Course("CS101", "Introduction to Computer Science");
    }

    // Getters and setters for studentId, studentName, and course
}
```

Before Implementation of Creator Principle: The responsibility of creating instances of the Course class is scattered across different classes, leading to code duplication.

Creator


```java
public class Course {
    private String courseCode;
    private String courseName;

    public Course(String courseCode, String courseName) {
        this.courseCode = courseCode;
        this.courseName = courseName;
    }

    // Getters and setters for courseCode and courseName
}
```

Object Oriented Analysis and Design using Java
Creator

```java
public class Student {
    private int studentId;
    private String studentName;
    private Course course;

    public Student(int studentId, String studentName, Course course) {
        this.studentId = studentId;
        this.studentName = studentName;
        this.course = course;
    }

    // Getters and setters for studentId, studentName, and course
}
```

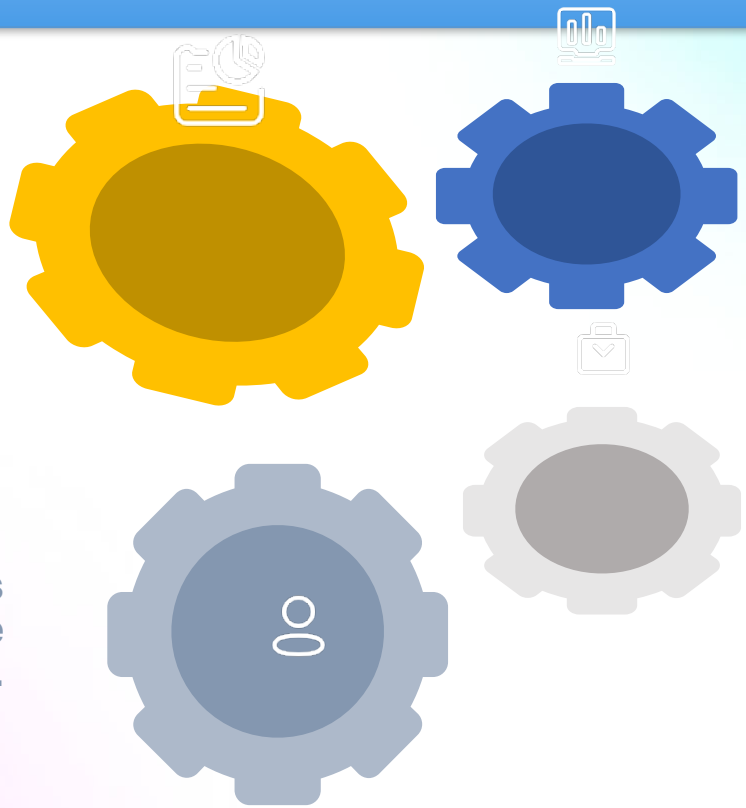
Creator

```
    public Department(String departmentName) {
        this.departmentName = departmentName;
    }

    public Course createCourse(String courseCode, String courseName) {
        // Centralized creation of Course instance within the Department class
        return new Course(courseCode, courseName);
    }

    public void assignCourseToStudent(Student student, Course course) {
        // Additional logic for assigning a course to a student
        student.setCourse(course);
    }

    // Getters and setters for departmentName
}
```

Creator

After Implementation: The responsibility of creating instances of the Course class is assigned to the Department class, which has all the necessary information to initialize a new Course object. This eliminates code duplication and centralizes the creation logic.

The content is about centralizing creation logic in the Department class.



The responsibility is to create Course instances assigned to a Department.

The Course class aims to remove code duplication.

The Department class contains essential information for initializing a Course object.

Information Expert

Analyze the provided Java code:

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenges might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of Information Expert. Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

Information Expert

```
class ShoppingCart {
    List<Item> items;

    void addItem(Item item) {
        // Add item to the shopping cart
    }

    double calculateTotal() {
        // Calculate total price of items in the shopping cart
    }
}

class Item {
    String name;
    double price;
    // other properties and methods
}
```

Information Expert

- After applying Information Expert:

```
class ShoppingCart {
    List<Item> items;

    void addItem(Item item) {
        // Add item to the shopping cart
    }

    double calculateTotal() {
        double total = 0;
        for (Item item : items) {
            total += item.getPrice();
        }
        return total;
    }
}

class Item {
    String name;
    double price;

    double getPrice() {
        return price;
    }
    // other properties and methods
}
```

Information Expert

Explanation:

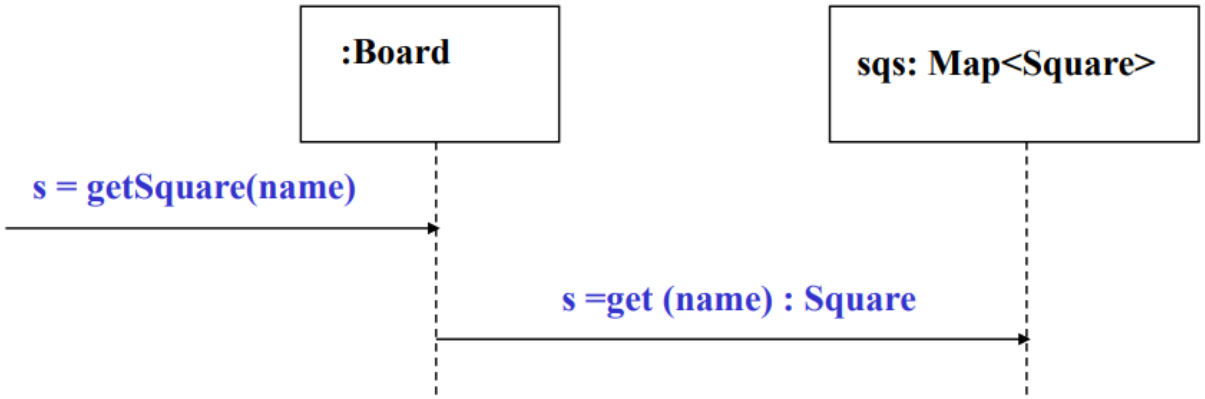
In the "Before" example, the calculateTotal method in the ShoppingCart class accesses the price of items directly, violating the principle of Information Expert.

After applying the principle, the responsibility of calculating the total price is moved to the Item class, which has the necessary information about its price.

Information Expert –Monopoly game

• The player Marker needs to find the square to which it is to move and the options pertaining to that square.

•The Board aggregates all of the Squares, so the Board has the Information needed to fulfill this responsibility.



Low Coupling

Analyze the provided Java code

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenges might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of Low Coupling. Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

Low Coupling contd..

Before applying Low Coupling:

```
class ShoppingCart {
    PaymentProcessor paymentProcessor;

    void checkout() {
        paymentProcessor.processPayment();
    }
}

class PaymentProcessor {
    void processPayment() {
        // Process payment
    }
}
```

Low Coupling - Example

After applying Low Coupling:

```
class ShoppingCart {
    PaymentProcessor paymentProcessor;

    void setPaymentProcessor(PaymentProcessor paymentProcessor) {
        this.paymentProcessor = paymentProcessor;
    }

    void checkout() {
        paymentProcessor.processPayment();
    }
}

class PaymentProcessor {
    void processPayment() {
        // Process payment
    }
}
```

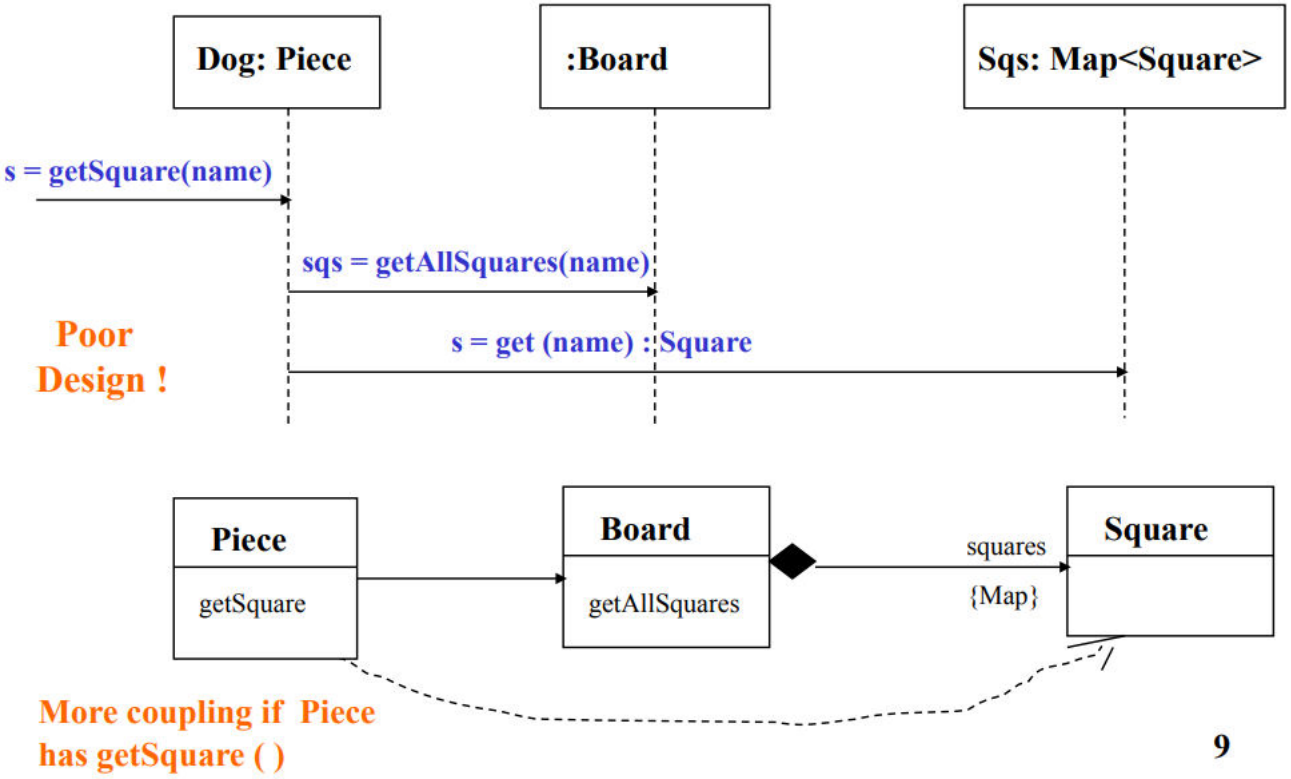
Low Coupling Explanation:

Explanation:

In the "Before" example, the `ShoppingCart` class is tightly coupled to the `PaymentProcessor` class, making it difficult to replace or modify the payment processing logic.

After applying the Low Coupling principle, the dependency between the `ShoppingCart` and `PaymentProcessor` classes is reduced by introducing a setter method to inject the `PaymentProcessor` instance, allowing for easier swapping of implementations.

Low Coupling – Monopoly example



Alternate Design: Bad solution

Controller

Analyze the provided Java code

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenges might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of Controller. Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

Controller contd..

Before applying Controller:

```
class ShoppingCart {
    void checkout() {
        // Handle checkout process
    }
}

class OrderProcessor {
    void processOrder() {
        // Process the order
    }
}
```

Object Oriented Analysis and Design using Java

Controller contd..

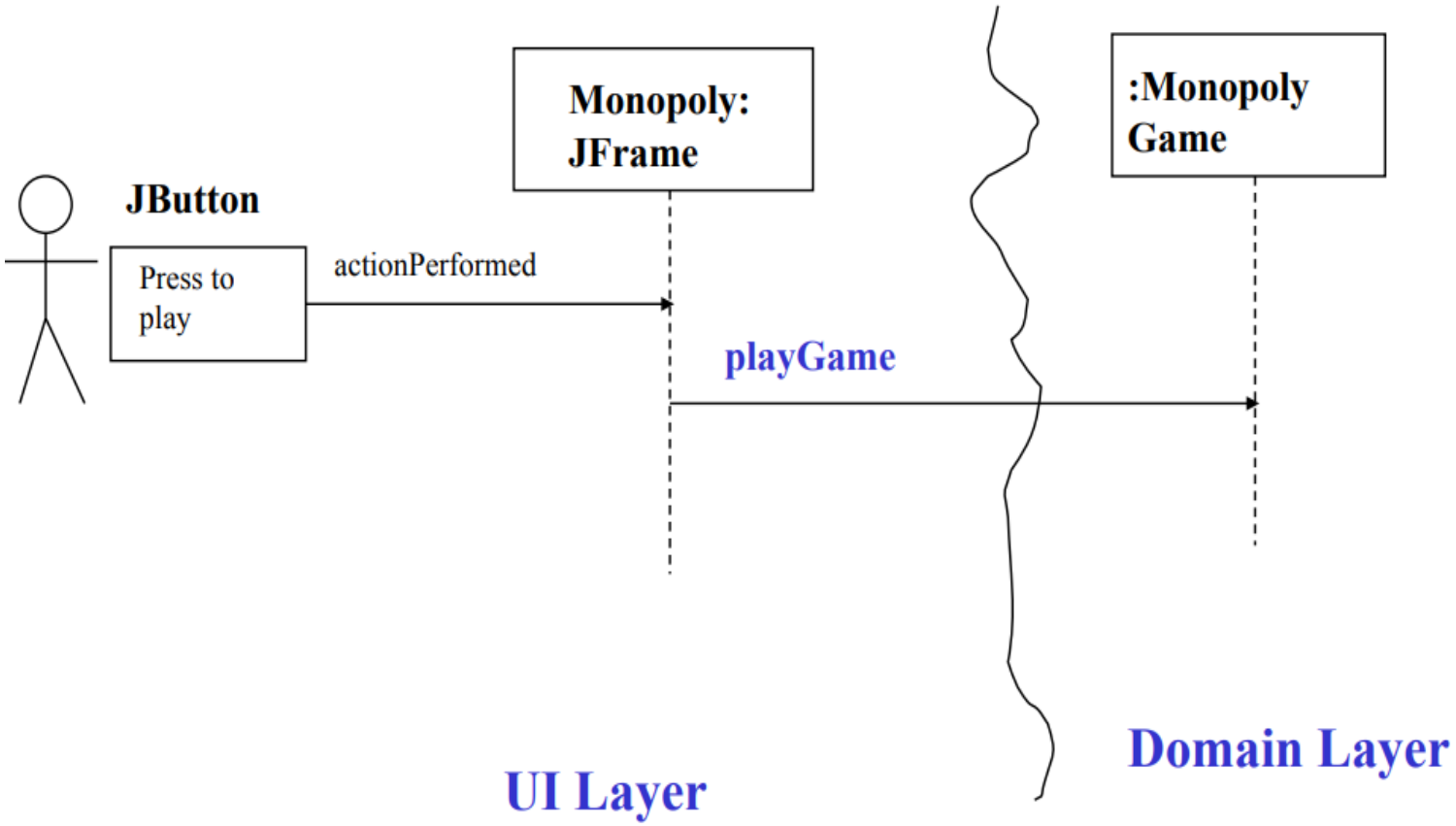
After applying Controller:

```
class ShoppingCartController {
    ShoppingCart cart;

    void checkout() {
        cart.checkout();
    }
}

class ShoppingCart {
    void checkout() {
        // Handle checkout process
    }
}

class OrderProcessor {
    void processOrder() {
        // Process the order
    }
}
```

Controller - Explanation

Explanation: In the "Before" example, the responsibilities of handling the checkout process and processing orders are spread across multiple classes. After applying the Controller principle, a dedicated controller class (`ShoppingCartController`) is introduced to handle the checkout process, which delegates the responsibility to the `ShoppingCart` class.

Example: Monopoly example



High cohesion

Analyze the provided Java code

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenges might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of High cohesion
- Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

High Cohesion

Before applying High Cohesion:

```
class ShoppingCart {
    List<Item> items;

    void addItem(Item item) {
        // Add item to the shopping cart
    }

    double calculateTotal() {
        // Calculate total price of items in the shopping cart
    }

    void processOrder() {
        // Process the order
    }
}
```

High Cohesion

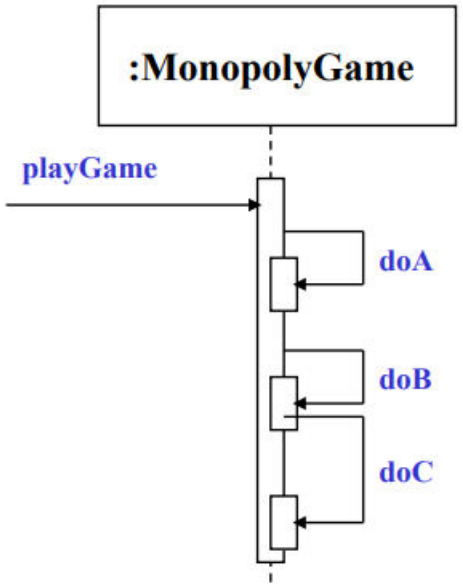
After applying High Cohesion:

```
class ShoppingCart {
    List<Item> items;

    void addItem(Item item) {
        // Add item to the shopping cart
    }

    double calculateTotal() {
        // Calculate total price of items in the shopping cart
    }
}

class OrderProcessor {
    void processOrder(ShoppingCart cart) {
        // Process the order
    }
}
```

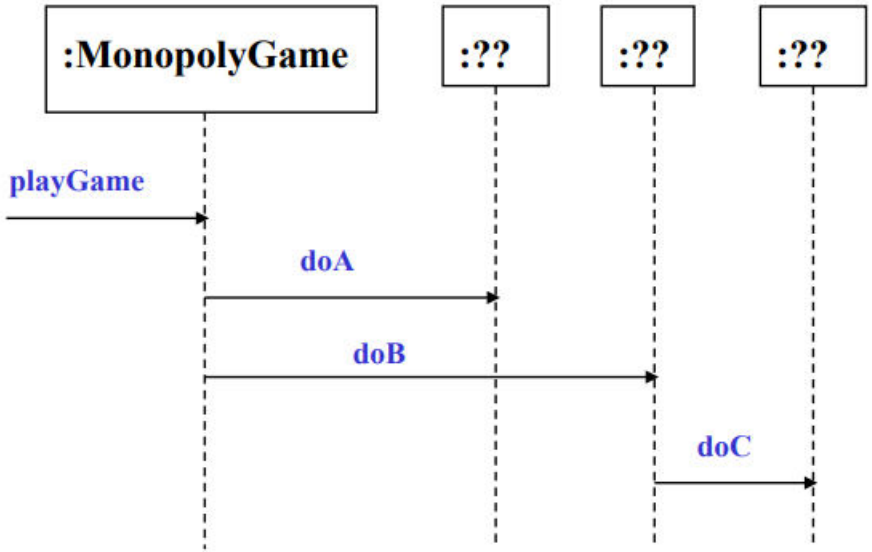
High Cohesion - Explanation

Explanation: In the "Before" example, the `ShoppingCart` class has multiple responsibilities including managing items, calculating the total, and processing orders, leading to low cohesion. After applying the High Cohesion principle, the `OrderProcessor` class is introduced to handle the responsibility of processing orders, resulting in higher cohesion for both classes.

High Cohesion - Monopoly example



Poor (low) Cohesion in the MonopolyGame object



Better Design

Polymorphism

Analyze the provided Java code

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenge might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of Polymorphism. Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

Object Oriented Analysis and Design using Java

Polymorphism contd..

Before applying Polymorphism:

```
class Shape {
    void draw() {
        // Draw shape
    }
}

class Circle extends Shape {
    @Override
    void draw() {
        // Draw circle
    }
}

class Square extends Shape {
    @Override
    void draw() {
        // Draw square
    }
}
```

Polymorphism : Example

After applying Polymorphism:

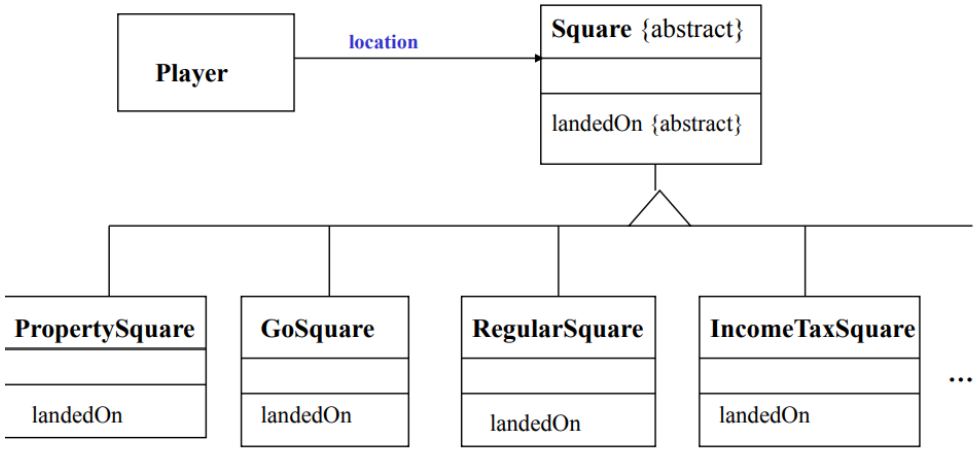
```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    @Override
    void draw() {
        // Draw circle
    }
}

class Square extends Shape {
    @Override
    void draw() {
        // Draw square
    }
}
```

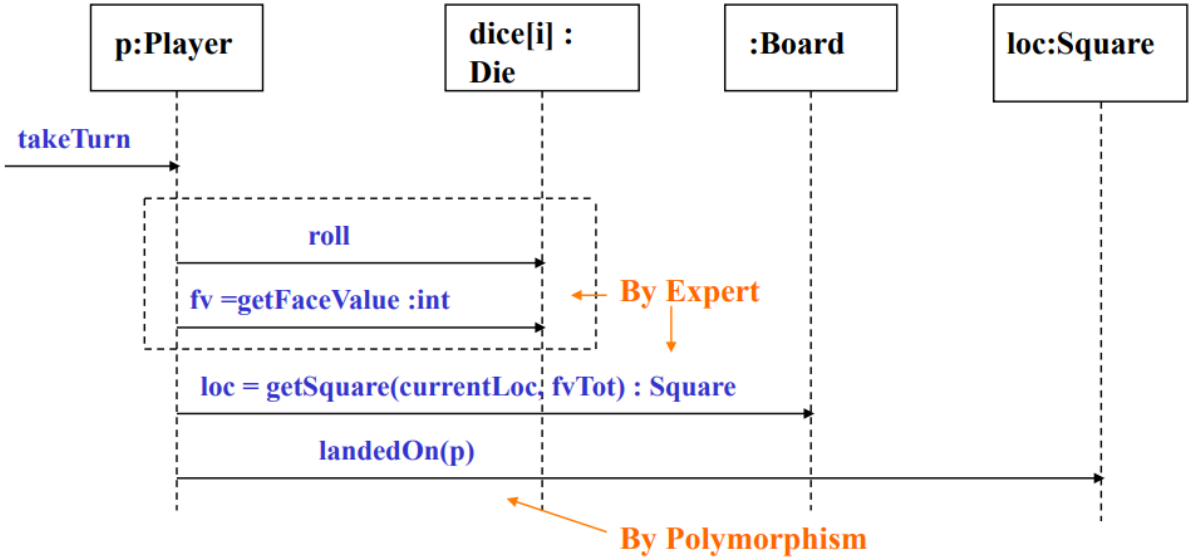
Polymorphism - Explanation:

Explanation: In the "Before" example, the `Shape` class has a method for drawing different shapes, but its implementation remains the same. After applying the Polymorphism principle, the `draw` method in the `Shape` class is made abstract, allowing subclasses like `Circle` and `Square` to provide their own implementation of the `draw` method based on their specific shape.

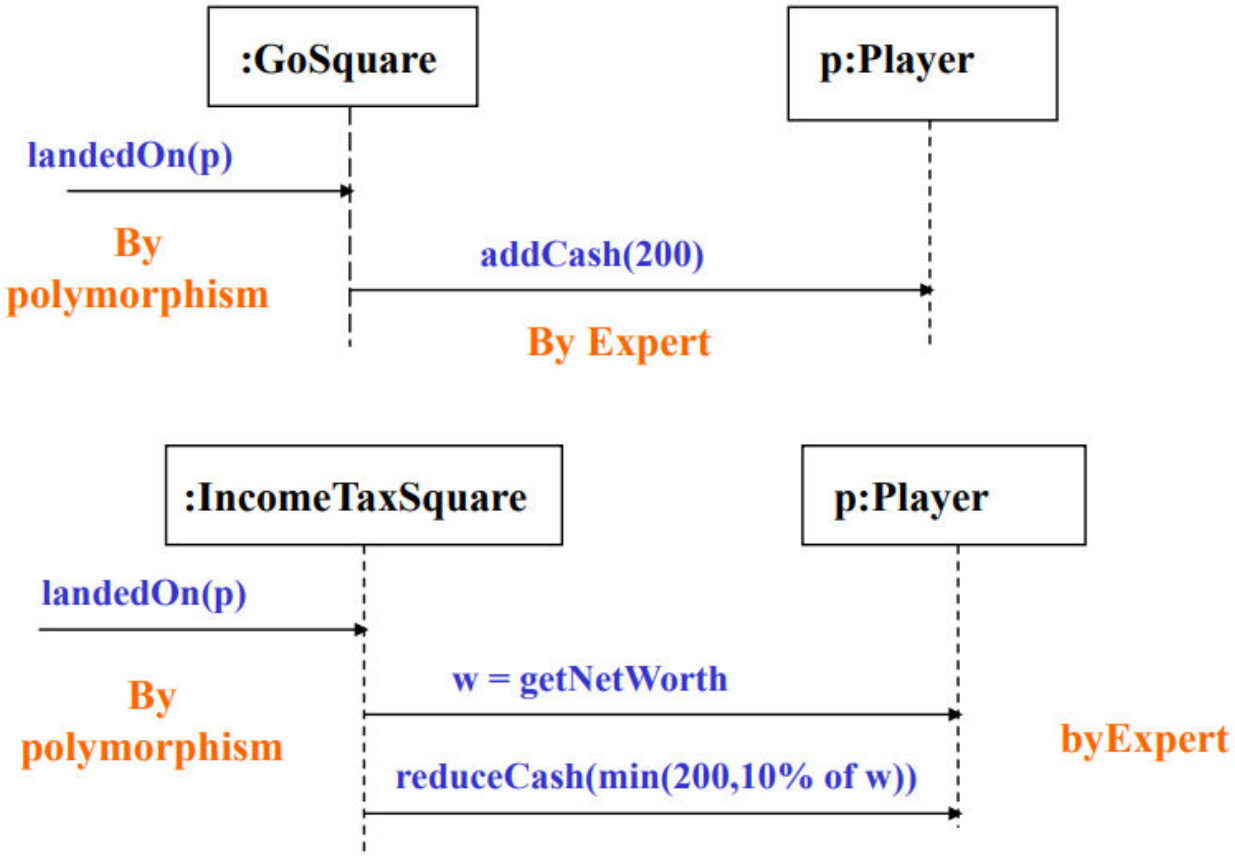
Object Oriented Analysis and Design using Java

Polymorphism : Monopoly game





Polymorphism : Monopoly game



Polymorphic cases

Indirection

Analyze the provided Java code

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenges might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of Indirection. Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

Indirection

Before applying Indirection:

```
class ShoppingCart {
    PaymentProcessor paymentProcessor;

    void checkout() {
        paymentProcessor.processPayment();
    }
}

class PaymentProcessor {
    void processPayment() {
        // Process payment
    }
}
```

Indirection

After applying Indirection:

```
class ShoppingCart {
    PaymentGateway paymentGateway;

    void checkout() {
        paymentGateway.processPayment();
    }
}

class PaymentGateway {
    PaymentProcessor paymentProcessor;

    void processPayment() {
        paymentProcessor.processPayment();
    }
}
```

Indirection

Explanation: In the "Before" example, the ShoppingCart class is tightly coupled to the PaymentProcessor class. After applying the Indirection principle, the PaymentGateway class is introduced as an intermediary, allowing the ShoppingCart class to interact with the PaymentGateway instead of directly with the PaymentProcessor, thus reducing direct dependencies.

Pure Fabrication

Analyze the provided Java code

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenges might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of Pure Fabrication. Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

Pure Fabrication

Before applying Pure Fabrication:

```
class ShoppingCart {
    PaymentProcessor paymentProcessor;

    void checkout() {
        paymentProcessor.processPayment();
    }
}

class PaymentProcessor {
    void processPayment() {
        // Process payment
    }
}
```

Pure Fabrication

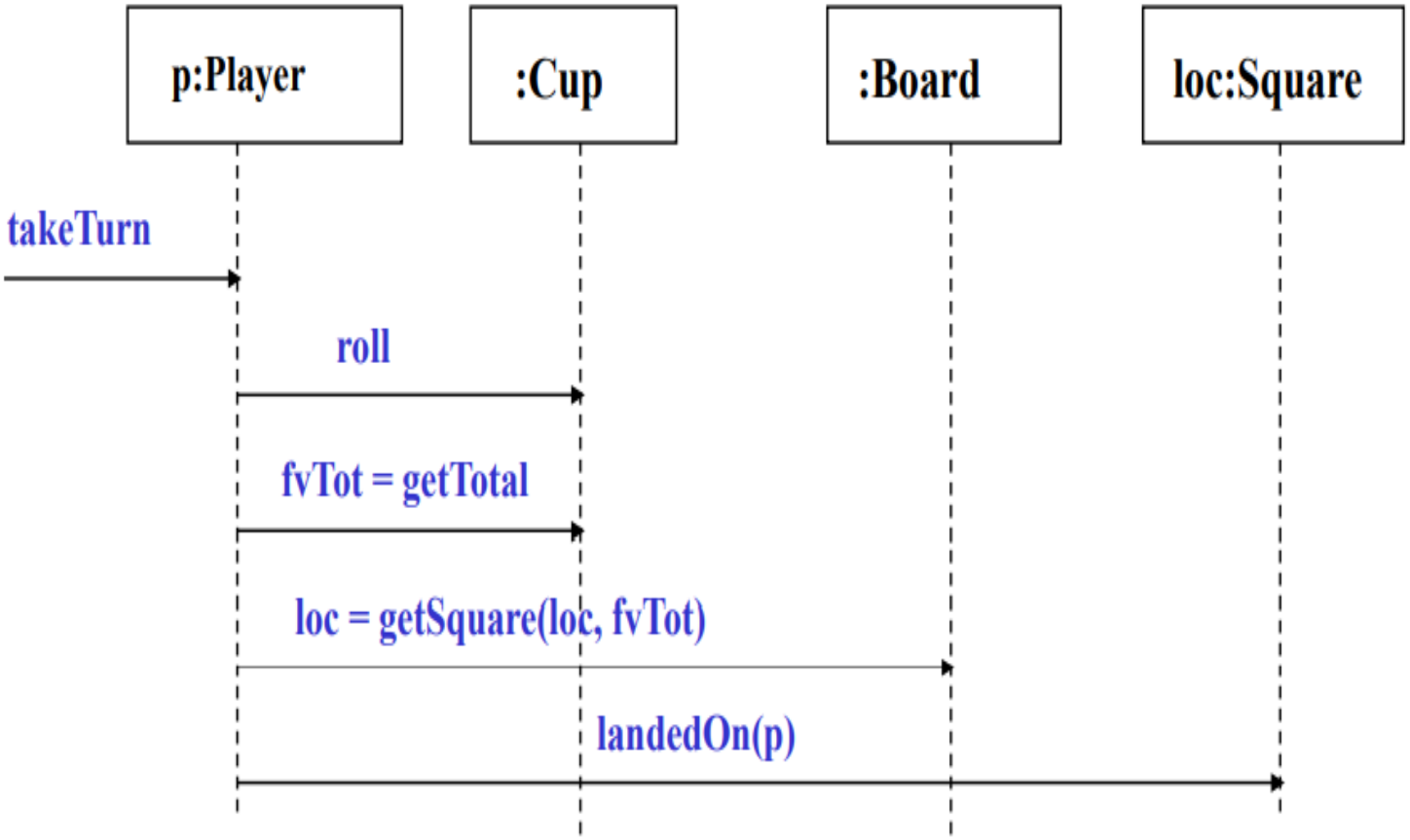
After applying Pure Fabrication:

```
class ShoppingCart {
    PaymentGateway paymentGateway;

    void checkout() {
        paymentGateway.processPayment();
    }
}

class PaymentGateway {
    void processPayment() {
        // Process payment
    }
}
```

Pure Fabrication - Explanation:

Explanation: In the "Before" example, the PaymentProcessor class represents a concept related to payment processing, but it may not naturally exist in the problem domain. After applying the Pure Fabrication principle, the PaymentGateway class is introduced to handle payment processing, abstracting away the payment logic from the ShoppingCart class.

Pure Fabrication – Monopoly game



Use a Cup to hold the dice, roll them, and know their total. It can be reused in many different applications where dice are involved.

Controlled /Protected variation

Analyze the provided Java code

- Identify a potential issue with the current implementation in terms of object-oriented design principles.
- How does the code violate the principle of code reusability, and what challenges might arise from the current design?
- Propose a solution to improve the code by applying the GRASP principle of Protected variations. Explain how this solution enhances the design and addresses the identified issue.
- Provide a refactored version of the code that aligns with the suggested improvement using the GRASP principle.
- Discuss the advantages of the refactored code in terms of maintainability, flexibility, and adherence to object-oriented design principles.

Controlled /Protected variation

Before applying Protected Variations:

```
class WeatherStation {
    WeatherSensor sensor;

    void readTemperature() {
        double temperature = sensor.getTemperature();
        // Process temperature reading
    }
}

class WeatherSensor {
    double getTemperature() {
        // Read temperature from sensor
        return 25.0; // Dummy value for demonstration
    }
}
```

Object Oriented Analysis and Design using Java

Controlled /Protected variation

After applying Protected Variations:

```java
interface WeatherSensor {
    double getTemperature();
}

class ExternalWeatherSensor implements WeatherSensor {
    @Override
    public double getTemperature() {
        // Read temperature from external sensor
        return 25.0; // Dummy value for demonstration
    }
}

class InternalWeatherSensor implements WeatherSensor {
    @Override
    public double getTemperature() {
        // Read temperature from internal sensor
        return 27.0; // Dummy value for demonstration
    }
}

class WeatherStation {
    WeatherSensor sensor;

    WeatherStation(WeatherSensor sensor) {
        this.sensor = sensor;
    }

    void readTemperature() {
        double temperature = sensor.getTemperature();
        // Process temperature reading
    }
}
```

Controlled /Protected variation

Explanation: In the "Before" example, the WeatherStation class directly depends on the WeatherSensor class, making it vulnerable to changes in the sensor implementation. After applying the Protected Variations principle, the WeatherStation class depends on the WeatherSensor interface, allowing different sensor implementations (ExternalWeatherSensor and InternalWeatherSensor) to be easily swapped without affecting the WeatherStation class.

THANK YOU

Dr. L. Kamatchi Priya

priyal@pes.edu

Department of Computer Science and Engineering

Object Oriented Analysis and Design Using Java

UE21CS352B

Prof. Shilpa S

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B: Object Oriented Analysis and Design using Java

OO Design Principles and Sample Implementation of Patterns in Java

SOLID: Single Responsibility

Prof. Shilpa S

Department of Computer Science and Engineering

Object Oriented Analysis and Design using Java

Agenda

- Important aspects of Bad Design
- Good Design
- Introduction – SOLID
- SRP
- Implementation of SRP
- References

Object Oriented Analysis and Design using Java

Important Aspects of Bad Design

Rigidity

- the impact of a change is unpredictable
- every change causes a cascade of changes in dependent modules
- costs become unpredictable

Fragility

- the software tends to break in many places on every change
- the breakage occurs in areas with no conceptual relationship
- on every fix the software breaks in unexpected ways

Immobility

- it's almost impossible to reuse interesting parts of the software
- the useful modules have too many dependencies
- the cost of rewriting is less compared to the risk faced to separate those parts

Viscosity

- a hack is cheaper to implement than the solution within the design
- preserving design moves are difficult to think and to implement
- it's much easier to do the wrong thing rather than the right one

Object Oriented Analysis and Design using Java

Good Design

- What's the reason why a design becomes rigid, fragile, immobile, and viscous?

Improper dependencies between modules

- Characteristics of a good design

High cohesion

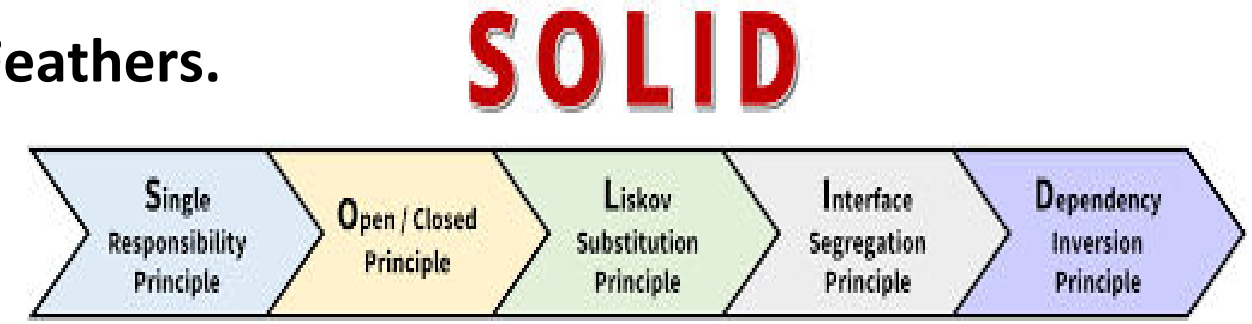
Low coupling

- How can we achieve a good design? - SOLID

Object Oriented Analysis and Design using Java

Introduction to SOLID

- Is an acronym for 5 important design principles introduced by **Robert J. Martin**.
- Leads to more **flexible and stable** software architecture that's easier to maintain and extend, and **less likely to break**
- Acronym identified by **Michael Feathers.**



- SOLID principles are software design coding standards
- Helps us to obtain good software with low coupling and high cohesion
- Applied to reduce dependencies - changes in one part of software will not be impacting others.

Object Oriented Analysis and Design using Java

Single Responsibility Principle(SRP)

- States that **every module or class should have responsibility over a single part of the functionality provided by the software**.
- A **class should have one, and only one, reason to change** - Robert C. Martin
  - Each class only does one thing.
  - Every class or module only has responsibility for one part of the software's functionality.
  - Ensures low coupling code.
  - Ensures easier coding to understand and maintain.
  - Can be applied to classes, software components, and microservices.
  - Code becomes easier to test and maintain, it makes software easier to implement, and it helps to avoid unanticipated side-effects of future changes.

Object Oriented Analysis and Design using Java

Violation of SRP

- The code violates the Single Responsibility Principle, as the Book class has two responsibilities.
- First, it sets the data related to the books (title and author).
- Second, it searches for the book in the inventory.

Note: The setter methods change the Book object, which might cause problems when we want to search the same book in the inventory.

```
class Book {
    String title;
    String author;

    String getTitle() {
        return title;
    }

    void setTitle(String title) {
        this.title = title;
    }

    String getAuthor() {
        return author;
    }

    void setAuthor(String author) {
        this.author = author;
    }

    void searchBook() {...}
}
```

Solution

- Identify things that are changing for different reasons
- Group together things that change for the same reason
- Decouple the responsibilities.
- In the refactored code, the Book class will only be responsible for getting and setting the data of the Book object.
- Create another class called InventoryView that will be responsible for checking the inventory.
- Move the searchBook() method to the new class and reference the Book class in the constructor.

Object Oriented Analysis and Design using Java

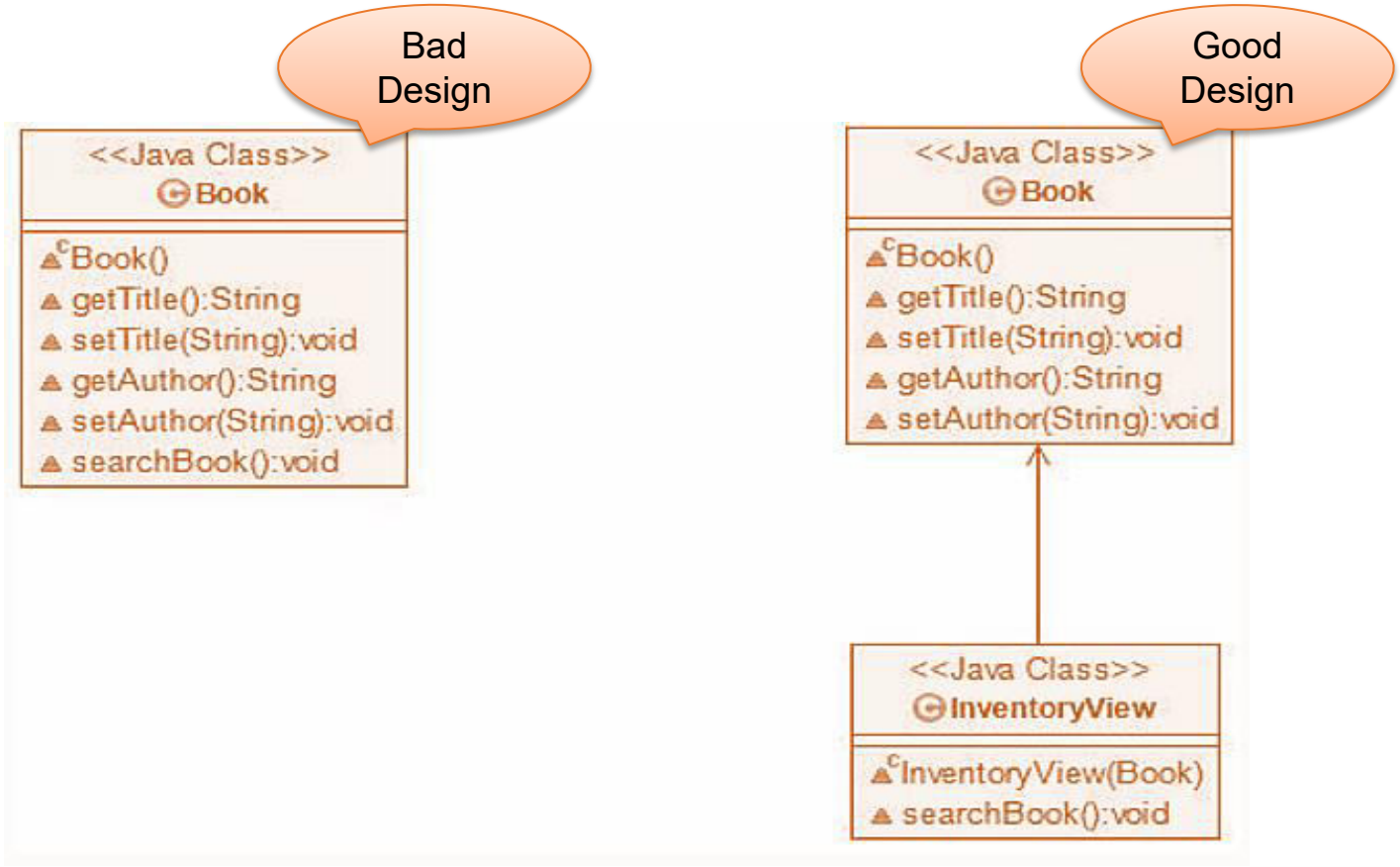
Good Design using SRP

```
class Book {
    String title;
    String author;

    String getTitle() {
        return title;
    }

    void setTitle(String title) {
        this.title = title;
    }

    String getAuthor() {
        return author;
    }

    void setAuthor(String author) {
        this.author = author;
    }
}
```

```
class InventoryView {
    Book book;

    InventoryView(Book book) {
        this.book = book;
    }

    void searchBook() {...}
}
```

Object Oriented Analysis and Design using Java

UML Class Diagram



Advantages

- **Testing** – A class with one responsibility will have fewer test cases.
- **Lower coupling** – Less functionality in a single class will have fewer dependencies.
- **Organization** – Smaller, well-organized classes are easier to search than monolithic ones.

Object Oriented Analysis and Design using Java

Implementation

Refer to Book or Student program

References

- Introduction To SOLID Principles (c-sharpcorner.com)
- A Solid Guide to SOLID Principles | Baeldung
- SOLID Principle in Programming: Understand With Real Life Examples - GeeksforGeeks

THANK YOU

Shilpa S

Department of Computer Science and Engineering

shilpas@pes.edu

Object Oriented Analysis and Design using Java

UE21CS352B

Prof. Sindhu R Pai

Teaching assistants : Nishanth M S

Vinay Padegal

Muskan Bansal

Department of Computer Science and Engineering

UE21CS352B: Object Oriented Analysis and Design using Java

OO Design Principles and Sample Implementation of Patterns in Java

Open – Closed Principle

Prof. Sindhu R Pai

Department of Computer Science and Engineering

Open-Closed Principle (OCP)

- States that **classes, modules, microservices, and other code units should be open for extension but closed for modification.**

- **Bertrand Meyer** originated the term OCP
  - A software entity must be easily extensible with new features without having to modify its existing code in use.
  - We should be able to **extend the existing code using OOP features like inheritance via subclasses and interfaces.**
  - Never modify classes, interfaces, and other code units that already exist, as it can lead to unexpected behavior.
  - While adding a new feature extend the code rather than modifying it, so that the risk of failure is minimized

Violation of OCP

- The store wants to hand out cookbooks at a discount price
- Based on the Single Responsibility Principle, Create two separate classes:

  CookbookDiscount to hold the details of the discount

  DiscountManager to apply the discount to the price

```
class CookbookDiscount {
    String getCookbookDiscount() {
        String discount = "30% between Dec 1 and 24";
        return discount;
    }
}

class DiscountManager {
    void processCookbookDiscount(CookbookDiscount
discount) {...}
}
```

Violation of OCP continued

The code works fine until the store management informs us that their cookbook discount sales were so successful that they want to extend it

```
class BiographyDiscount
{
    String getBiographyDiscount()
    {
    String discount = "50% on the
subject's birthday.";
    return discount;
    }
}
```

To process the new type of discount, we need to add the new functionality to the DiscountManager class, too:

```
class DiscountManager {
    void processCookbookDiscount(CookbookDiscount
discount) {...}
    void
processBiographyDiscount(BiographyDiscount
discount) {...}
}
```

Solution

- Refactor the code by adding an extra layer of abstraction that represents all types of discounts.
- Create a new interface called BookDiscount that, the CookbookDiscount and BiographyDiscount classes will implement.

```
public interface BookDiscount{
    String getBookDiscount();
}
```

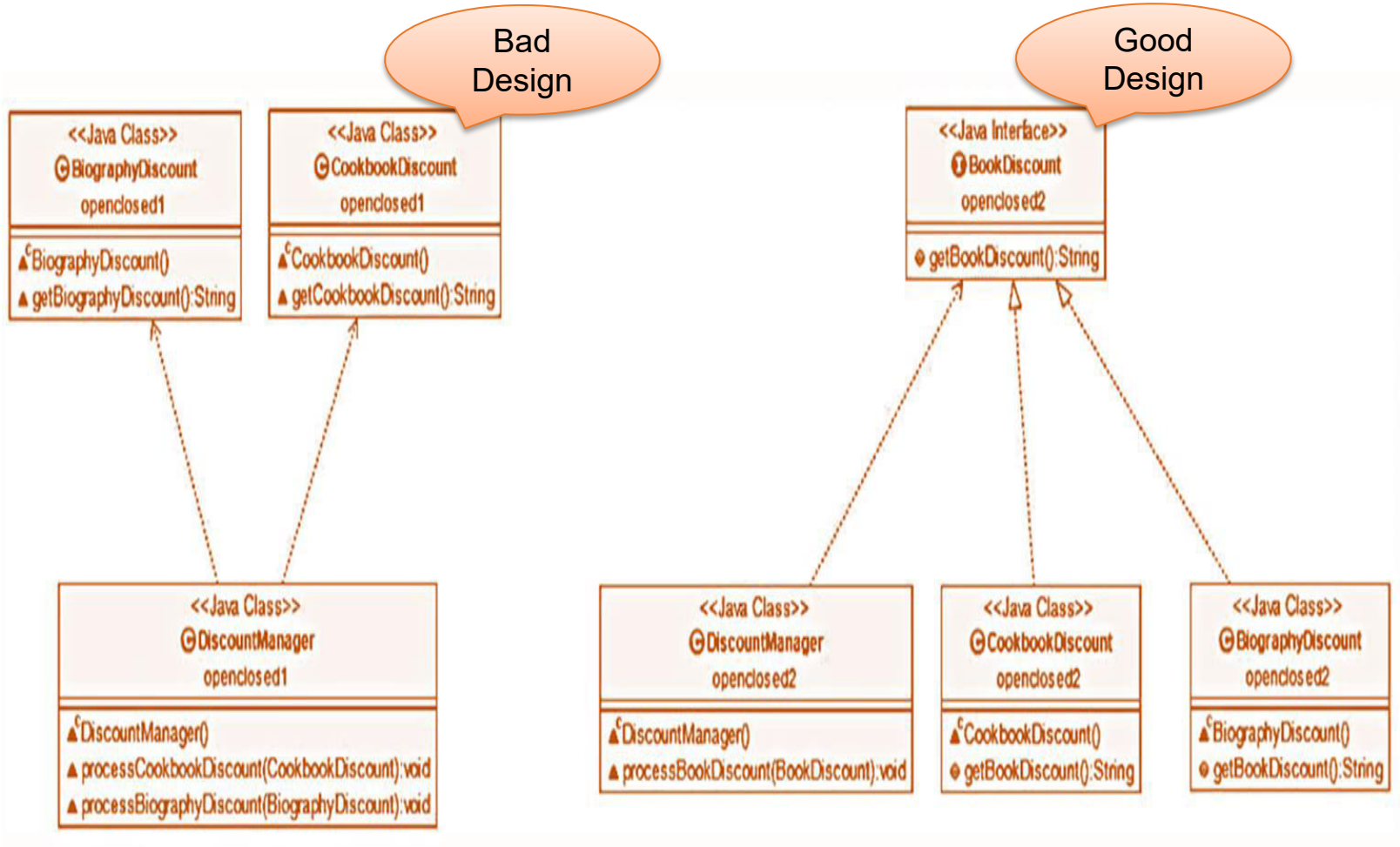
```
class CookbookDiscount implements
BookDiscount {
    @Override
    public String getBookDiscount() {
        String discount = "30% between Dec 1
and 24";
        return discount;
    }
}
class BiographyDiscount implements
BookDiscount {
    @Override
    public String getBookDiscount() {
        String discount = "50% on the
subject's birthday";
        return discount;
    }
}
```

Solution continued

Now, DiscountManager can refer to the BookDiscount interface instead of the concrete classes. When the processBookDiscount() method is called, pass both CookbookDiscount and BiographyDiscount as an argument, as both are the implementation of the BookDiscount interface.

```
class DiscountManager
{
    void processBookDiscount(BookDiscount discount) {...}
}
```

UML Class Diagram



Why OCP?

- If not followed

❑ End up testing the entire functionality

❑ QA Team needs to test the entire flow

❑ Costly process for the organization

❑ Breaks the SRP as well

❑ Maintenance overhead increase on the classes

Object Oriented Analysis and Design with Java

Implementation

Refer to Employee Program in drive

References

- Introduction To SOLID Principles (c-sharpcorner.com)
- A Solid Guide to SOLID Principles | Baeldung
- SOLID Principle in Programming: Understand With Real Life Examples - GeeksforGeeks

THANK YOU

Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

Object Oriented Analysis and Design using Java
UE21CS352

Prof. Sindhu R Pai

Teaching assistants : Nishanth M S

Vinay Padegal

Muskan Bansal

Department of Computer Science and Engineering

UE21CS352: Object Oriented Analysis and Design using Java

OO Design Principles and Sample Implementation of Patterns in Java

SOLID: Liskov Substitution Principle (LSP) & Interface Segregation Principle (ISP)

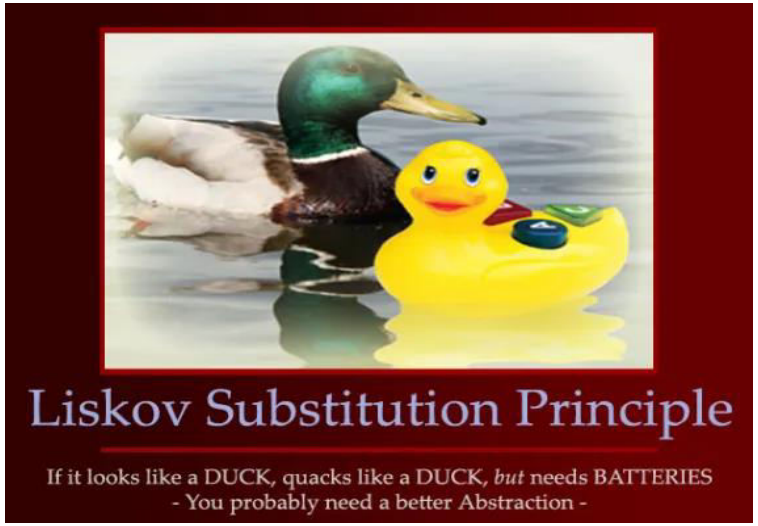
Prof. Sindhu R Pai

Department of Computer Science and Engineering

Object Oriented Analysis and Design with Java

Liskov Substitution Principle (LSP)

- Introduced by Barbara Liskov in 1987
- **"Let S be a subtype of T, then for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2".**
- **Derived types must be completely substitutable for their base types**
- An object of a superclass should be replaceable by objects of its subclasses without causing issues in the application.
- A child class should never change the characteristics of its parent class
- Derived classes should never do less than their base class.
- Applies to inheritance(is – a relationship) hierarchies and avoids overuse/misuse



Object Oriented Analysis and Design with Java

Example code for violation of LSP

- Consider the same book store example. The book store asks us to add a new delivery functionality to the application. The store also sells fancy hardcovers. It wants to deliver that to their high street shops
- Create a BookDelivery class that informs customers about the number of locations where they can collect their order. Create a new HardcoverDelivery subclass that extends BookDelivery and overrides the getDeliveryLocations() method with its own functionality

```
class BookDelivery {
    String titles;
    int userID;
    void getDeliveryLocations()
{...}
}
```

```
class HardcoverDelivery extends BookDelivery {
    @Override
    void getDeliveryLocations() {...}
}
```

Object Oriented Analysis and Design with Java

Example code for violation of LSP

- Later, the store asks us to create delivery functionalities for audiobooks.
- Now, we extend the existing BookDelivery class with an AudiobookDelivery subclass

```
class AudiobookDelivery extends BookDelivery {
    @Override
    void getDeliveryLocations() {/* can't be implemented */}
    // audiobooks can’t be delivered to physical locations.
}
```

- We have to change some characteristics of the getDeliveryLocations() method. That would violate the Liskov Substitution Principle.
- After the modification, we couldn’t replace the BookDelivery superclass with the AudiobookDelivery subclass without breaking the application

Solution

- To solve the problem, we need to fix the inheritance hierarchy.
- Let's introduce an extra layer of abstraction that better differentiates book delivery types.
- The new OfflineDelivery and OnlineDelivery classes split up the BookDelivery superclass.
- Move the getDeliveryLocations() method to OfflineDelivery and create a new getSoftwareOptions() method for the OnlineDelivery class (as this is more suitable for online deliveries).

Object Oriented Analysis and Design with Java

Good Design

```
class BookDelivery {

    String title;
    int userID;
}

class OfflineDelivery extends BookDelivery {

    void getDeliveryLocations() {...}
}

class OnlineDelivery extends BookDelivery {

    void getSoftwareOptions() {...}
}
```

Good Design

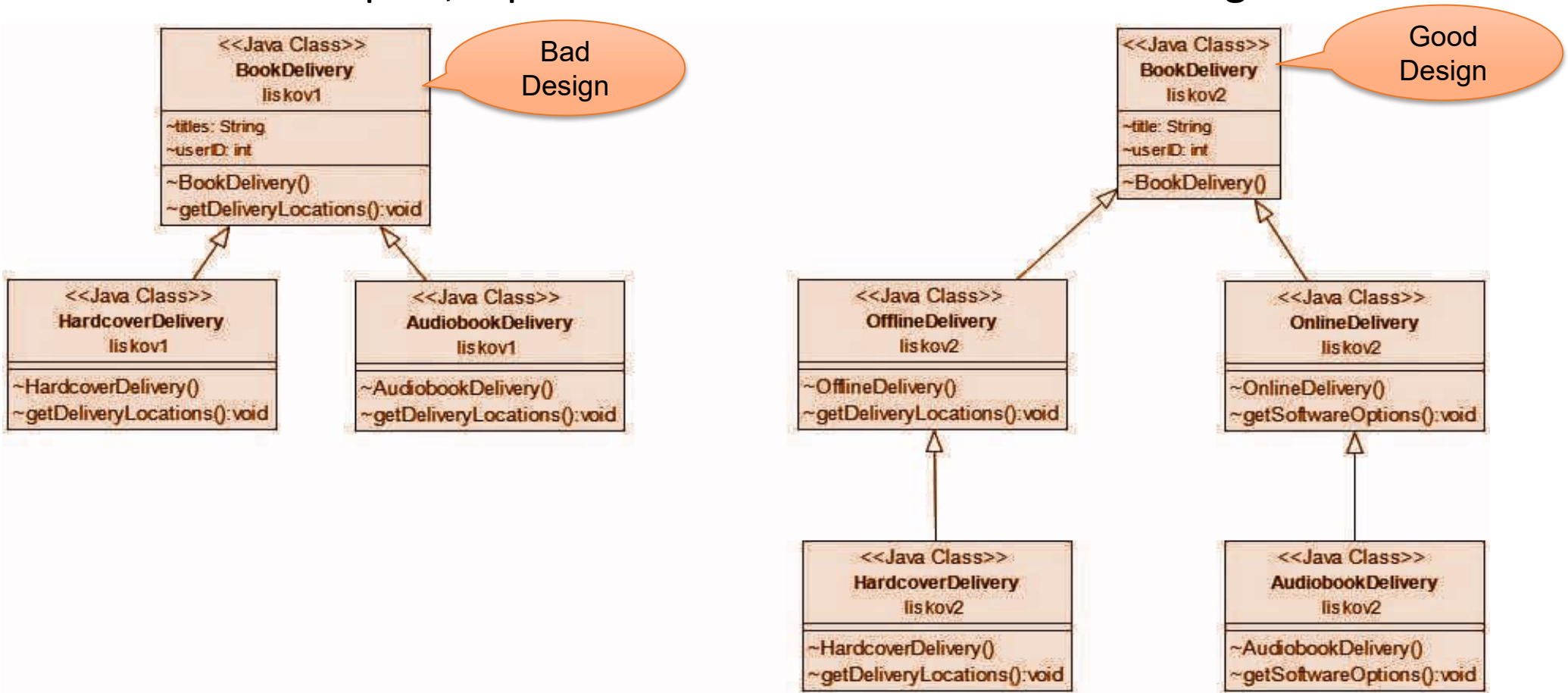
- In the refactored code, HardcoverDelivery will be the child class of OfflineDelivery and it will override the getDeliveryLocations() method with its own functionality.
- AudiobookDelivery will be the child class of OnlineDelivery, now it doesn't have to deal with the getDeliveryLocations() method.
- Instead, it can override the getSoftwareOptions() method of its parent with its own implementation (for instance, by listing and embedding available audio players).

```
class HardcoverDelivery extends OfflineDelivery {
   @Override
   void getDeliveryLocations() {...}
}

class AudiobookDelivery extends OnlineDelivery {
   @Override
   void getSoftwareOptions() {...}
}
```

Object Oriented Analysis and Design with Java

UML Class Diagram

- An extra layer is added to the inheritance hierarchy.
- New architecture is more complex, it provides us with a more flexible design.



Object Oriented Analysis and Design with Java

Implementation

Implementation of Employee object

References

- Liskov's Substitution Principle | SOLID Design Principles (ep 1 part 1) – YouTube
- SOLID Design Principles Explained: Interface Segregation with Code Examples (stackify.com)
- Interface Segregation Principle explained with example in Java – JavaBrahman
- Interface Segregation Principle in Java with Example (javaguides.net)
- (221) Interface Segregation Principle - YouTube

THANK YOU

Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

Object Oriented Analysis and Design using Java

Prof. Sindhu R Pai

Teaching assistants : Nishanth M S

Vinay Padegal

Muskan Bansal

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design using Java

UE21CS352B

OO Design Principles and Sample Implementation of Patterns in Java

SOLID: Interface Segregation Principle (ISP)

Prof. Sindhu R Pai

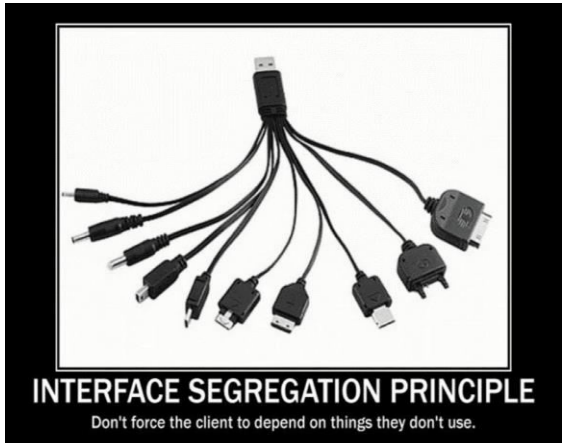
Department of Computer Science and Engineering

Object Oriented Analysis and Design using Java

Interface Segregation Principle (ISP)

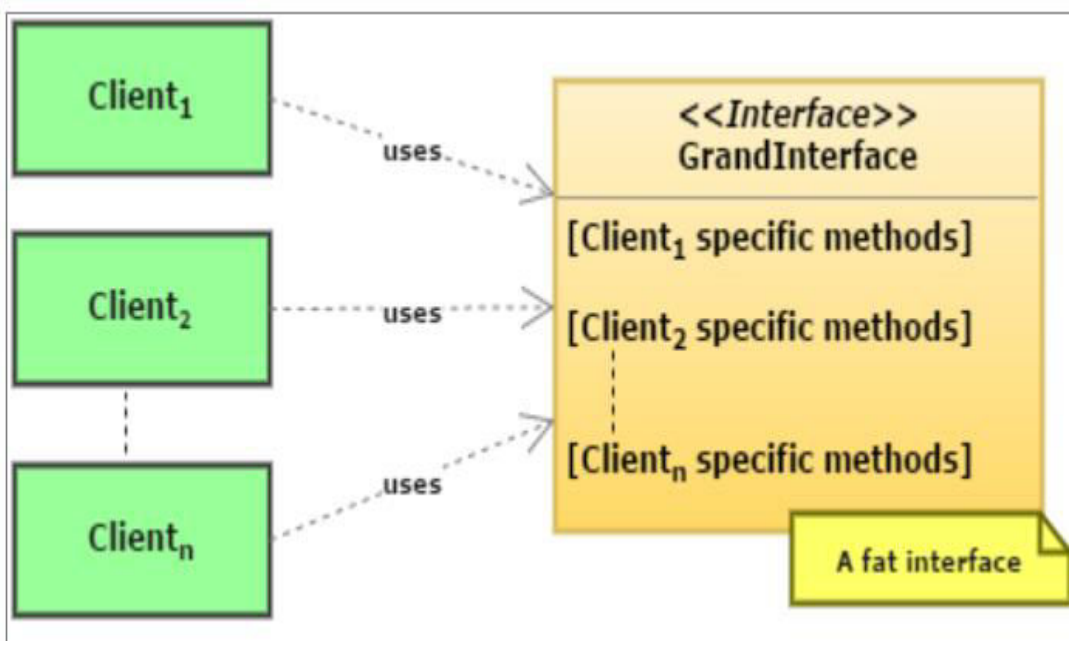
States that "Make fine grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not use".

- Break the fat interfaces which has too many methods.
- The principle states that many client-specific interfaces are better than one general-purpose interface.
- Clients should not be forced to implement a function they do not need.
- Instead, start by building a new interface and then let your class implement multiple interfaces as needed.
- Create smaller interfaces that you can implement more flexibly
- Failure to comply with this principle means that in our implementations we will have dependencies on methods that we do not need but that we are obliged to define.



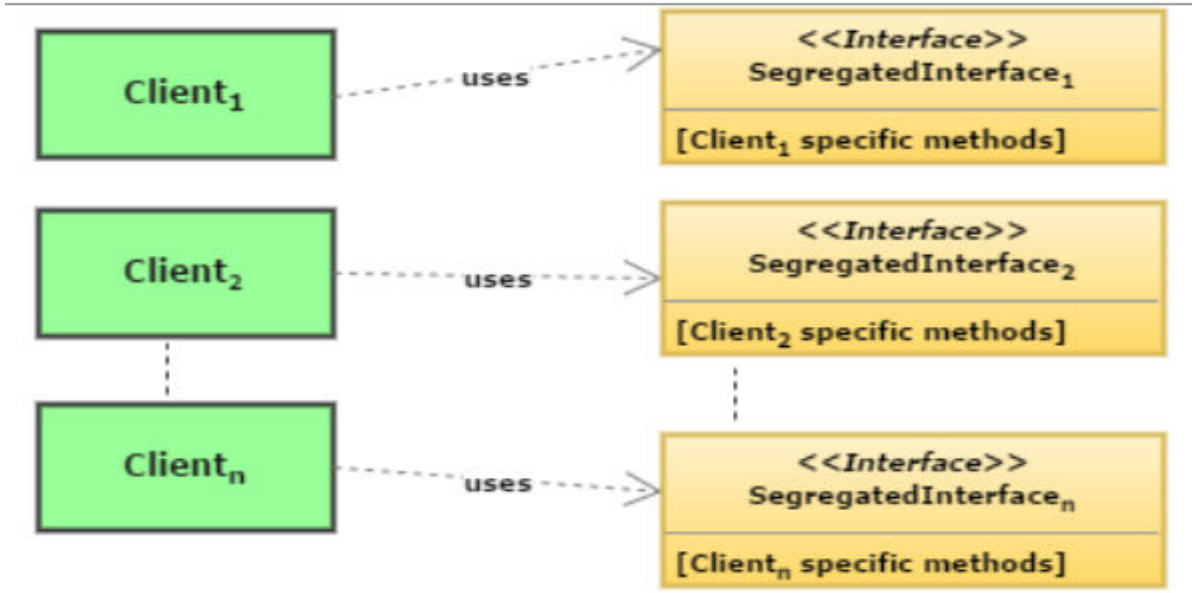
Object Oriented Analysis and Design using Java

Interface Segregation Principle (ISP)



Before Applying ISP: Fat interface

Note: Interface should not be bloated with methods that implementing classes don't require.



After Applying ISP: Lean interface/role interface

Object Oriented Analysis and Design using Java

Example code

```
interface IUser{
    boolean login(String name, String password);
    boolean register(String name, String password, String email);
    void logError(String error);
    boolean sendmail(String emailContent);
}
```

Code violating ISP

Code following ISP

```
interface IUser{
    boolean login(String name, String password);
    boolean register(String name, String password, String email);
}

interface ILogger{
    void logError(String error);
}

interface IMail{
    boolean sendmail(String emailContent);
}
```

Object Oriented Analysis and Design using Java

Example code for violation of ISP

- Adding some user actions to the online bookstore so that customers can interact with the content before making a purchase by creating an interface called BookAction with three methods:

```
public interface BookAction {
    void seeReviews();
    void searchSecondhand();
    void listenSample();
}
```

Object Oriented Analysis and Design using Java

Example code for violation of ISP continued

- Create two classes: HardcoverUI and an AudiobookUI that implement the BookAction interface with their own functionalities
- Both classes depend on methods they don't use, so it violates the Interface Segregation Principle

```
class HardcoverUI implements
BookAction {
    @Override
    public void seeReviews() {...}
    @Override
    public void searchSecondhand()
{...}
    @Override
    public void listenSample() {...}
}
```

```
class AudiobookUI implements
BookAction {
    @Override
    public void seeReviews() {...}
    @Override
    public void searchSecondhand()
{...}
    @Override
    public void listenSample() {...}
}
```

Object Oriented Analysis and Design using Java

Solution

- Extend the code with two more specific sub-interfaces: HardcoverAction and AudioAction.

```
public interface BookAction {
    void seeReviews();
}

public interface HardcoverAction extends
BookAction {
    void searchSecondhand();
}

public interface AudioAction extends
BookAction {
    void listenSample();
}
```

Object Oriented Analysis and Design using Java

Solution

- The HardcoverUI class can implement the HardcoverAction interface and the AudiobookUI class can implement the AudioAction interface.

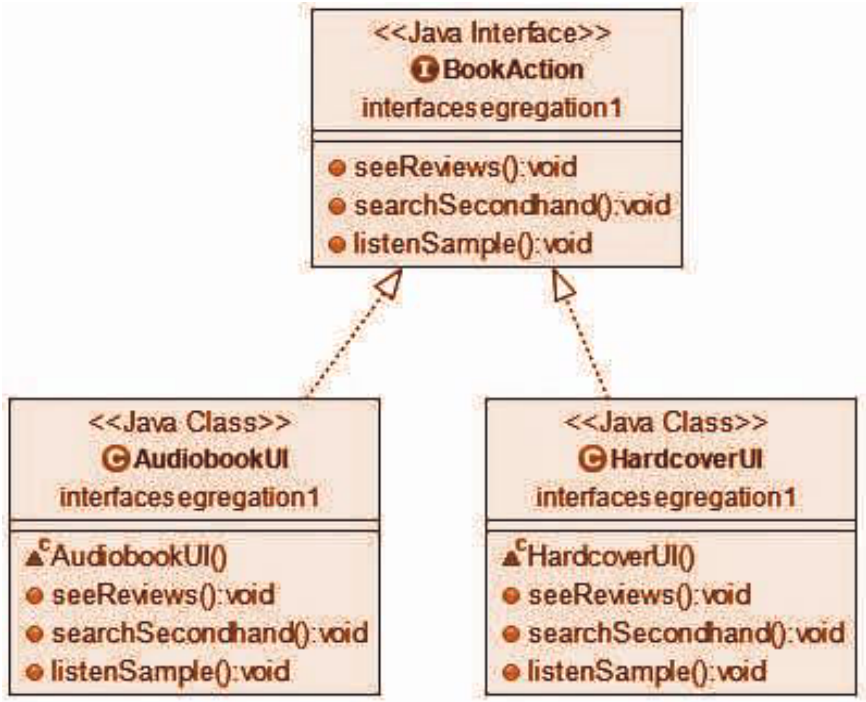
- Both the classes can implement the seeReviews() method of the BookAction super-interface.

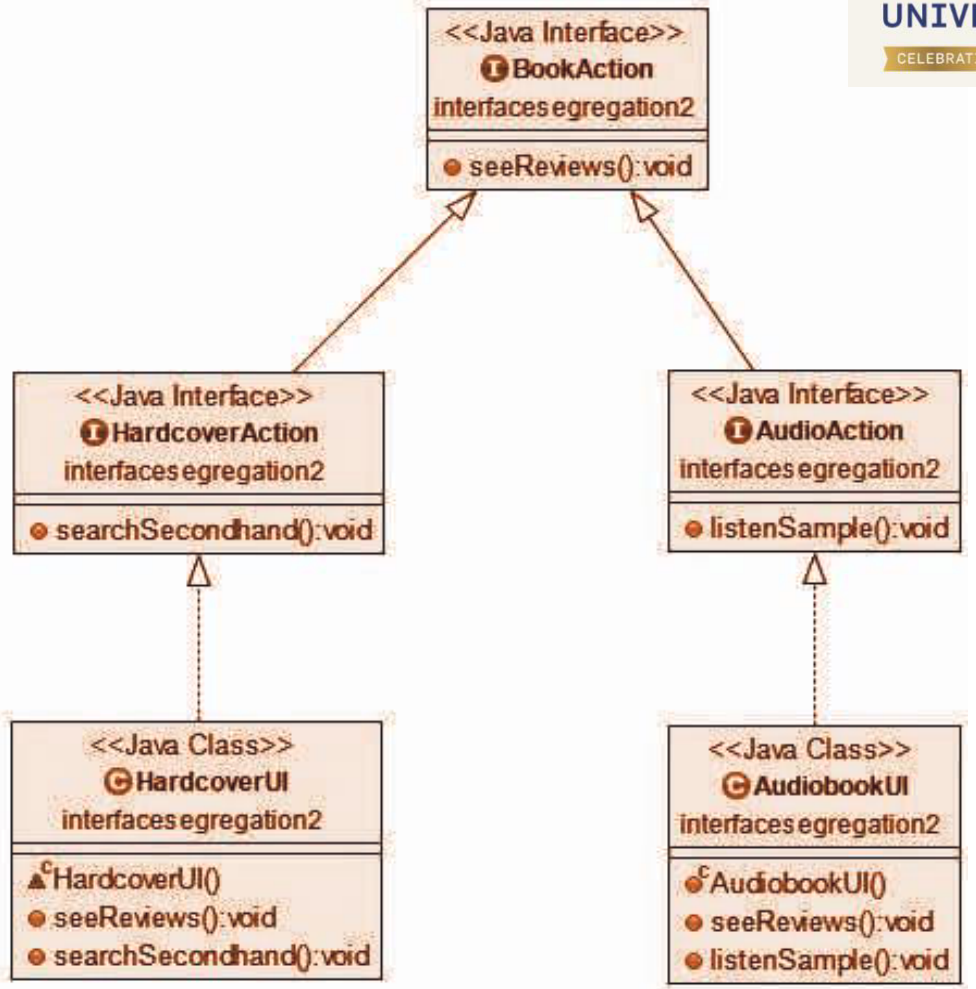
```
class HardcoverUI implements HardcoverAction
{
    @Override
    public void seeReviews() {...}
    @Override
    public void searchSecondhand() {...}
}
```

```
class AudiobookUI implements AudioAction
{
    @Override
    public void seeReviews() {...}
    @Override
    public void listenSample() {...}
}
```

Object Oriented Analysis and Design using Java

UML Class Diagram





Object Oriented Analysis and Design using Java

Case Study : Xerox

- First used and formulated by Robert C. Martin while consulting for Xerox
- Xerox had created a new printer system that could perform a variety of tasks such as stapling and faxing along with the regular printing task
- The software for this system was created from the ground up
- Modifications and Deployment to the system became more complex

Problem: One large Job class having all the responsibilities of the entire system

Solution: One large Job class is segregated to multiple interfaces depending on the requirement

Object Oriented Analysis and Design using Java

Implementation of ISP

Implementation of Xerox case study

Object Oriented Analysis and Design using Java

References

- SOLID Design Principles Explained: Interface Segregation with Code Examples (stackify.com)
- Interface Segregation Principle explained with example in Java – JavaBrahman
- Interface Segregation Principle in Java with Example (javaguides.net)
- (221) Interface Segregation Principle - YouTube

THANK YOU

Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

Object Oriented Analysis and Design using Java

Prof. Sindhu R Pai

Teaching assistants : Nishanth M S

Vinay Padegal

Muskan Bansal

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design using Java

OO Design Principles and Sample Implementation of Patterns in Java

UE21CS352B

Dependency Inversion Principle (DIP)

Prof. Sindhu R Pai
Department of Computer Science and Engineering

Object Oriented Analysis and Design using Java
Dependency Inversion Principle (DIP)

Program to the interface, not the implementation

- **High level modules should not depend on low level modules, both should depend upon abstractions.**
- **Abstractions should not depend upon details , details should depend upon abstractions.**
- The goal is to avoid tightly coupled code, as it easily breaks the application.
- Decouple high-level and low-level classes. The interaction between them must be thought of as an **abstract interaction**
- According to Robert C Martin, Dependency Inversion Principle is a **specific combination of the Open-Closed and Liskov Substitution Principles.**

Object Oriented Analysis and Design using Java

Example code violates DIP

- Suppose a book store asked us to build a new feature that enables customers to put their favorite books on a shelf.
- To implement this functionality, we create a lower-level Book class and a higher-level Shelf class. The Book class will allow users to see reviews and read a sample of each book they store on their shelves. The Shelf class will let them add a book to their shelf and customize the shelf.

```
class Book {
    void seeReviews() {...}
    void readSample() {...}
}
```

```
class Shelf {
    Book book;
    void addBook(Book book) {...}
    void customizeShelf() {...}
}
```

Object Oriented Analysis and Design using Java

Example code violates DIP

- When the store asks us to enable customers to add DVDs to their shelves,

First create DVD.

```
class DVD {
    void seeReviews() {...}
    void watchSample() {...}
}
```

Now, we should modify the Shelf class so that it can accept DVDs, too.

- However, this would clearly break the Open-Closed Principle.

The solution is to create an abstraction layer for the lower-level classes

(Book and DVD).

Object Oriented Analysis and Design using Java

Solution

- Introduce the Product interface and both classes will implement.

```
public interface Product {
    void seeReviews();
    void getSample();
}
```

```
class Book implements Product {
    @Override
    public void seeReviews() {...}
    @Override
    public void getSample() {...}

}
```

```
class DVD implements Product {
    @Override
    public void seeReviews() {...}
    @Override
    public void getSample() {...}

}
```

Solution

Now, Shelf can reference the Product interface instead of its implementations (Book and DVD).

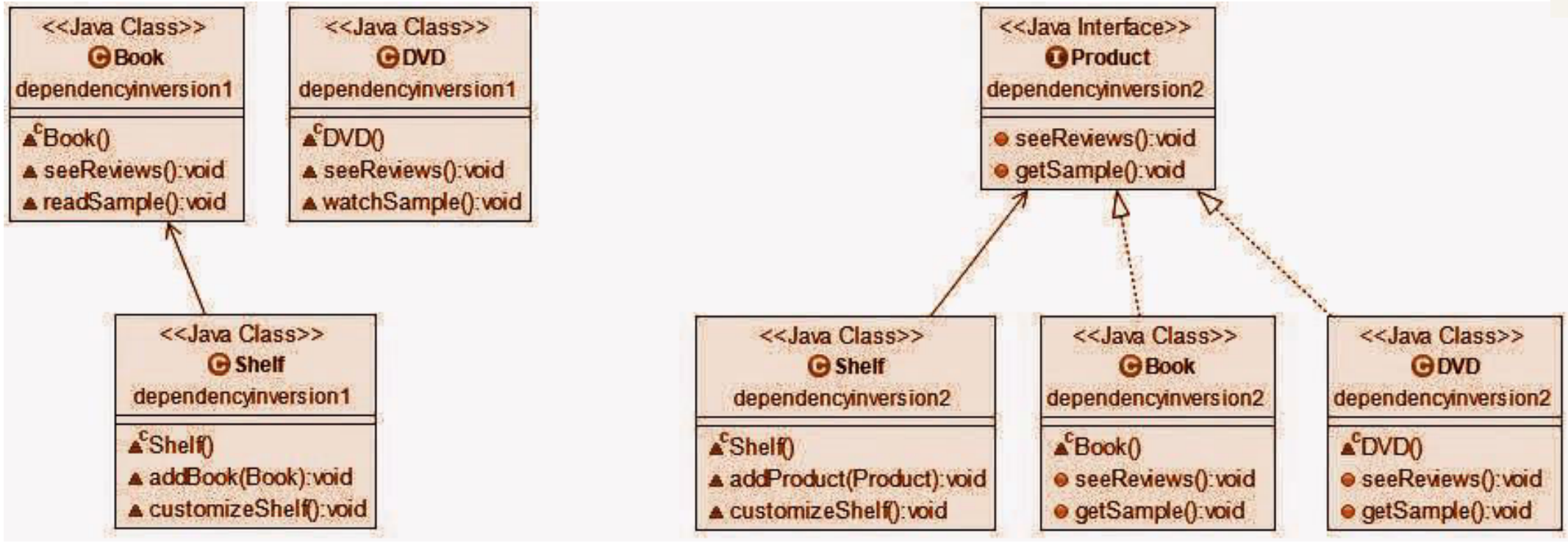
The refactored code also allows us to later introduce new product types (for instance, Magazine) that customers can put on their shelves, too.

```
class Shelf {
    Product get Sample;
    void addProduct(Product product) {...}
    void customizeShelf() {...}
}
```

Note: Code follows the **Liskov Substitution Principle**, as the Product type can be substituted with both of its subtypes (Book and DVD) without breaking the program. At the same time, we have also implemented the **Dependency Inversion** Principle, as high-level classes don't depend on low-level classes

Object Oriented Analysis and Design using Java

UML Class Diagram



Object Oriented Analysis and Design using Java

Intention of Usage – DIP and it's Implementation

Implementation

THANK YOU

J.Ruby Dinakar and Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

Object Oriented Analysis and Design using Java

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design using Java

UE21CS352B

Selection or Usage of a Design Patterns

Object Oriented Analysis and Design using Java

Agenda

- Recap of Design Pattern
- Types of Design Pattern
- Format and section of Design patterns
- Selection or Usage of a design Pattern

Object Oriented Analysis and Design using Java

Design pattern-Recap

” Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers. “

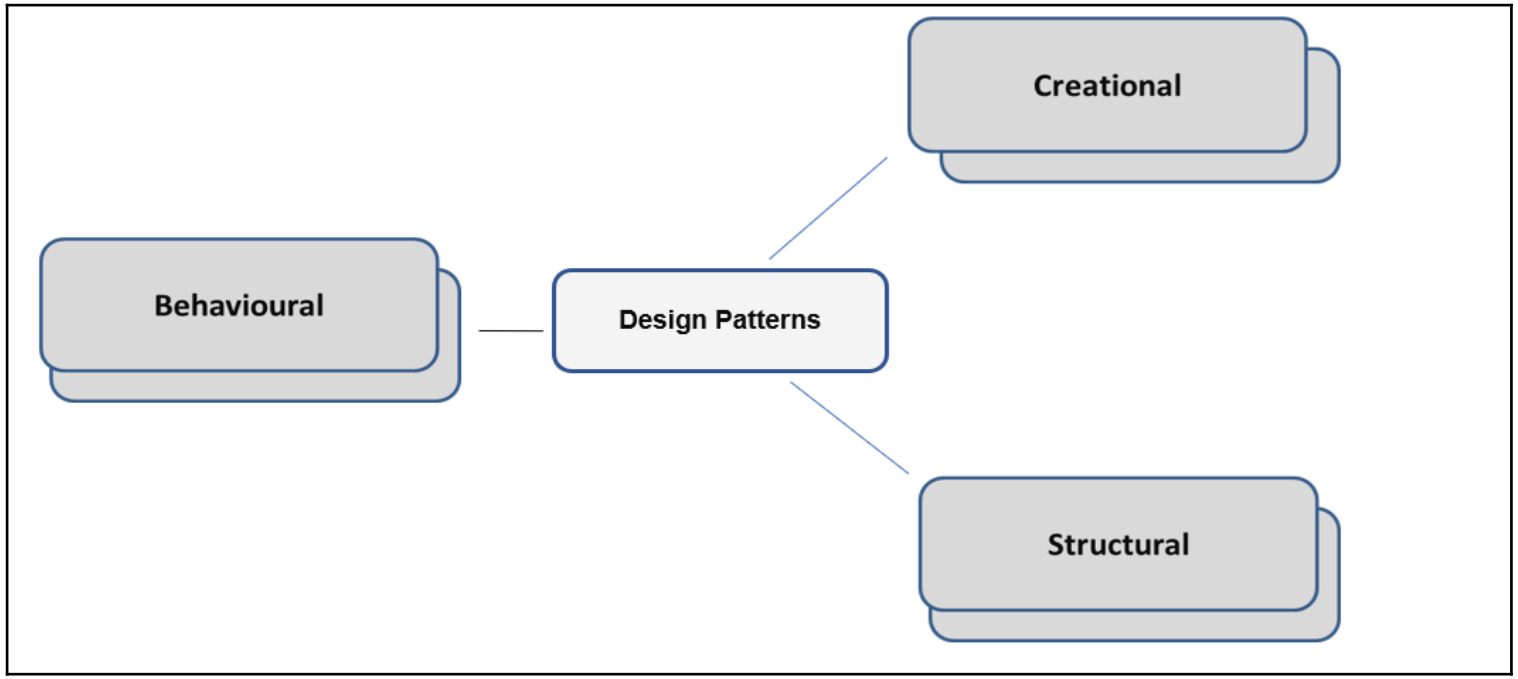


Fig 1: Categories of Design Patterns

Object Oriented Analysis and Design using Java

Principles of Design Pattern

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF).** According to these authors design patterns are primarily based on the following principles of object oriented design.

- **Program to an interface not an implementation**
- **Favor object composition over inheritance**

Uses of Design Pattern in Software Development

Design Patterns have two main usages in software development.

- **Common platform for developers**

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

- **Best Practices**

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps un-experienced developers to learn software design in an easy and faster way.

Object Oriented Analysis and Design using Java

Why use Design Patterns?

Design Objectives

- Good Design (the “ilities”)

- High readability and maintainability
- High extensibility
- High scalability
- High testability
- High reusability

Elements of a Design Pattern

A pattern has four essential elements (GoF)

- **Name**
  - Describes the pattern
  - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
- **Problem**
  - Describes when to apply the pattern
  - Answers - What is the pattern trying to solve?

Object Oriented Analysis and Design using Java

Elements of a Design Pattern (cont.)

- **Solution**
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
- **Consequences**
  - Results of applying the pattern
  - Benefits and Costs
  -Subjective depending on concrete scenarios

How to describe Design Patterns

This is critical because the information has to be conveyed to peer developers in order for them to be able to evaluate, select and utilize patterns.

- A format for design patterns
  - Pattern Name and Classification
  - Intent
  - Also Known As
  - Motivation
  - Applicability
  - Structure
  - Participants
  - Collaborations
  - Consequences
  - Implementation
  - Sample Code
  - Known Uses
  - Related Patterns

These contents help in selection and usage of a design pattern

Object Oriented Analysis and Design using Java

Organizing a Design Pattern space

-

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Fig 2: Organizing Design Patterns

9

- By *Purpose* (reflects what a pattern does):
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- By *Scope*: specifies whether the pattern applies primarily to
  - *classes* or to
  - *objects*.

Selection of a Design Pattern

As there are 23 patterns in the catalog it will be difficult to choose a design pattern. It might become hard to find the design pattern that solves our problem, especially if the catalog is new and unfamiliar to you. Below is a list of approaches we can use to choose the appropriate design pattern:

- **Consider how design patterns solve design problems**: Considering how design patterns help you find appropriate objects, determine object granularity, specify object interfaces and several other ways in which design patterns solve the problems will let you choose the appropriate design pattern.
- **Scan intent sections:** Looking at the intent section of each design pattern's specification lets us choose the appropriate design pattern.
- **Study how patterns interrelate**: The relationships between the patterns will direct us to choose the right patterns or group of patterns.

Selection of a Design Pattern

- **Study patterns of like purpose**: Each design pattern specification will conclude with a comparison of that pattern with other related patterns. This will give you an insight into the similarities and differences between patterns of like purpose.
- **Examine a cause of redesign:** Look at your problem and identify if there are any causes of redesign. Then look at the catalog of patterns that will help you avoid the causes of redesign
- **Consider what should be variable in your design**: Consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies.

Object Oriented Analysis and Design using Java

Usage of Design Pattern-How to use a selected Design Pattern

Below steps will show you how to use a design pattern after selecting one:

- **Read the pattern once through for a overview:**

Give importance to the applicability and consequences sections(in format) to ensure that the pattern is right for your problem.

- **Study the structure, participants and collaborations sections:**

Make sure to understand the classes and objects in the pattern and how they relate to one another.

- **Look at the sample code section to see a concrete example of the pattern in action:**

Studying the code helps us to implement the pattern.

Usage of a Design Pattern-How to use a Design Pattern

- **Define the classes:**

Declare their interfaces, inheritance relationships and instance variables, which represent the data and object references. Identify the affected classes by the pattern and modify them accordingly.

- **Define application-specific names for the operations in the pattern:**

Use the responsibilities and collaborations as a guide to name the operations. Be consistent with the naming conventions.

- **Implement the operations to carry out the responsibilities and collaborations in the pattern:**

The implementation section provides hints to guide us in the implementation. The examples in the sample code section can also help as well.

References

- https://sourcemaking.com/design_patterns/
- https://refactoring.guru/design-patterns/java
- https://www.startertutorials.com/patterns/select-design-pattern.html
- Design Patterns: Elements of Reusable Object-Oriented SoftwareErich Gamma, Ralph Johnson, Richard Helm · 1995

THANK YOU

Dr. Sudeepa Roy Dey and Dr Geetha D

Department of Computer Science and Engineering

Object Oriented Analysis and Design Using Java

UE21CS352B

Dr. J. Mannar Mannan

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B: Object Oriented Analysis and Design Using Java

Unit-3
Introduction To Design Patterns

Agenda

- The Beginning of Design Patterns
- Design Challenges
- What is a design pattern?
- What is the advantage of knowing/using design patterns?

The Beginning of Patterns

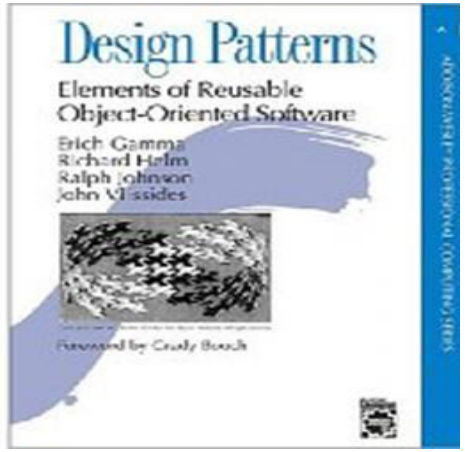
Christopher Alexander, architect

- A Pattern Language--Towns, Buildings, Construction
- Timeless Way of Building (1979)
- "Each pattern describes a *problem* which occurs over and over again in our environment, and then describes the core of the *solution* to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

"Gang of Four" (GoF) Book

Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994 Written by this "gang of four"

- Dr. Erich Gamma, then Software Engineer, Taligent, Inc.
- Dr. Richard Helm, then Senior Technology Consultant, DMR Group
- Dr. Ralph Johnson, then and now at University of Illinois, Computer Science Department
- Dr. John Vlissides, then a researcher at IBM



"Gang of Four" (GoF) Book

Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley

This book defined 23 patterns in three categories

- Creational patterns deal with the process of object creation
- Structural patterns, deal primarily with the static composition and structure of classes and objects
- Behavioral patterns, which deal primarily with dynamic interaction among classes and objects

Design challenges

- Designing software for reuse is hard. One must find:

i. a good problem decomposition, and the right software

ii. a design with flexibility, modularity and elegance

- Designs often emerge from trial and error
- Successful designs do exist

i. two designs they are almost never identical

ii. they exhibit some recurring characteristics

Can designs be described, codified or standardized?

- this would short circuit the trial and error phase
- produce "better" software faster

Architecture vs Design Patterns

Architecture

- High-level framework for structuring an application
  - "client-server based on remote procedure calls"
  - "abstraction layering"
  - "distributed object-oriented system based on CORBA"
- Defines the system in terms of computational components & their interactions

Design Patterns

- Lower level than architectures (Sometimes, called *micro-architecture*)
- Reusable collaborations that solve subproblems within an application
  - how can I decouple subsystem X from subsystem Y?

Why Design Patterns?

- Design patterns support *object-oriented reuse* at a high level of abstraction
- Design patterns provide a "framework" that guides and constrains object-oriented implementation

What are Design Patterns?

Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

- Reusable solutions to the problems: In software engineering, a design pattern is general reusable solution to commonly occurring problem in software design.
- Interaction between the objects :Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying final application classes or objects that are involved.
- Template, not a solution :It is not a finished design that can be transferred directly into code.

- Language independent

Pattern has four Essential Elements

.

- Pattern Name- is a handle we can use to describe a design problem, its solutions, and consequences in a word or two
- The Problem – describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
- The Solution - describes the elements that make up the design, their relationships, responsibilities, and collaborations.
- Consequences - are the results and trade-offs of applying the pattern.

- **Applicability -** What are the situations in which the design pattern can be applied?
- **Structure -** A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT)
- **Participants -** The classes and/or objects participating in the design pattern and their responsibilities.
- **Collaborations -** How the participants collaborate to carry out their responsibilities.
- **Consequences -** How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?
- **Implementation -** What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?
- **Related Pattern -** What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Pros of Design Patterns

Pros

- Add consistency to designs by solving similar problems the same way, independent of language
- Add clarity to design and design communication by enabling a common vocabulary
- Improve time to solution by providing templates which serve as foundations for good design
- Improve reuse through composition

Cons of Design Patterns

Cons

- Some patterns come with negative consequences (i.e. object proliferation, performance hits, additional layers)
- Consequences are subjective depending on concrete scenarios
- Patterns are subject to different interpretations, misinterpretations, and philosophies
- Patterns can be overused and abused--□ Anti-Patterns

References

- https://sourcemaking.com/design_patterns/
- https://refactoring.guru/design-patterns/java
- Design Patterns: Elements of Reusable Object-Oriented SoftwareErich Gamma, Ralph Johnson, Richard Helm · 1995

THANK YOU

Dr. Sudeepa Roy Dey and Dr Geetha D

Department of Computer Science and Engineering

Object Oriented Analysis and Design using Java

UE21CS352B

Prof. K V N M Ramesh
Department of Computer Science and Engineering

UE21CS352B: Object Oriented Analysis and Design using Java

OO Design Patterns

Prof. K V N M Ramesh

Department of Computer Science and Engineering

Agenda

- ❑ Introduction to creational design patterns
- ❑ Types of creational Design Patterns.
- ❑ Singleton-definition
  - ✔ Motivation
  - ✔ Intent
  - ✔ Problem(use case)
  - ✔ Solution (without singleton and with singleton)
  - ✔ Implementation
  - ✔ Applicability
  - ✔ Consequence

Introduction to Creational Design Pattern

- Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.
- The basic form of object creation could result in design problems or added complexity to the design.
- Creational design patterns solve this problem by somehow controlling this object creation.

Recurring themes:

- Encapsulate knowledge about which concrete classes the system uses (so we can change them easily later)
- Hide how instances of these classes are created and put together (so we can change it easily later)

Introduction to Creational Design Pattern

Everyone knows an object is created by using *new* keyword in java.
For example:

```
StudentRecord s1=new StudentRecord();
```

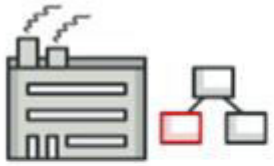
The *new* operator is often considered harmful as it scatters objects all over the application. Over time it can become challenging to change an implementation because classes become tightly coupled.

Hard-Coded code is not a good programming approach. Here, we are creating the instance by using the new keyword. Sometimes, the nature of the object must be changed according to the nature of the program. In such cases, we must get the help of creational design patterns to provide more general and flexible approach.

Types of Creational Design pattern

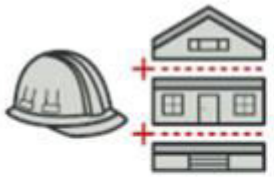
Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.



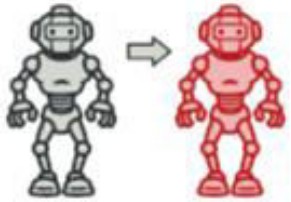
Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

Lets you copy existing objects without making your code dependent on their classes.

Composition over inheritance: "Favor 'object composition' over 'class inheritance'." (Gang of Four 1995:20)

Types of Creational Design pattern: Scope

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Singleton: Class, Object Structural

Motivation

The Singleton is part of the Creational Design Pattern Family .Singleton Design Pattern aims to keep a check on initialization of objects of a particular class by **ensuring that only one instance of the object exists throughout the Java Virtual Machine.**

A Singleton class also provides one unique global access point to the object so that each subsequent call to the access point returns only that particular object.

A real-life example will be, **Imagine you work for a big company that has cloud storage system for storing shared resources of files, images and documents we create shared storage as creating separate cloud storage for every user may be costly**.

Definition:

The singleton pattern is a design pattern that restricts the instantiation of a class to one object.

Why implement Singleton Design Pattern?

1.We can be sure that a class has only a single instance.

2.We gain a global access point to that instance.

3.The Singleton Object is initialized only when it's requested for the first time.

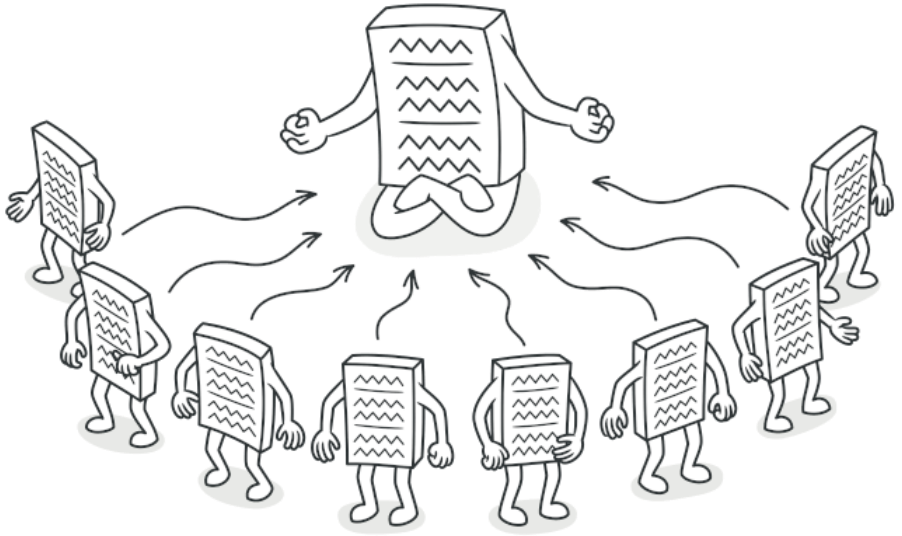
Singleton: Class, Object Structural

Intent

There are only two points in the definition of a singleton design pattern,

1) There should be only one instance allowed for a class and
2) We should allow global point of access to that single instance.

Ensure a class only has one instance, and provide a global point of access to it.



Real-World Analogy

The government is an excellent example of the Singleton pattern.

A country can have only one official government.

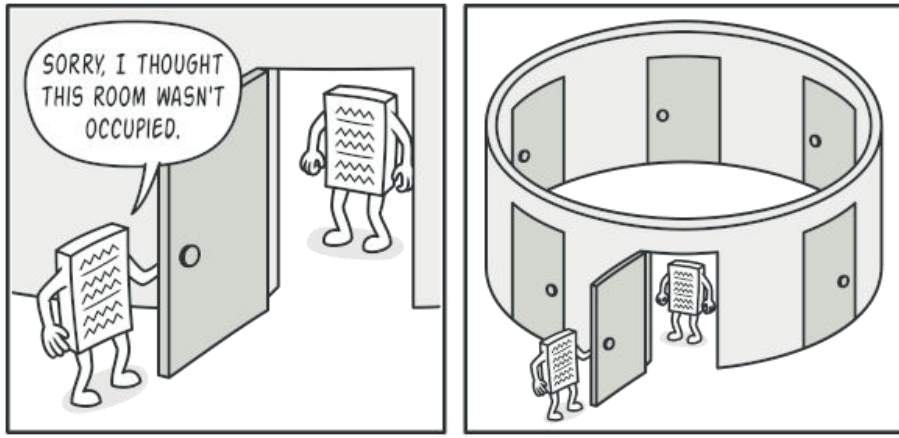
Regardless of the personal identities of the individuals who form governments, the title, “The Government of X”, is a global point of access that identifies the group of people in charge.

Singleton: Class, Object Structural

Problem

The Singleton pattern solves two problems at the same time, violating the Single Responsibility Principle:

1. Ensure that a class has just a single instance. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.



Clients may not even realize that they're working with the same object all the time.

Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.

Note that this behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design.

Singleton: Class, Object Structural

Problem

2.Provide a global access point to that instance. Remember those global variables that you (all right, me) used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

There's another side to this problem: you don't want the code that solves problem #1 to be scattered all over your program. It's much better to have it within one class, especially if the rest of your code already depends on it.

Singleton: Class, Object Structural

Solution

All implementations of the Singleton have these two steps in common:

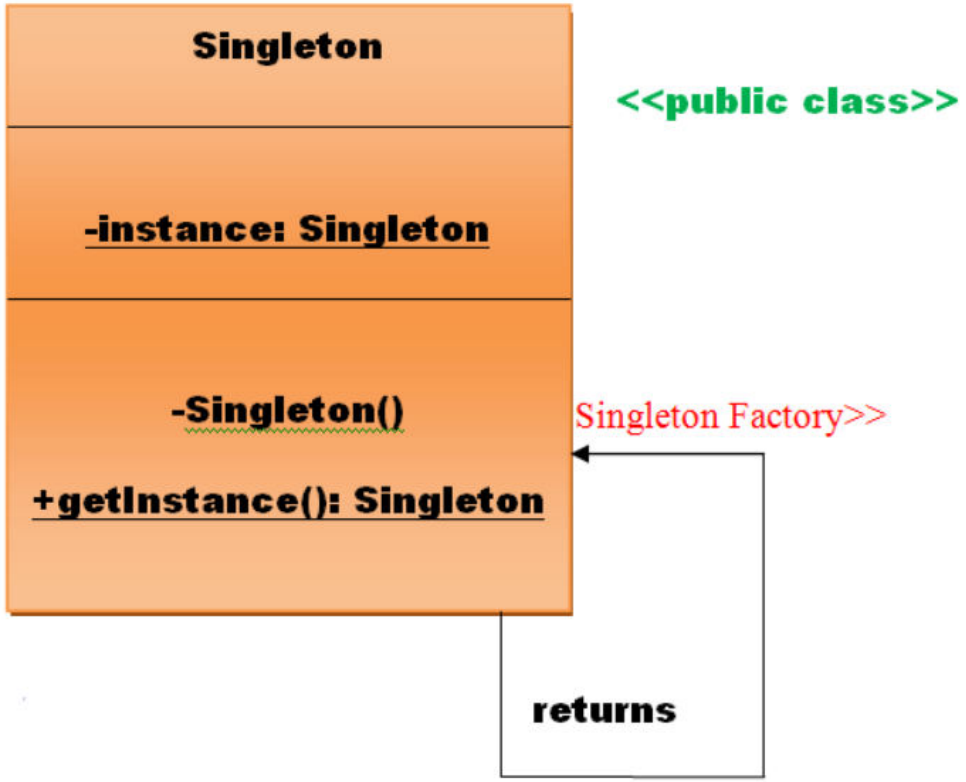
- Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
- Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

Object Oriented Analysis and Design Using Java

Singleton: Implementation

UML class diagram for the Singleton Pattern



Singleton pattern is used for logging, drivers objects, caching, and thread pool.

Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade, etc.

Singleton design pattern is used in core Java classes also (for example, java.lang.Runtime, java.awt.Desktop).

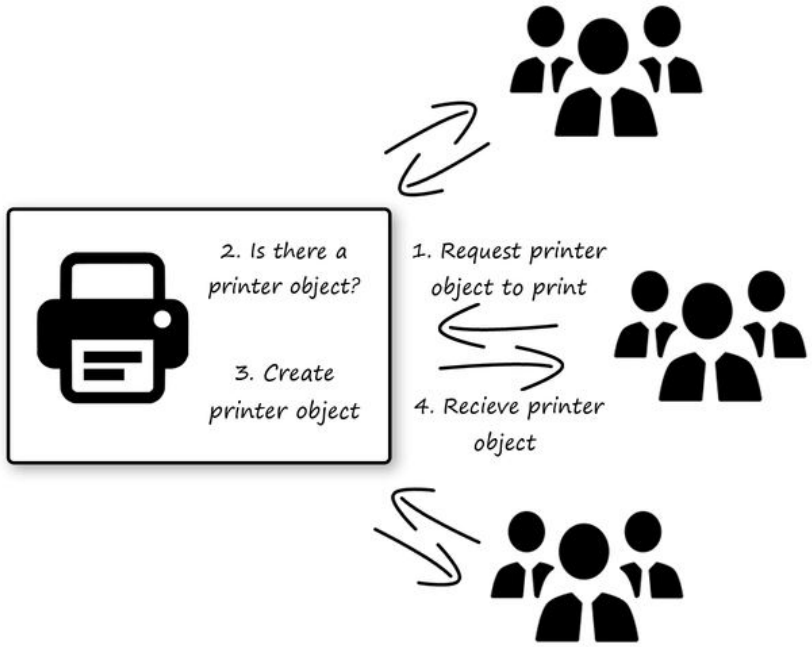
Singleton: Implementation example

❑ **Problem Statement**:

How can a shared office space with many employees, share the same printer. This makes sense, since it is often the reality for corporations and companies with employees sharing office facilities.

Design Solution

- The concept of the singleton design pattern is based around a private constructor and a public static initialization method.
- The private constructor ensures that the class can only be initialized by itself, which makes the public static initialization method the only way of getting an instance of the class.



1. Only one instance of the object available to the whole system.
2. No additional arguments used to distribute the object around in the system.

Design Solution

- However, we do not want the method to start a new instance for each time it is called. Therefore, we save the instance of the class in a static field inside its own class.

- Now that we have an instance of the class ready to work with inside the class, we null check on it when we call the static initialization method. If the class has not previously been initialized, then we initialize it, otherwise we return our static field instance.

- This means that there will always only be one instance of the class available, hence the name *singleton* pattern.

Class Diagram (UML)

- In terms of class diagram, the singleton pattern is not worth talking about. As you might have figured out by now, the singleton pattern is based around two methods inside the same class, which makes the class diagram simple to understand.

- Thereby said, that if you understand the concept of the singleton pattern from the previous section, you should be ready to dive into the code.

Code (Java)

Printer

```
public class Printer {
    private static Printer printer;
    private int nrOfPages;
    private Printer() {
    }
    public static Printer getInstance() {
        return printer == null ?
                printer = new Printer() :
                printer;
    }
    public void print(String text){
        System.out.println(text +
                "n" + "Pages printed today " + ++nrOfPages +
                "n" + "---------");
    }
}
```

object variable is created

Employee

```
public class Employee {
    private final String name;
    private final String role;
    private final String assignment;
    public Employee(String name, String role, String assignment)
{
        this.name = name;
        this.role = role;
        this.assignment = assignment;
    }
    public void printCurrentAssignment(){
        Printer printer = Printer.getInstance();
        printer.print("Employee: " + name + "n" +
                "Role: " + role + "n" +
                "Assignment: " + assignment + "n");
    }
}
```

instance of class is called

Code (Java)

How To Use The Singleton Pattern

```
public class Main {
    public static void main(String[] args) {

        Employee graham = new Employee("Graham",   "CEO", "Making excutive decisions");
        Employee sara = new Employee("Sara","Consultant",  "Consuting the company");
        Employee tim = new Employee("Tim","Salesmen", "Selling the company's products");
        Employee emma = new Employee("Emma", "Developer", "Developing the latest mobile app.");

        graham.printCurrentAssignment();
        sara.printCurrentAssignment();
        tim.printCurrentAssignment();
        emma.printCurrentAssignment();
    }
}
```

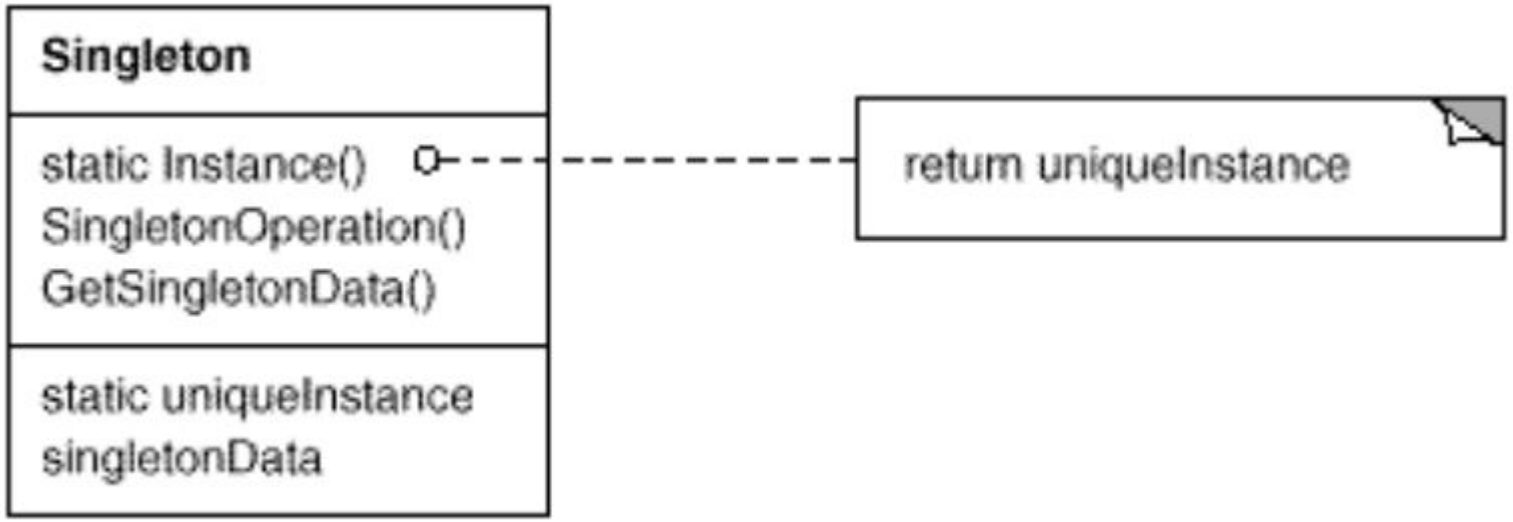
Applicability

Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Object Oriented Analysis and Design Using Java

Structure

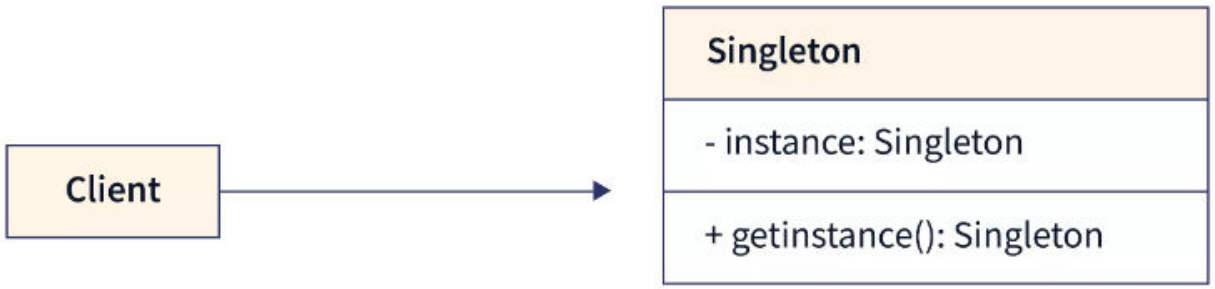


Participants

- Singleton design pattern has two core participants: Singleton and Client.

- defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method and a static member function).

- may be responsible for creating its own unique instance.



Collaborations

Clients access a Singleton instance solely through Singleton's Instance operation.

Consequence

The Singleton pattern has several benefits:

1. Controlled access to sole instance. Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

1. Reduced name space. The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.

1. Permits refinement of operations and representation. The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.

Consequence

The Singleton pattern has several benefits:

4. Permits a variable number of instances. The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change

5. More flexible than class operations. Another way to package a singleton's functionality is to use class operations (that is, static member functions or class methods). But both of these language techniques make it hard to change a design to allow more than one instance of a class. Moreover, static member functions in class are compile time, so subclasses can't override them polymorphically.

Implementation

Java Singleton Pattern Implementation

1.Add a private static field to the class for storing the singleton instance.
2.Declare a public static creation method for getting the singleton instance.
3.Implement “lazy initialization” inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4.Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
5.Go over the client code and replace all direct calls to the singleton’s constructor with calls to its static creation method.

Implementation

various design options for implementing Singleton:

Method 1: lazy instantiation

```
// Classical Java implementation of singleton design pattern
class Singleton
{
        private static Singleton obj;

        // private constructor to force use of
        // getInstance() to create Singleton object
        private Singleton() {}

        public static Singleton getInstance()
        {
                if (obj==null)
                obj = new Singleton();
                return obj;
        }
}
```

- Here we have declared getInstance() static so that we can call it without instantiating the class.
- The first time getInstance() is called it creates a new singleton object and after that it just returns the same object.
- Note that Singleton obj is not created until we need it and call getInstance() method. This is called lazy instantiation.

Implementation

The main problem with above method is that it is not thread safe. Consider the following execution sequence.

Thread one

```
public static Singleton getInstance(){
if(obj==null)

obj=new Singleton();
return obj;
}
```

Thread two

```
public static Singleton getInstance(){
if(obj==null)


obj=new Singleton();
return obj;
}
```

This execution sequence creates two objects for singleton. Therefore this classic implementation is not thread safe.

Implementation

Method 2: make getInstance() synchronized

```
// Thread Synchronized Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj;

    private Singleton() {}

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

- Here using synchronized makes that only one thread at a time can execute getInstance().
- The main disadvantage of this is method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program.
- However if performance of getInstance() is not critical for your application this method provides a clean and simple solution.

Implementation

Method 3: Eager Instantiation

```
// Static initializer based Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj = new
Singleton();

    private Singleton() {}

    public static Singleton getInstance()
    {
        return obj;
    }
}
```

- Here we have created instance of singleton in static initializer.
- JVM executes static initializer when the class is loaded and hence this is guaranteed to be thread safe.
- Use this method only when your singleton class is light and is used throughout the execution of your program.

Implementation

Method 4 (Best): Use "Double Checked Locking"

- By considering method 2,once an object is created , the synchronization is no longer useful because obj will not be null and any sequence of operations will lead to consistent results.
- So we will only acquire lock on the getInstance() once, when the obj is null.
- This way we only synchronize just what we want.

```
// Double Checked Locking based Java implementation of
// singleton design pattern
class Singleton
{
    private static volatile Singleton obj  = null;


    private Singleton() {}


    public static Singleton getInstance()
    {
        if (obj == null)
        {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj==null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

Implementation

- We have declared the obj volatile which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance.

- This method drastically reduces the overhead of calling the synchronized method every time.

Pros and Cons

PROS

1. You can be sure that a class has only a single instance.
2. You gain a global access point to that instance.
3. The singleton object is initialized only when it's requested for the first time.

CONS

1. Violates the Single Responsibility Principle. The pattern solves two problems at the time.
2. The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
3. The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
4. It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects.

References

Text Reference

- Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference

https://integu.net/singleton-pattern/
https://refactoring.guru/design-patterns/singleton
https://www.geeksforgeeks.org/singleton-design-pattern/
https://refactoring.guru/design-patterns/singleton/java/example
https://sourcemaking.com/design_patterns/singleton

THANK YOU

Prof K V N M Ramesh

Department of Computer Science and Engineering

kvnmramesh@pes.edu

Object Oriented Analysis and Design Using Java

UE21CS352B

Prof. Shilpa S
Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design using Java

OO Design Patterns-Factory

Prof. Shilpa S

Department of Computer Science and Engineering

Agenda

- Factory-definition
- Motivation
- Intent
- Implementation
- Applicability
- Structure-Consequence
- Issues

Introduction

The Factory Design Pattern or Factory Method Design Pattern is one of the most used design patterns in Java.

According to GoF, this pattern "**defines an interface for creating an object, but let subclasses decide which class to instantiate.** The Factory method lets a class defer instantiation to subclasses".

This pattern delegates the responsibility of initializing a class from the client to a particular factory class by creating a type of virtual constructor.

To achieve this, we rely on a factory which provides us with the objects, hiding the actual implementation details. The created objects are accessed using a common interface.

Also Known As Virtual Constructor

Factory: Class, Object Structural

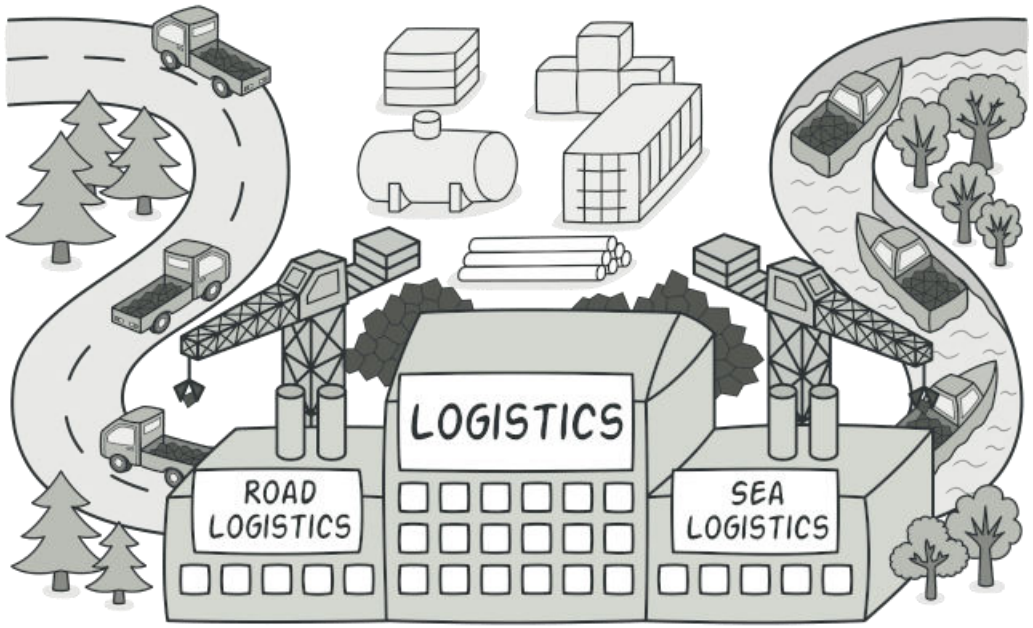
Motivation

- To deal with the problem of creating objects without having to specify the exact class of the object that will be created.
- Creating an object often requires complex processes not appropriate to include within a composing object.
- The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns.
- The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

Factory: Class, Object Structural

Intent

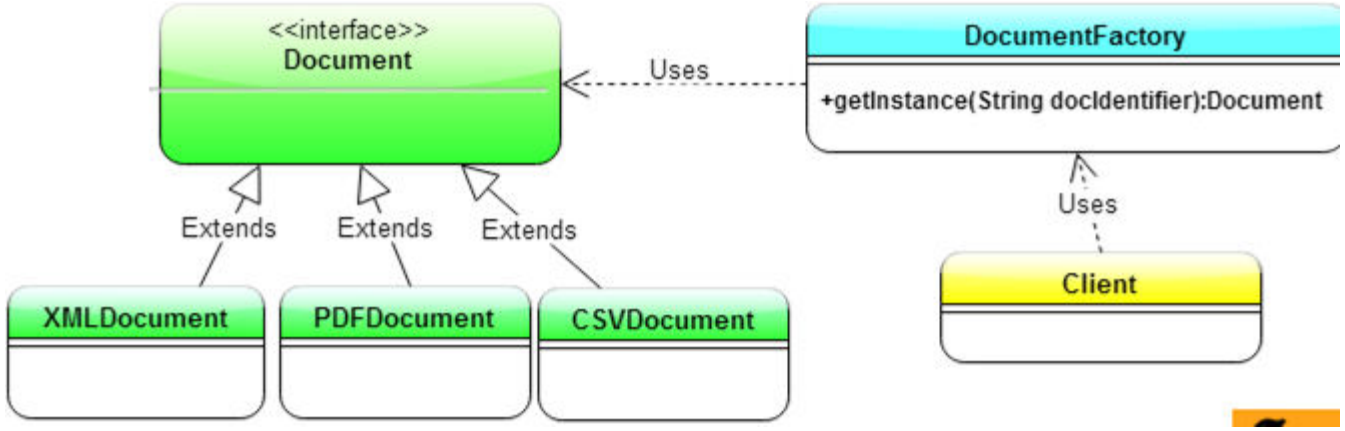
Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



Object Oriented Analysis and Design using Java

Factory: Implementation

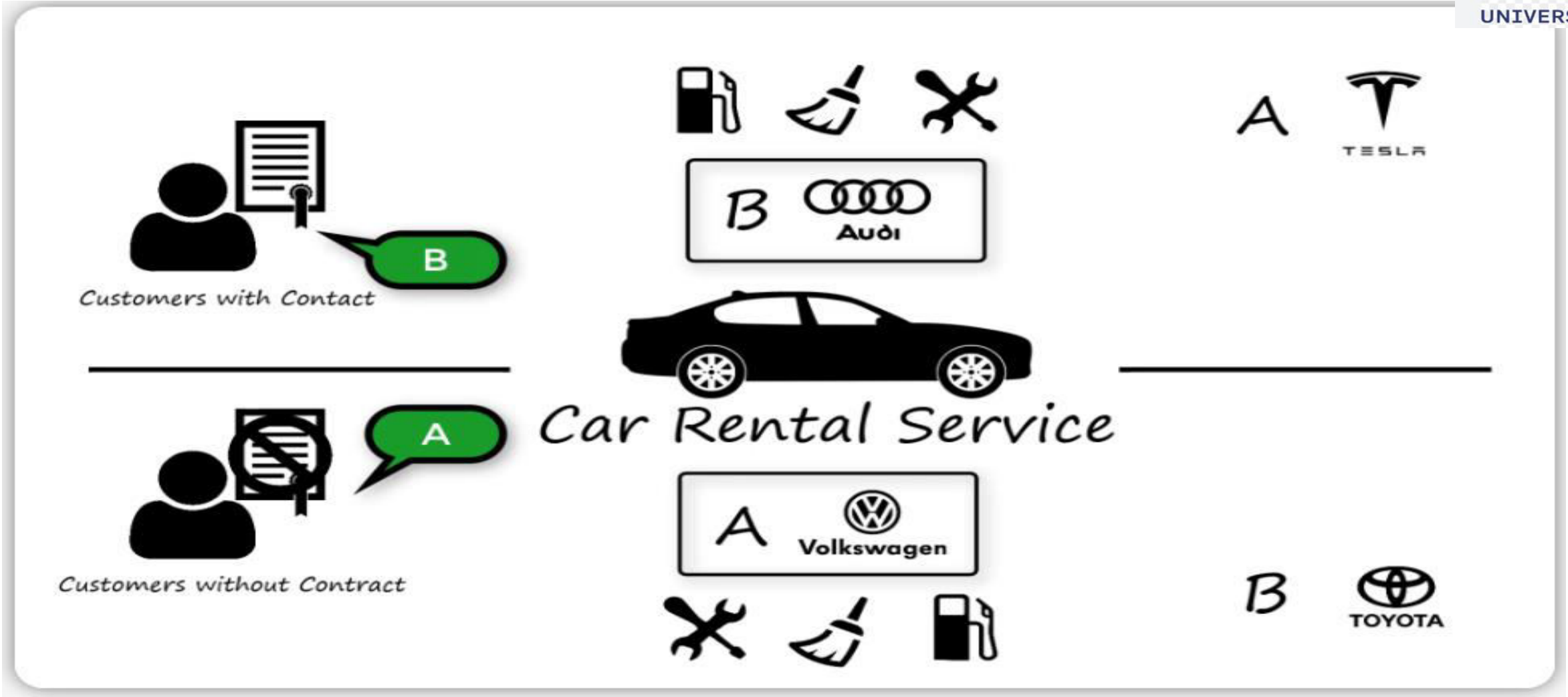
UML class diagram for the Factory Pattern



Object Oriented Analysis and Design using Java

Factory: Implementation example-1

Problem Statement:



Solution

Open Creation

As a beginning, we start by checking whether our customer has a company contract. If so, we retrieve their requested grade of car and runs through the options to see if we have a match.

For this scenario, we have already a Car super-class, which can be extended by all car brands sub-classes (Tesla, Audi, Volkswagen, and Toyota). This allows us to utilize the common methods of the Car super-class to perform a mechanical service check, clean up the car, and put fuel on it.

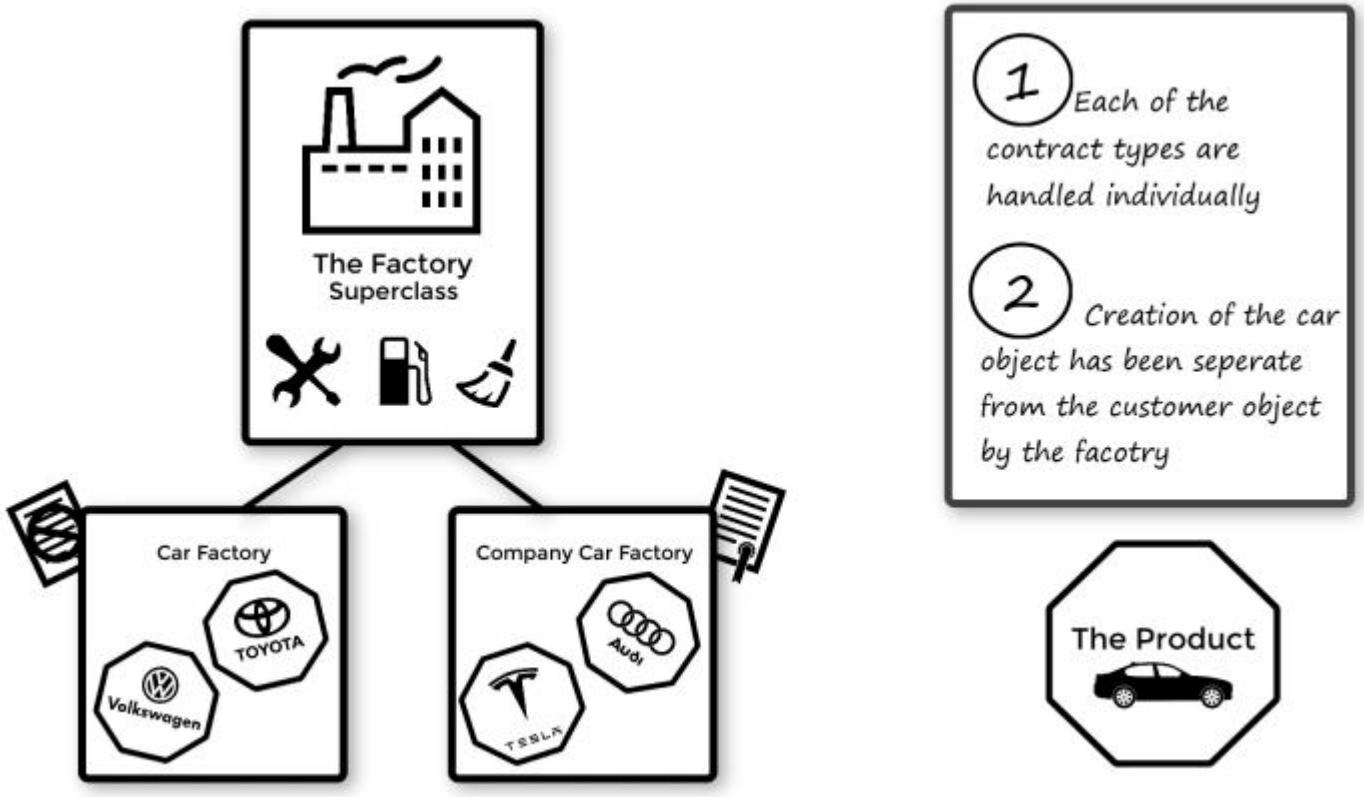
All of the above mentioned steps have all be accomplished inside the main-method of our system, as can be seen from the code sample below.

Solution

```
public static void main(String[] args) {
        Customer customerOne = new Customer("B", true);
        if(customerOne.hasCompanyContract()){
            switch (customerOne.getGradeRequest()){
                case "A":
                    carOne = new Tesla();
                    break;
                case "B":
                    carOne = new Audi();
                    break;
                default:
                    System.out.println("The requested car was unfortunately not available.");
                    carOne = null;
            }
        }else {
```

Solution

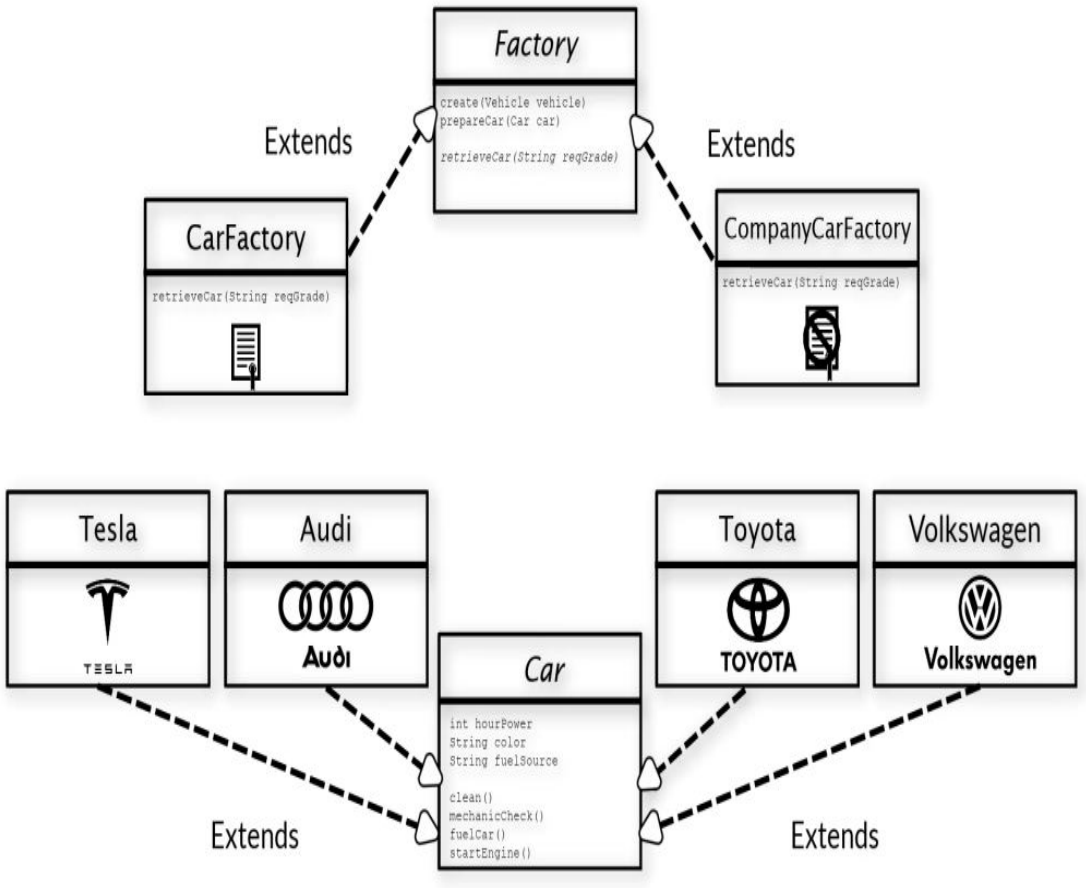
Concept Of The Factory Pattern



The core concept of the factory pattern is to make a class (the factory), which defines the rules of how to create a specific type of object (the product). The pattern can always be recognized by the create-method.

Solution

Class Diagram (UML)



- The Factory super-class contains the **create-method,** which is close to obligatory for any factory pattern. Within this method the initial step is to retrieve the requested car. However, since the super-class does not know anything about the selection logic, *it simply states the retrieveCar-method as abstract.*

- Thereby it requires any class, which extends it, must override the retrieveCar-method and implement its own selection logic.

- After having retrieved the car, we come back to the create-method. The remaining part of the method then goes into preparation procedures. This procedure is the same for all cars and we can, therefore, state the steps within the factory super-class.

Solution

Code (Java)

Link to java implementation

Note: Java files in the src/Factory Design Pattern Demo (it's an eclipse source file)

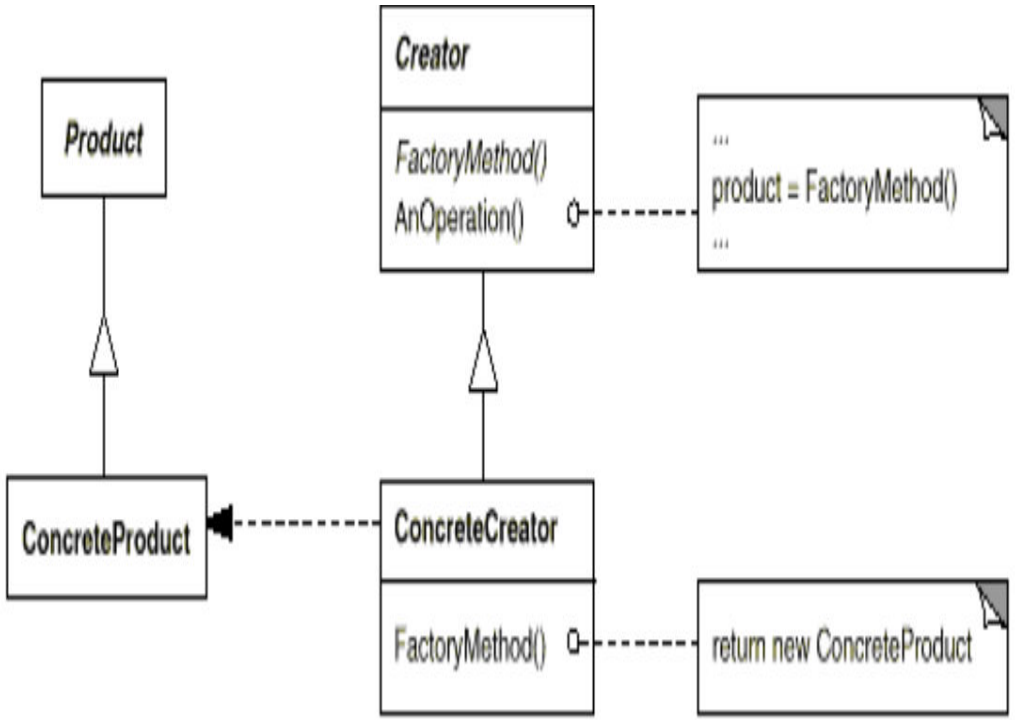
Applicability

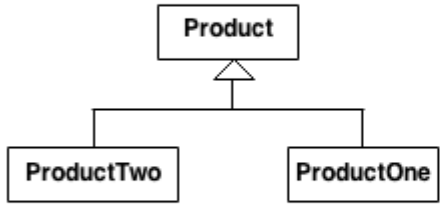
Use the Factory pattern when

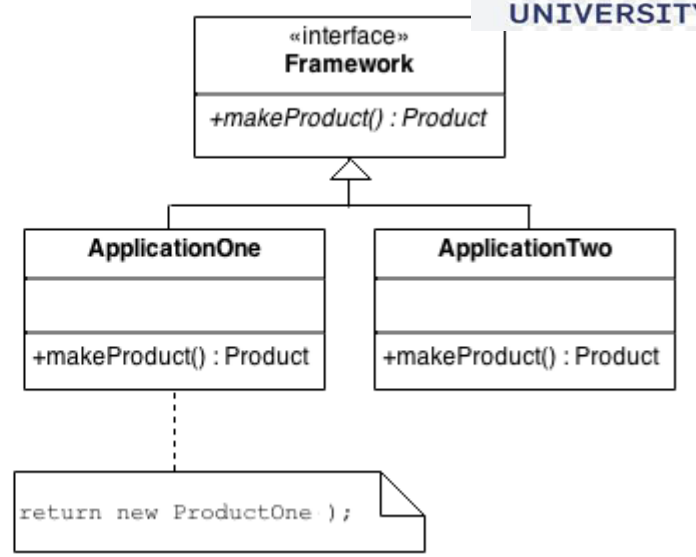
- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

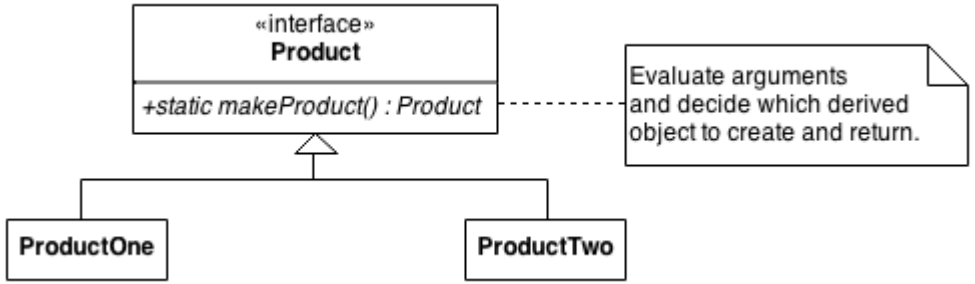
Object Oriented Analysis and Design using Java

Structure









Participants

- Product -defines the interface of objects the factory method creates.
- ConcreteProduct - implements the Product interface.
- Creator - declares the factory method, which returns an object of type Product.Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.

  - may call the factory method to create a Product object.

- ConcreteCreator -overrides the factory method to return an instance of a ConcreteProduct.

Collaboration

Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequence

1. Provides hooks for subclasses. Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.

2. Connects parallel class hierarchies. In the examples we've considered so far, the factory method is only called by Creators. But this doesn't have to be the case; clients can find factory methods useful, especially in the case of parallel class hierarchies.

Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class. Consider graphical figures that can be manipulated interactively; that is, they can be stretched, moved, or rotated using the mouse. Implementing such interactions isn't always easy. It often requires storing and updating information that records the state of the manipulation at a given time.

Pros and Cons

Pros and Cons

- ✓ You avoid tight coupling between the creator and the concrete products.
- ✓ *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
- ✓ *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.
- ✗ The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

Issues to consider when using the Factory pattern

1.**Two major varieties**. The two main variations of the Factory Method pattern are (1) the case wh Creator class is an abstract class and does not provide an implementation for the factory method it declares, and (2) the case when the Creator is a concrete class and provides a default implementation for the factory method. It's also possible to have an abstract class that defines a default implementation, but this is less common. The first case requires subclasses to define an implementation, because there's no reasonable default.

2.**Parameterized factory methods**. Another variation on the pattern lets the factory method create multiple kinds of products. The factory method takes a parameter that identifies the kind of object to create. All objects the factory method creates will share the Product interface.

3.**Language-specific** variants and issues. Different languages lend themselves to other interesting variations and caveats. Smalltalk programs often use a method that returns the class of the object to be instantiated. A Creator factory method can use this value to create a product, and a ConcreteCreator may store or even compute this value. The result is an even later binding for the type of ConcreteProduct to be instantiated.

References

Text Reference

- Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference

https://www.tutorialspoint.com/design_pattern/factory_pattern.htm
https://www.javatpoint.com/factory-method-design-pattern
https://refactoring.guru/design-patterns/factory-method
https://www.geeksforgeeks.org/factory-method-design-pattern-in-java/
https://www.digitalocean.com/community/tutorials/factory-design-pattern-in-java
https://www.baeldung.com/java-factory-pattern
https://sourcemaking.com/design_patterns/factory_method
https://www.scaler.com/topics/factory-design-pattern-in-java/

THANK YOU

Prof. Shilpa S

Department of Computer Science and Engineering

shilpas@pes.edu

Object Oriented Analysis and Design using Java

UE21CS352B

Dr.L.Kamatchi Priya
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE20CS352: Object Oriented Analysis and Design with Java

OO Design Patterns

Dr. L. Kamatchi Priya

Department of Computer Science and Engineering

UE21CS352B: Object Oriented Analysis and Design using Java

Unit-3

Creational Patterns – Builder

Agenda

- ❑ Builder-definition
- ✔ Motivation
- ✔ Intent
- ✔ Implementation
- ✔ Applicability
- ✔ Structure-Consequence
- ✔ Issues

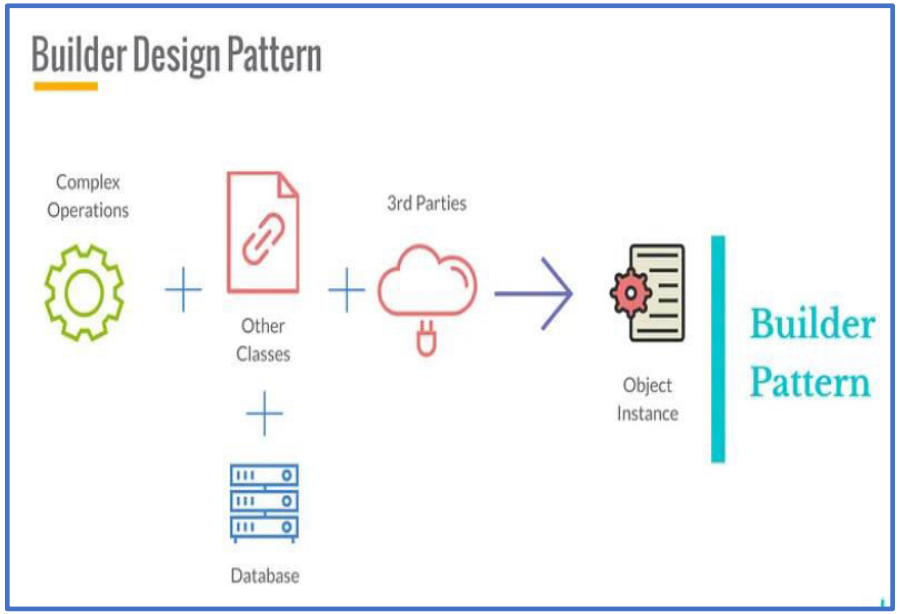
Introduction

- **Builder Design Pattern** as it was intended implies that a sequence of complex operations is needed in order to **produce an object instance**.

- The idea is that we delegate the construction of an object to a specialized class, which is aware of the complex process and logic required to build that specific instance.

- This helps us create Single Responsibility classes for complex object creation while at the same time ensuring separation of object creation from business logic.

- Based on the nature of the application, Builder implementations might lead to code re-usability, reducing the code base and improving SOLID compliance of our code.



Builder

Definition:

The Builder Design Pattern is another creational pattern designed to deal with the construction of comparatively complex objects.

When the complexity of creating object increases, the Builder pattern can separate out the instantiation process by using another object (a builder) to construct the object.

This builder can then be used to create many other similar representations using a simple step-by-step approach.

Why Builder Pattern?

The Builder design pattern solves problems like:

- How can a class (the same construction process) create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

The Builder design pattern describes how to solve such problems:

- Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
- A class delegates object creation to a Builder object instead of creating the objects directly.

A class (the same construction process) can delegate to different Builder objects to create different representations of a complex object.

Builder : Class, Object Structural

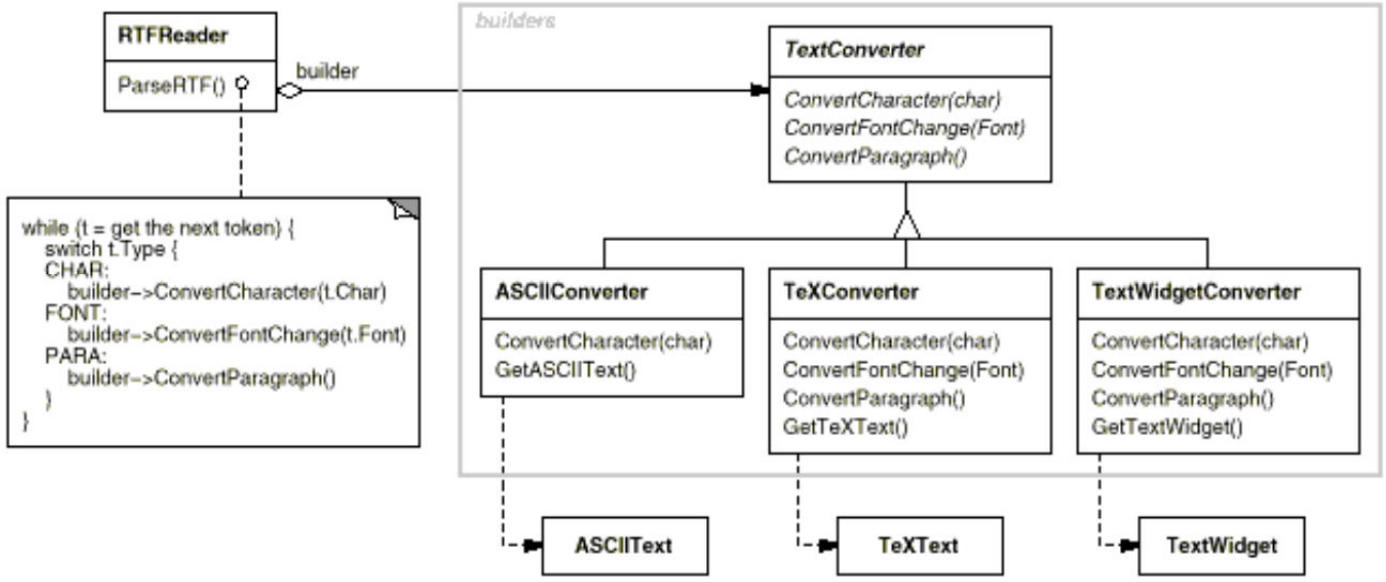
Motivation

- A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats. The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively.

- The **problem**, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

- A **solution** is to configure the RTFReader class with a TextConverter object that converts RTF to another textual representation.

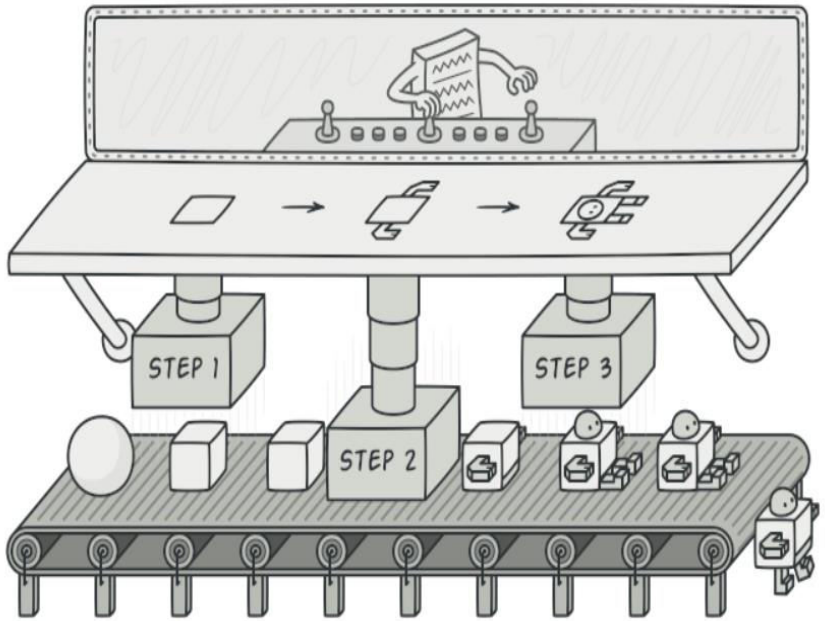
- The Builder pattern captures all these relationships. **Each converter class is called a builder** in the pattern, and the **reader is called the director.**



Builder : Class, Object Structural

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations



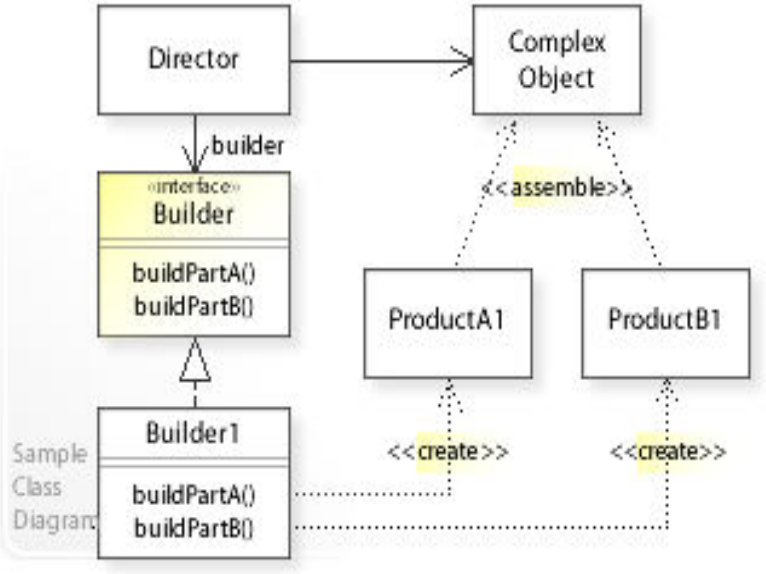
Think of a car factory

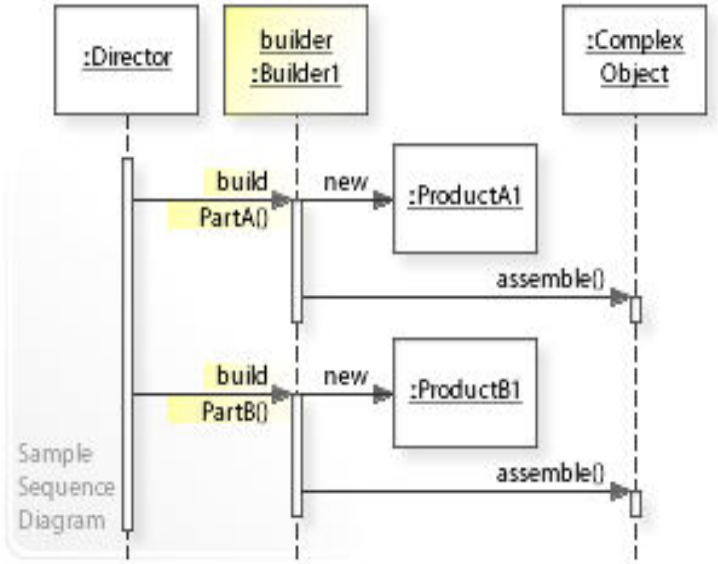
Boss tells workers (or robots) to build each part of a car
Workers build each part and add them to the car being constructed

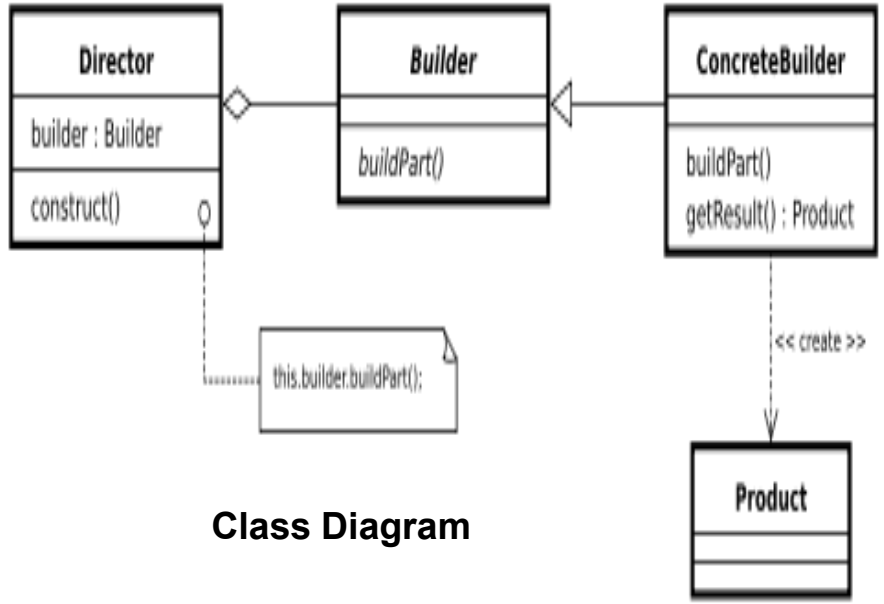
Object Oriented Analysis and Design with Java

Builder : Implementation

UML class diagram for the Builder Pattern







Class Diagram

Builder

Abstract interface for creating objects (product).

ConcreteBuilder

Provides implementation for Builder. It is an object able to construct other objects. Constructs and assembles parts to build the objects.

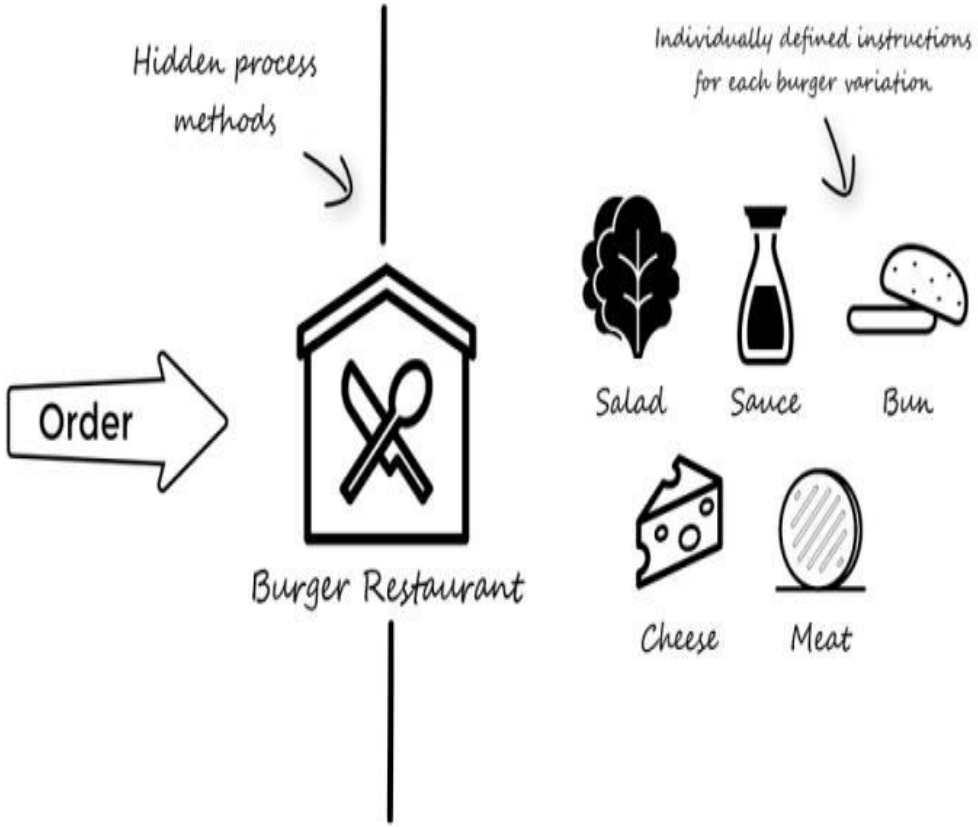
Builder : Implementation example-1

Problem Statement: Case study

- We will set the objective of making a burger restaurant, which can make different variations of burgers.

- Secondly it would also be preferable to have defined instructions for how to build each of the different variations of burgers (e.g. cheeseburger), so that we do not have to provide all the ingredients each time when we are making a burger.

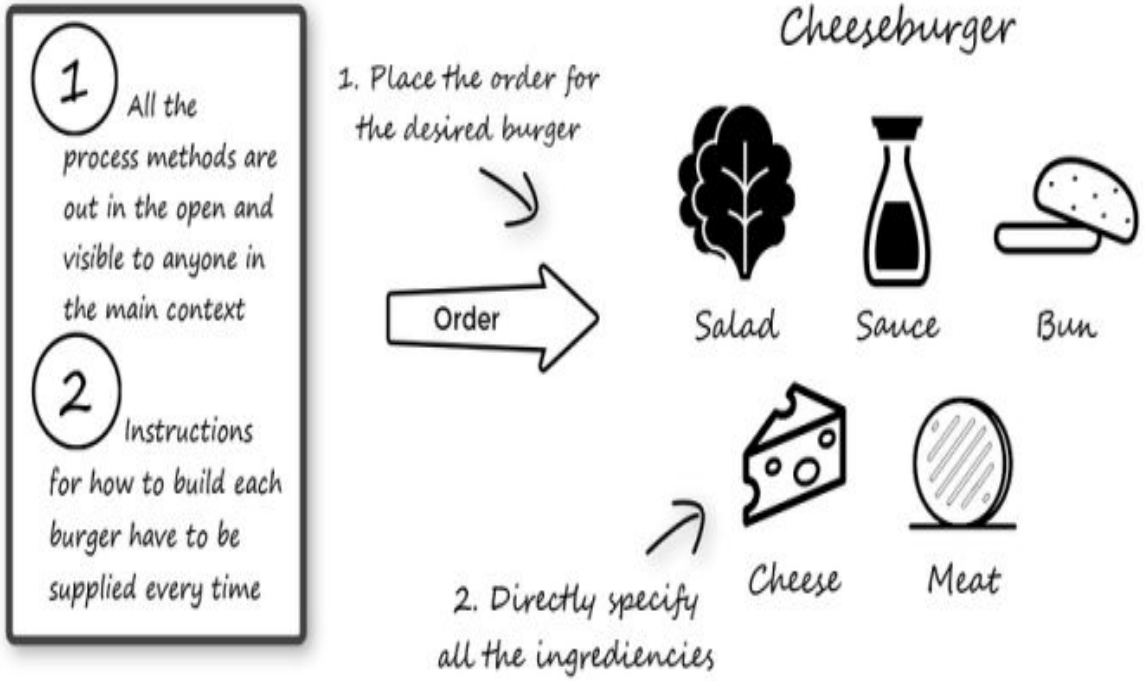
- Before, how the builder design pattern will be able to solve these issues, we will start by seeing how this system could be build based on construction of the burger object in the main context.



Object Oriented Analysis and Design with Java

Builder : Implementation example-1

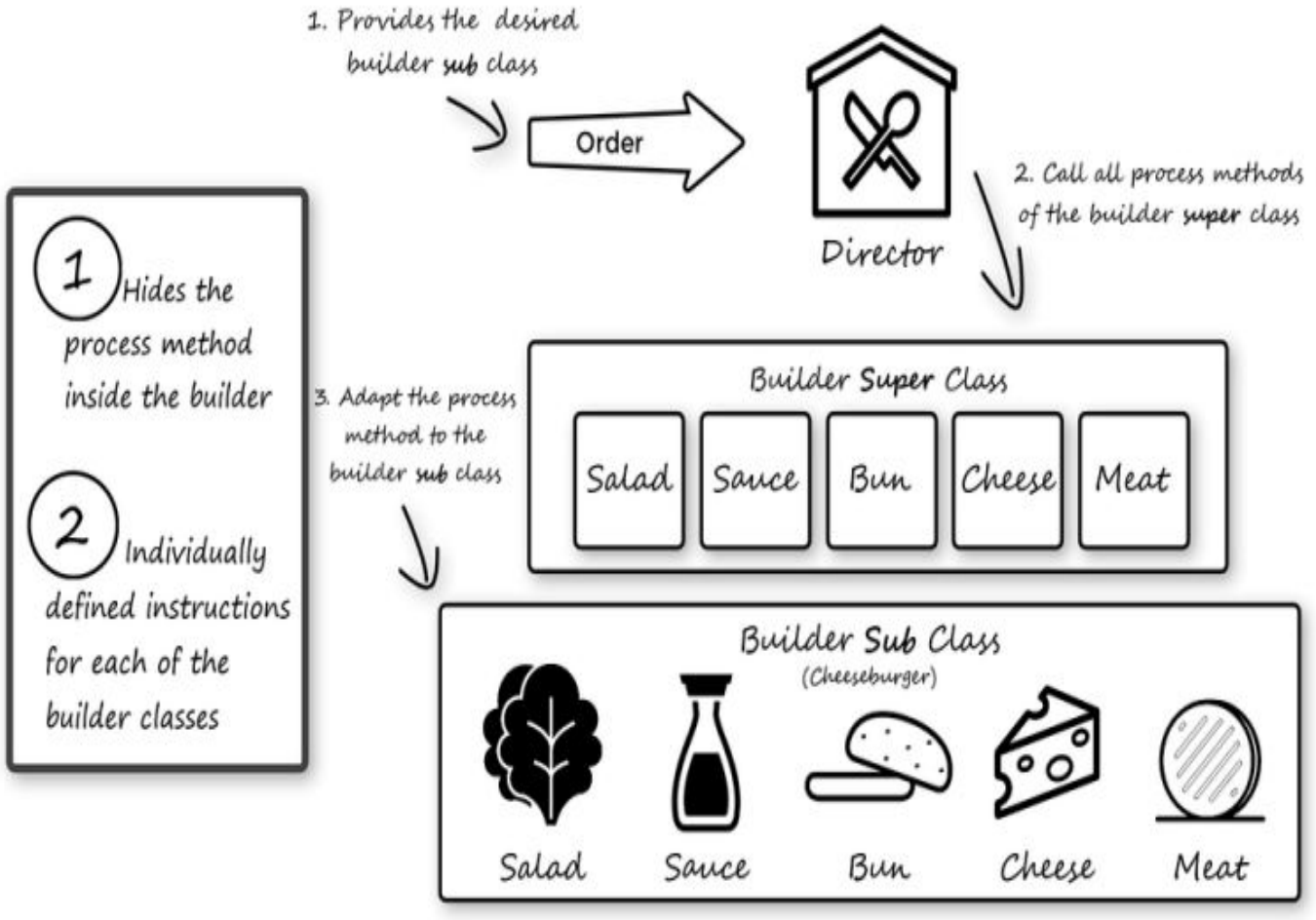
Solution: Construction In The Main Context




```
public class Main {
    public static void main(String[] args) {
        Burger cheeseBurger = new Burger();
        cheeseBurger.setBun("White Bread");
        cheeseBurger.setMeat("Beef");
        cheeseBurger.setSalad("Iceberg");
        cheeseBurger.setCheese("American Chesse");
        cheeseBurger.setSauce("Secret Sauce");
        cheeseBurger.print();
    }
```

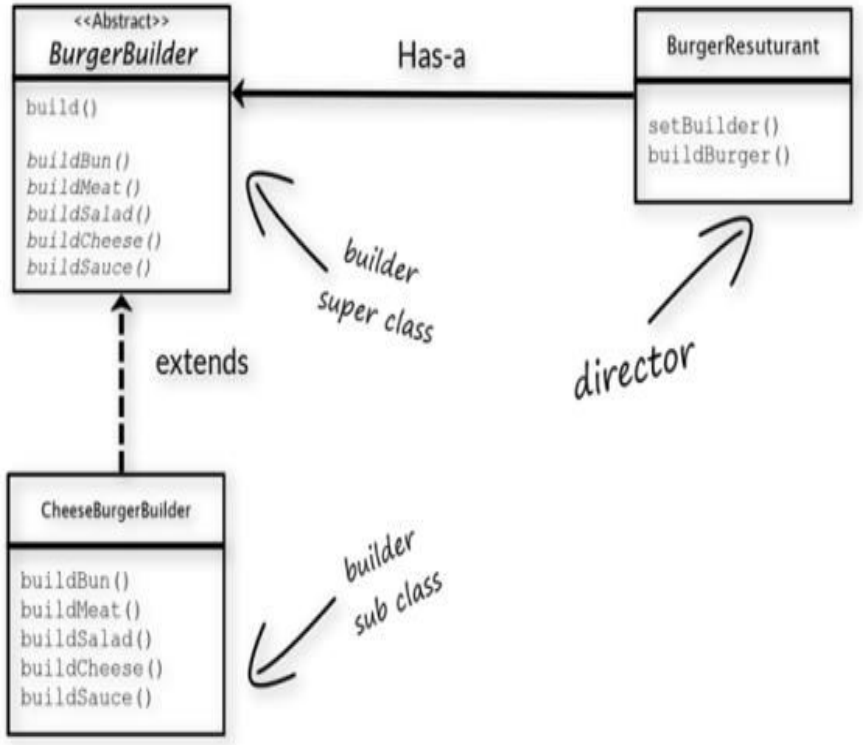
Solution: Concept Of The Builder Pattern

- When talking about the builder design pattern, it is important to understand the concept of the **Director and the Builder.**
- The **director's job** is to invoke the building process of the builder. The **builder's job** is to manage the different building procedures associated with each of the different variations of objects, in this case the burgers.
- The builder pattern consists of two classes, a sub- and super class.



Design Solution :Class Diagram (UML)

- Builder pattern consists of two main class types: the builder and the director
- With the context of program we will be using the builder pattern, it means we actually only want to be interacting with the director.
- The director will ensure that we build the correct burger object based on which builder we provide it with.
- Inside the director, we set its builder object field (setBuilder-method) and afterwards ask it to build the object based on the provided builder (build-method).



Solution : Java Code

Link to java implementation

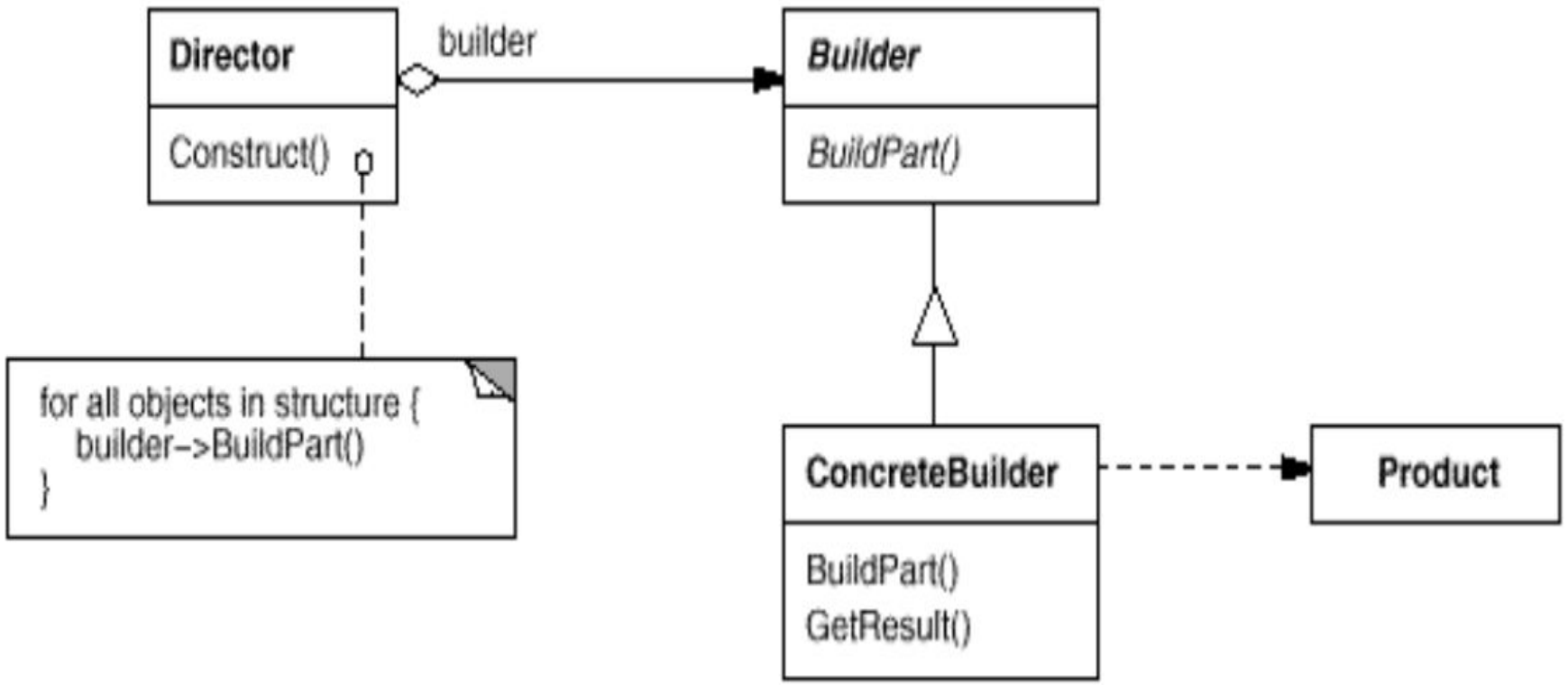
Note: Java files in the src/builderPatternDemo (it's a eclipse source file)

Applicability

Use the Builder pattern when

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for the object that's constructed.
- You want to get rid of a "telescoping constructor". Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters.
- You want your code to be able to create different representations of some product (for example, stone and wooden houses). The Builder pattern can be applied when construction of various representations of the product involves similar steps that differ only in the details.

Structure


Director
builder
Builder
Construct()
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
Product
BuildPart()
GetResult()

Participants

Builder (TextConverter)
o specifies an abstract interface for creating parts of a Product object.

ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)
o constructs and assembles parts of the product by implementing the Builder interface.
o defines and keeps track of the representation it creates.
o provides an interface for retrieving the product (e.g., GetASCIIText, GetTextWidget). ·

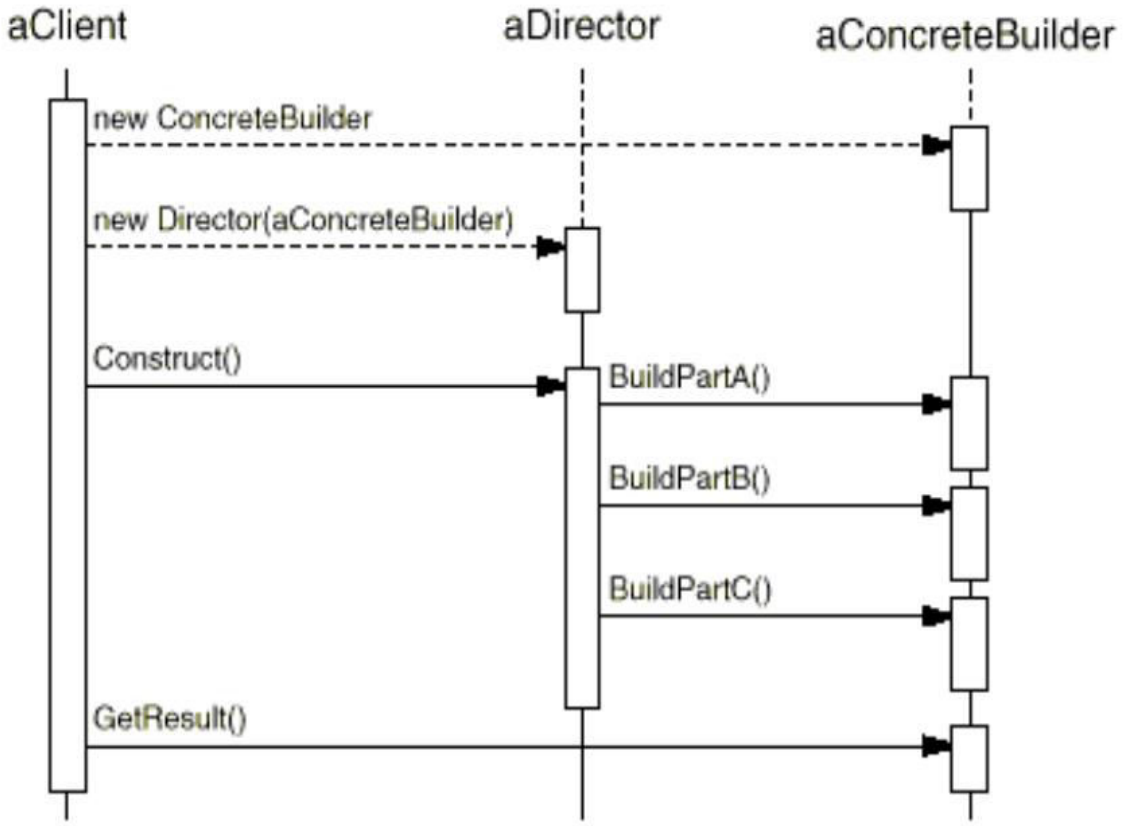
Director (RTFReader)
o constructs an object using the Builder interface.

Product (ASCIIText, TeXText, TextWidget)
o represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
o includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaboration

- Client creates Director object and configures it with a Builder
- Director notifies Builder to build each part of the product
- Builder handles requests from Director and adds parts to the product
- Client retrieves product from the Builder

The following interaction diagram illustrates how Builder and Director cooperate with a client.



Consequence

- Lets you vary a product's internal representation by using different Builders
- Isolates code for construction and representation
- Gives finer-grain control over the construction process

Issues to consider when using the Builder pattern

- Assembly and construction interface: generality
- Is an abstract class for all Products necessary?
- Usually products don't have a common interface
- Usually there's an abstract Builder class that defines an operation for each component that a director may ask it to create.
- These operations do nothing by default (empty, static methods)
- The ConcreteBuilder overrides operations selectively

How to Implement

1.Make sure that you can clearly define the common construction steps for building all available product representations.

1.Declare these steps in the base builder interface.

1.Create a concrete builder class for each of the product representations and implement their construction steps.

1.Think about creating a director class. It may encapsulate various ways to construct a product using the same builder object.

1.The client code creates both the builder and the director objects. Before construction starts, the client must pass a builder object to the director.

1.The construction result can be obtained directly from the director only if all products follow the same interface. Otherwise, the client should fetch the result from the builder.

Pro and Cons

Pros

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.

Cons

- The overall complexity of the code increases since the pattern requires creating multiple new classes.

References

Text Reference

- Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference

https://www.baeldung.com/creational-design-patterns
https://www.javatpoint.com/builder-design-pattern
https://sourcemaking.com/design_patterns/builder
https://integu.net/builder-pattern/
https://www.c-sharpcorner.com/article/builder-design-pattern-with-java/

THANK YOU

Dr.L.Kamatchi Priya

Department of Computer Science and Engineering

priyal@pes.edu

Object Oriented Analysis and Design using Java

UE21CS352B

Dr. L. Kamatchi Priya
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE20CS352: Object Oriented Analysis and Design using Java

OO Design Patterns

Creational Patterns – Prototype

Dr.L.Kamatchi Priya

Department of Computer Science and Engineering

Agenda

- ❑ Prototype-definition
- ✔ Motivation
- ✔ Intent
- ✔ Implementation
- ✔ Applicability
- ✔ Structure-Consequence
- ✔ Issues

Prototype : Class, Object Structural

Motivation

The Prototype Pattern specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

Use the Prototype Pattern when a client needs to create a set of objects that are alike or differ from each other only in terms of their state and creating an instance of a such object (e.g., using the “new” keyword) is either expensive or complicated.

The Prototype Pattern allows you to make new instances by copying existing instances.

– In Java this typically means using the clone() method or de-serialization when you need deep copies

Key aspect of this pattern:

– Client code can make new instances without knowing which specific class is being instantiated

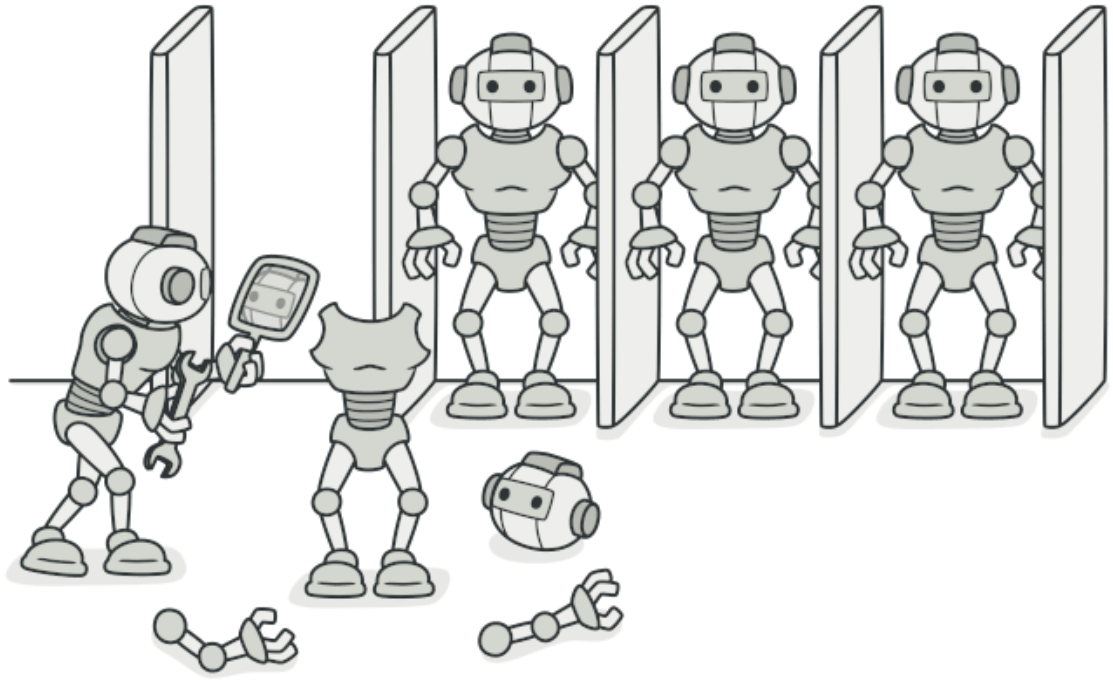
Prototype : Class, Object Structural

Intent

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

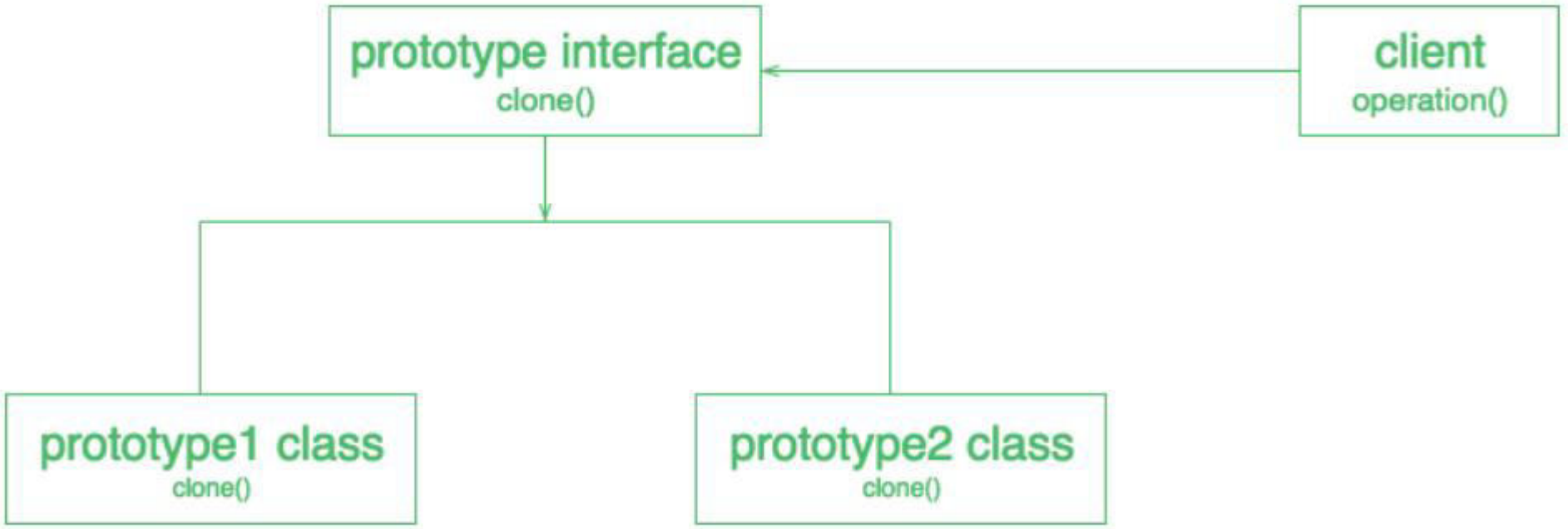
Usage examples: The Prototype pattern is available in Java out of the box with a Cloneable interface.
Any class can implement this interface to become cloneable.

java.lang.Object#clone() (class should implement the java.lang.Cloneable interface)
Identification: The prototype can be easily recognized by a clone or copy methods, etc.



Prototype : Implementation

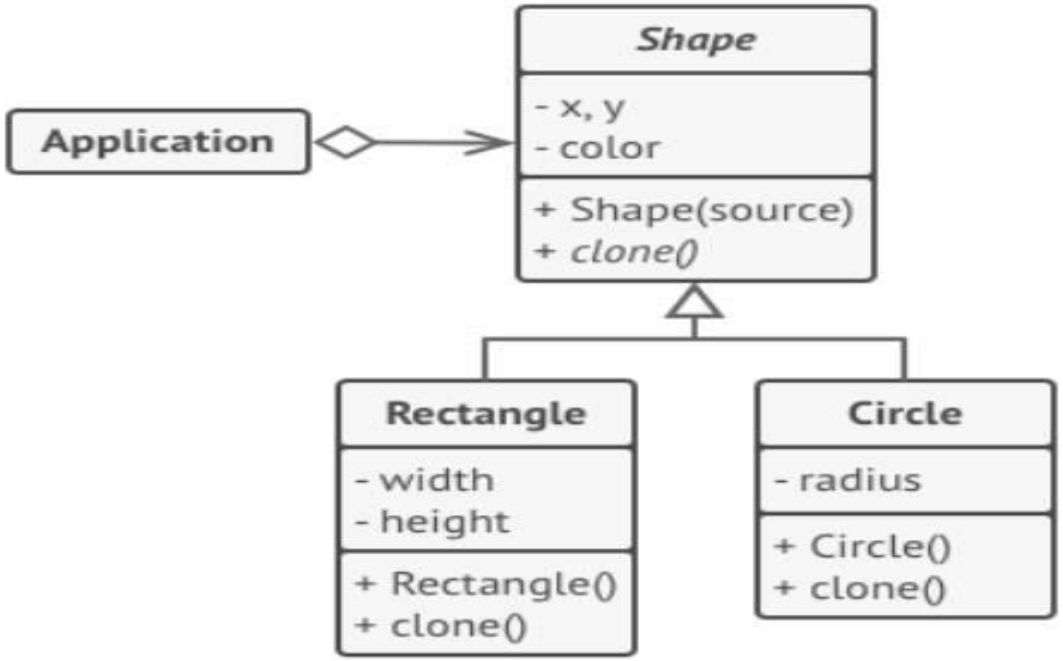
UML class diagram for the Prototype Pattern



Prototype : Implementation example-1

Problem Statement:

In this example, the **Prototype** pattern lets you produce exact copies of geometric objects, without coupling the code to their classes.



Cloning a set of objects that belong to a class hierarchy.

All shape classes follow the same interface, which provides a cloning method. A subclass may call the parent's cloning method before copying its own field values to the resulting object.

Solution

Let's take a look at how the Prototype can be implemented without the standard Cloneable interface.

Link to Java Implementation

Note: Its example for prototype with different shape objects

Its eclipse file java files will be in **src/prototypeDesignPatternDemo**

Prototype registry

You could implement a centralized prototype registry (or factory), which would contain a set of pre-defined prototype objects. This way you could retrieve new objects from the factory by passing its name or other parameters. The factory would search for an appropriate prototype, clone it and return you a copy.

Note: Use main method which is implemented using **BundledShapeCache**

```
BundledShapeCache cache = new BundledShapeCache();
```

Applicability

Use the Prototype pattern when

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

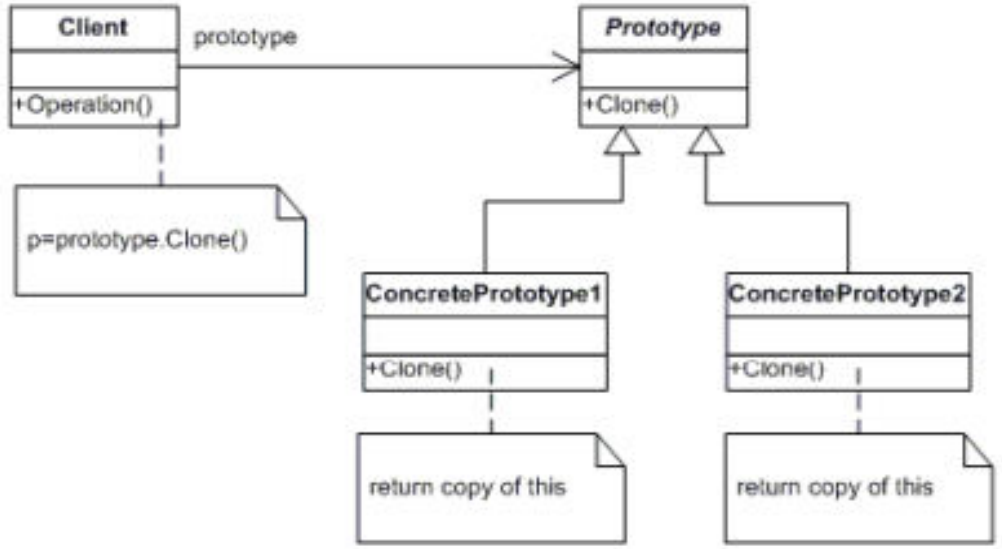
Structure

Participants

Prototype: declares an interface for cloning itself.

ConcretePrototype: implements an operation for cloning itself.

Client: creates a new object by asking a prototype to clone itself and then making required modifications.



Collaboration

A client asks a prototype to clone itself.

Consequence

Additional benefits of the Prototype pattern are listed below.

- Adding and removing products at run-time.
- Specifying new objects by varying values
- Specifying new objects by varying structure.
- Reduced subclassing.
- Configuring an application with classes dynamically.

Issues to consider when using the Prototype pattern

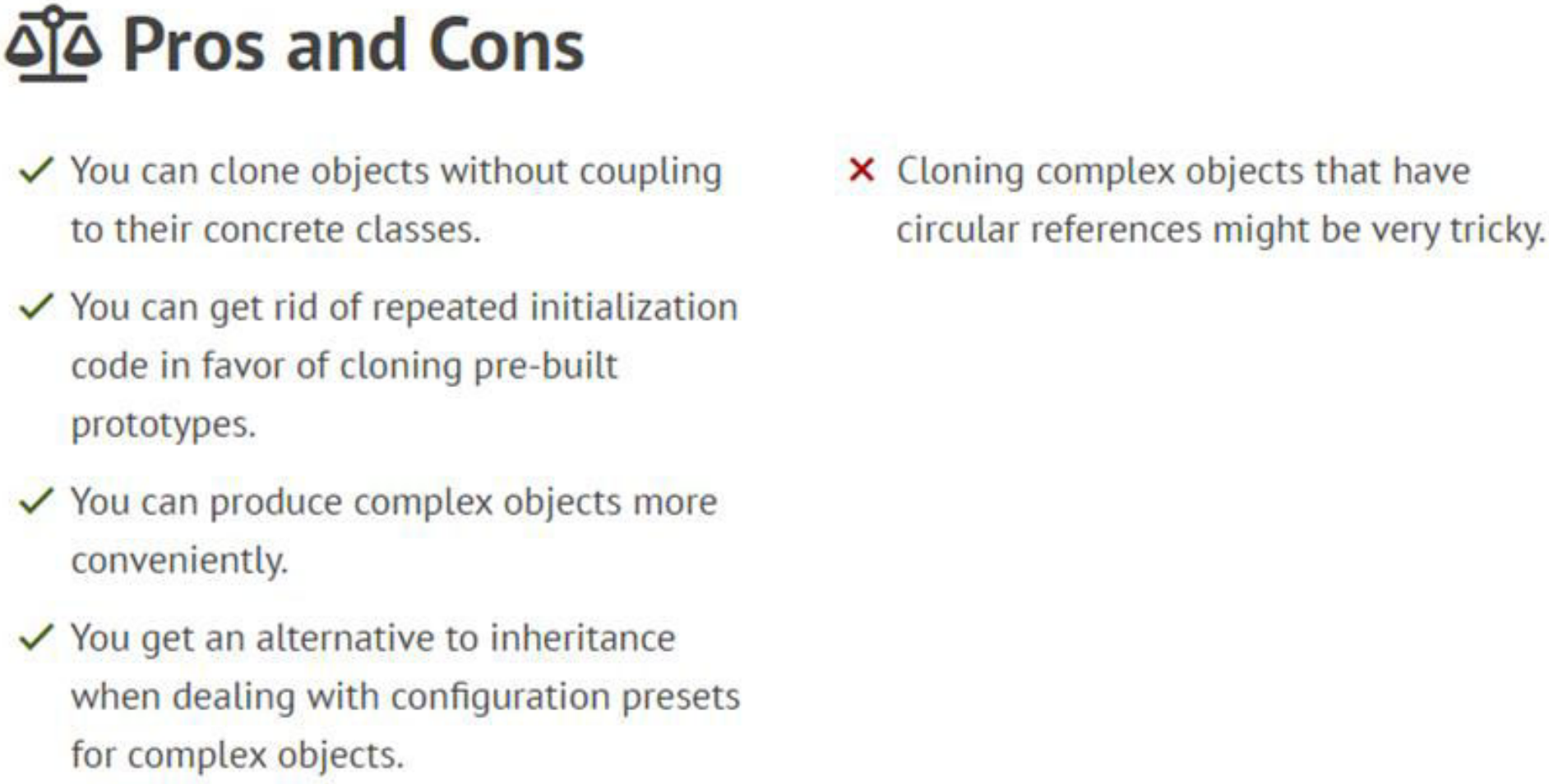
Consider the following issues when implementing prototypes:

- **Using a prototype manager**. : When the number of prototypes in a system isn't fixed (that is, they can be created and destroyed dynamically), keep a registry of available prototypes. Clients won't manage prototypes themselves but will store and retrieve them from the registry. A client will ask the registry for a prototype before cloning it. We call this registry a prototype manager.

- **Implementing the Clone operation**. The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references.

- **Initializing clones**. While some clients are perfectly happy with the clone as is, others will want to initialize some or all of its internal state to values of their choosing. You generally can't pass these values in the Clone operation, because their number will vary between classes of prototypes. Some prototypes might need multiple initialization parameters; others won't need any. Passing parameters in the Clone operation precludes a uniform cloning interface.

Pros and Cons



References

Text Reference

- Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference

https://sourcemaking.com/design_patterns/prototype
https://www.geeksforgeeks.org/prototype-design-pattern/

THANK YOU

Dr. L. Kamatchi Priya

Department of Computer Science and Engineering

priyal@pes.edu