Object Oriented Analysis and Design with Java

UE20CS352

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

UE20CS352: Object Oriented Analysis and Design with Java

Unit-4
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

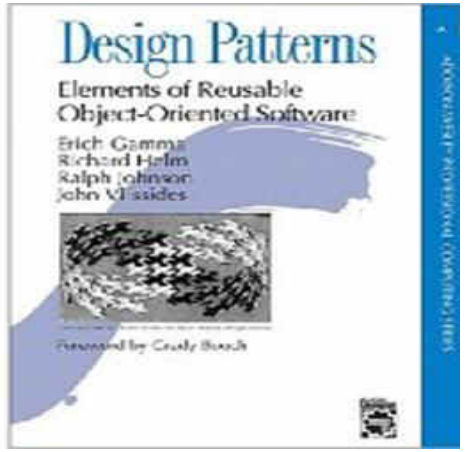
Christopher Alexander, architect

- 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

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 with Java

UE20CS352

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

UE20CS352: Object Oriented Analysis and Design with Java

Unit-4
Selection or Usage of a Design Patterns

Agenda

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

Design pattern-Recap

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

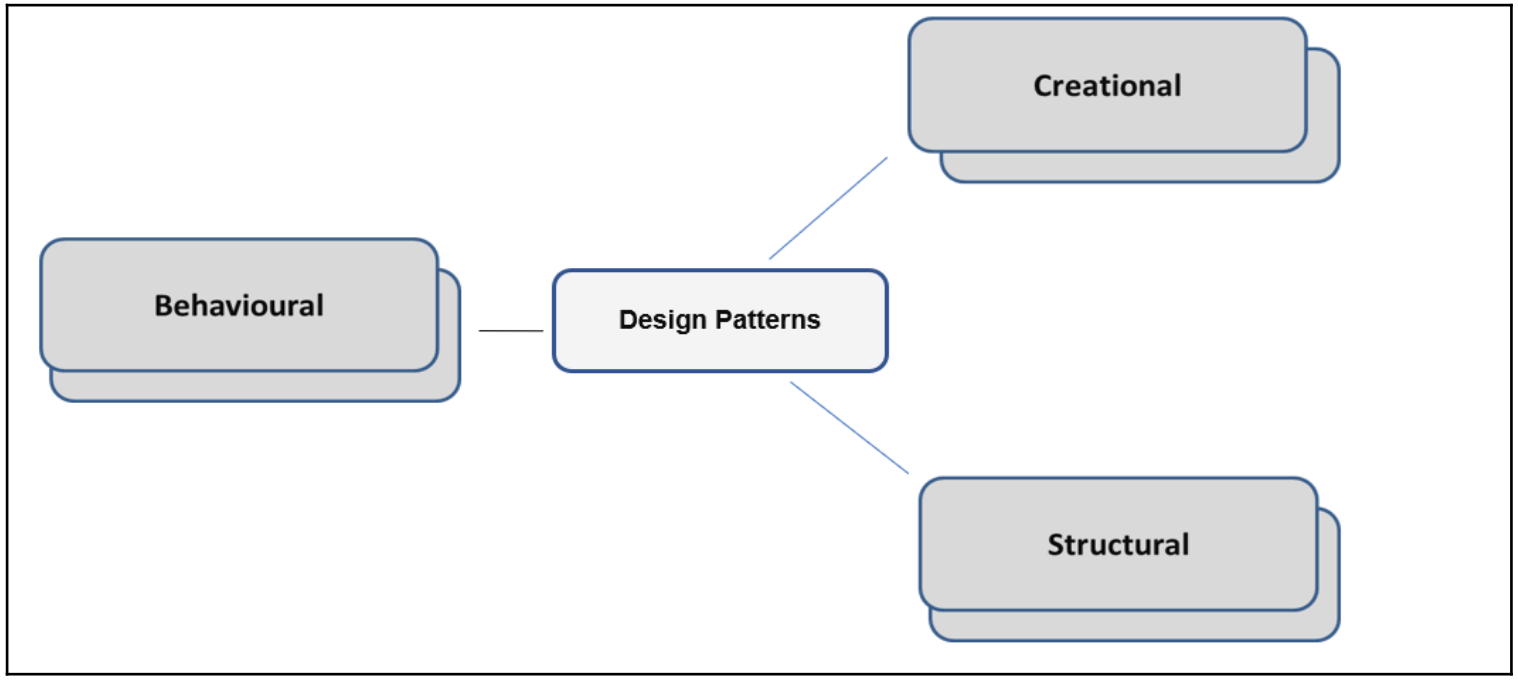


Fig 1: Categories of Design Patterns

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.

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?

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

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

- 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.

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 with Java

UE20CS352

Prof. Nivedita Kasturi
Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design with Java

OO Design Patterns

Prof. Nivedita Kasturi

Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design with Java

Unit-4

Creational Patterns – Singleton

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();
```

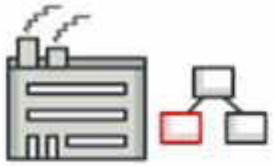
The *new* operator is often considered harmful as it scatters objects all over the application. Over time it can become challenging to change an implementation because classes become tightly coupled.

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

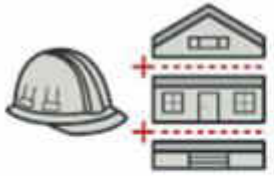
Singleton

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



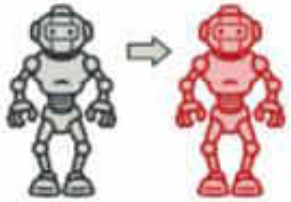
Factory Method

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.

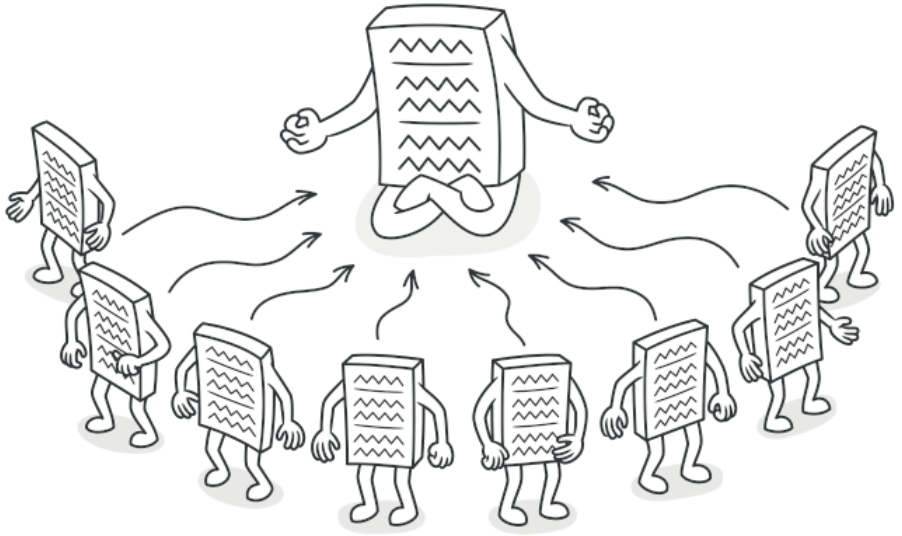
Singleton: Class, Object Structural

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.

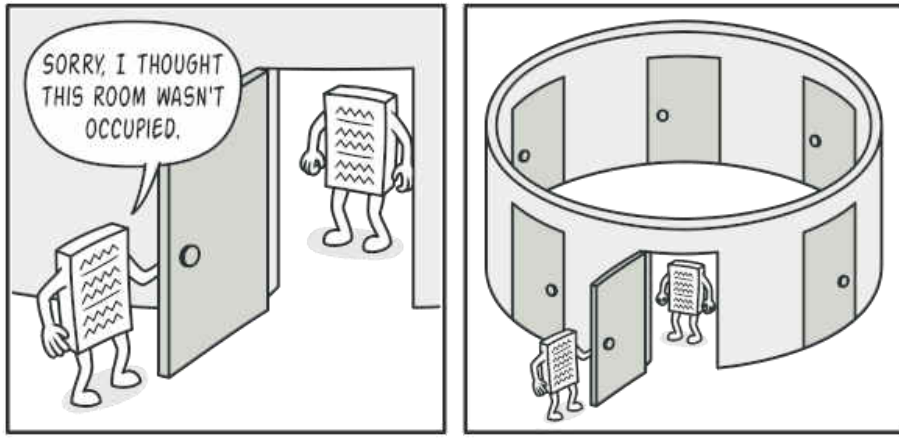
Regardless of the personal identities of the individuals who form governments, the title, “The Government of X”, is a global point of access that identifies the group of people in charge.

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

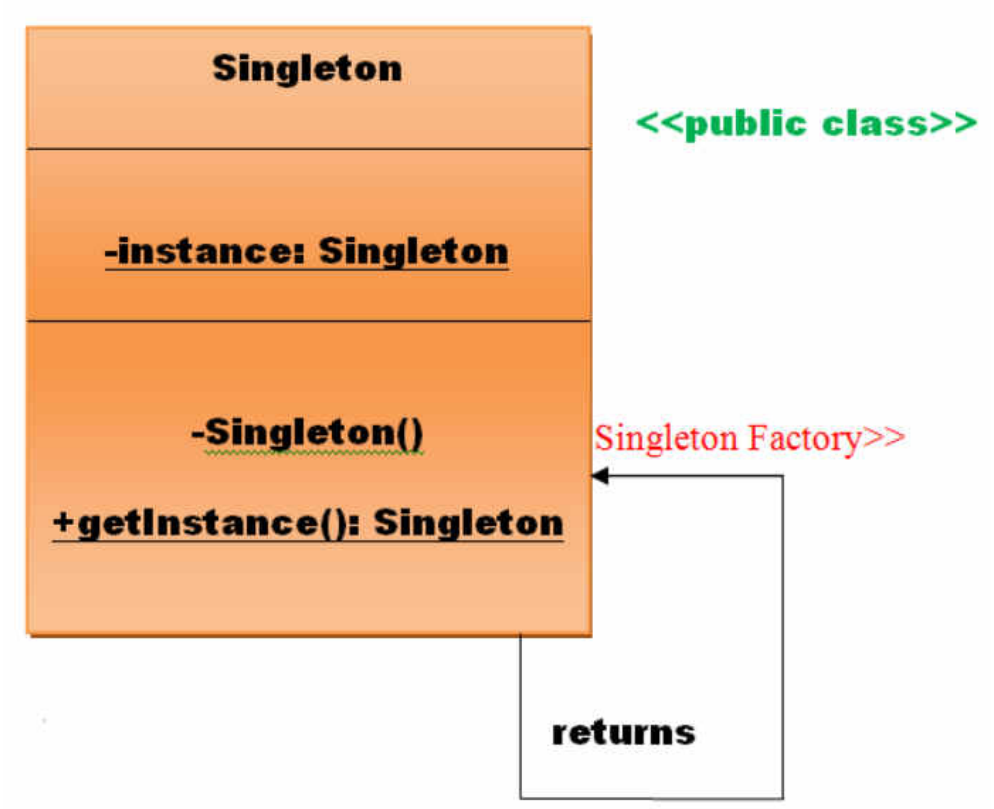
All implementations of the Singleton have these two steps in common:

- 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.

Singleton: Implementation

UML class diagram for the Singleton Pattern



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

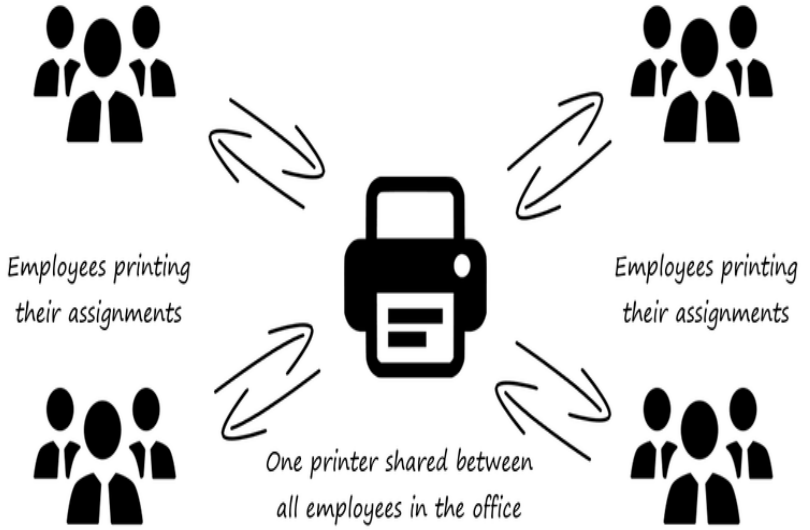
Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade, etc.

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

Singleton: Implementation example

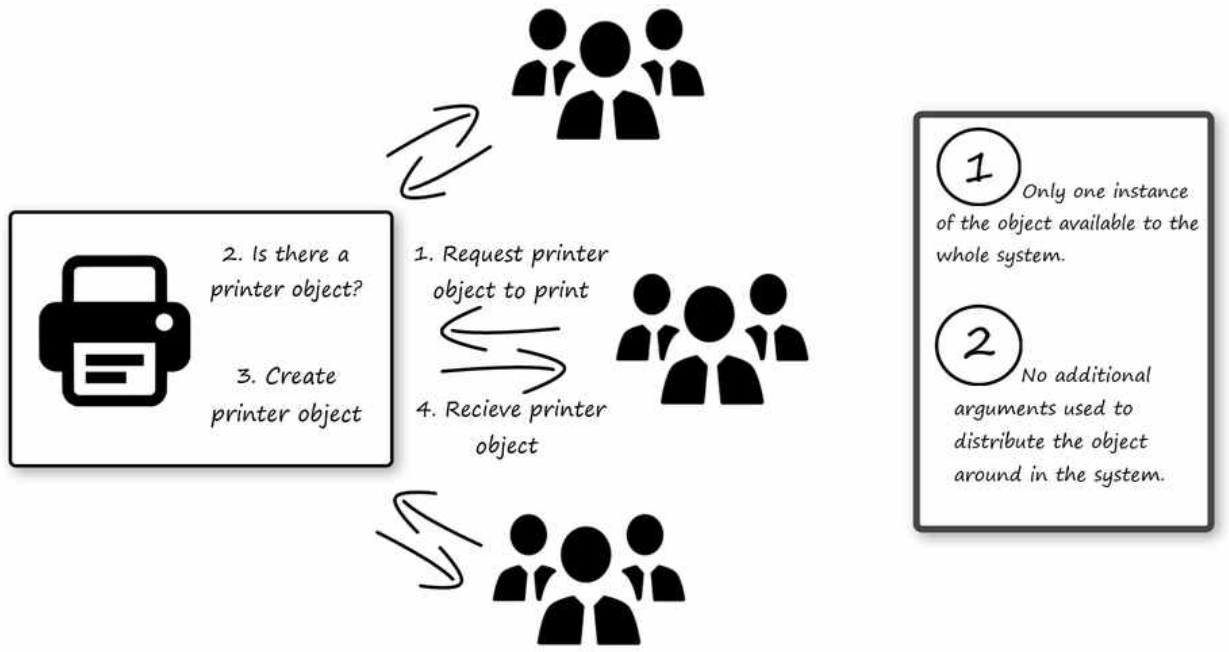
- **Problem Statement**:

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.



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.

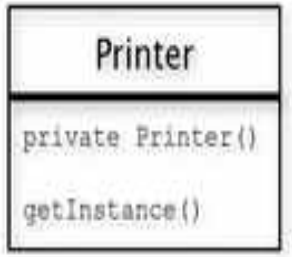
- Now that we have an instance of the class ready to work with inside the class, we null check on it when we call the static initialization method. If the class has not previously been initialized, then we initialize it, otherwise we return our static field instance.

- 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" + "---------");
    }
}
```

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");
    }
}
```

Code (Java)

How To Use The Singleton Pattern

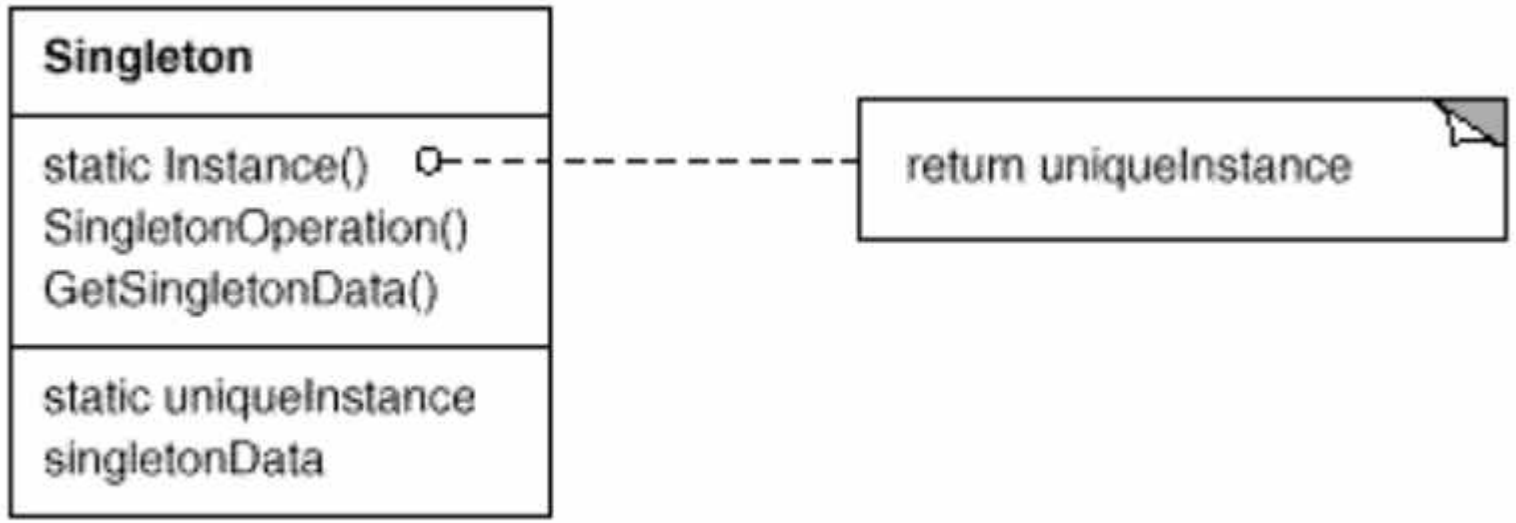
```
public class Main {
    public static void main(String[] args) {

        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.

Structure

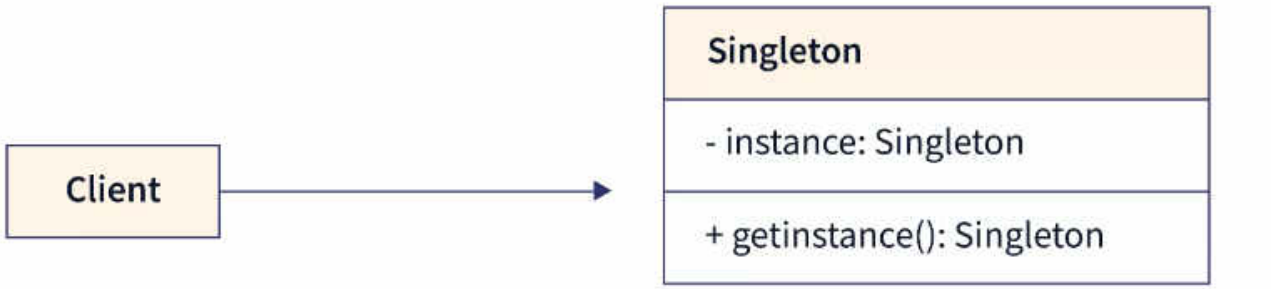

Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

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 make sure 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.

Relations with Other Patterns

- A **Facade** class can often be transformed into a **Singleton** since a single facade object is sufficient in most cases.

- **Flyweight** would resemble **Singleton** if you somehow managed to reduce all shared states of the objects to just one flyweight object. But there are two fundamental differences between these patterns:
  - There should be only one Singleton instance, whereas a Flyweight class can have multiple instances with different intrinsic states.
  - The Singleton object can be mutable. Flyweight objects are immutable.

- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.
  - Facade objects are often Singletons because only one Facade object is required.

Demo

Singleton Pattern_ Example. txt

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 Nivedita Kasturi

Department of Computer Science and Engineering

niveditak@pes.edu@pes.edu

Object Oriented Analysis and Design with Java

UE20CS352

Prof Nivedita Kasturi
Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design with Java

OO Design Patterns

Prof Nivedita Kasturi

Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design with Java

Unit-4

Creational Patterns – Factory

Agenda

- ❑ Introduction to Structural design patterns
- ❑ Types of Structural Design Patterns.
- ❑ 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

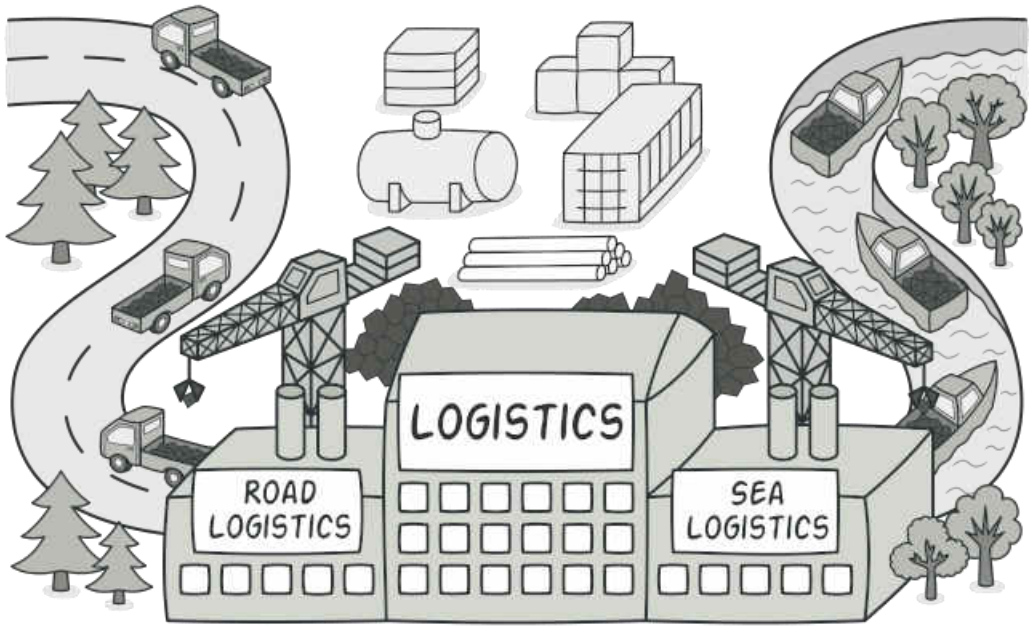
Motivation

- 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

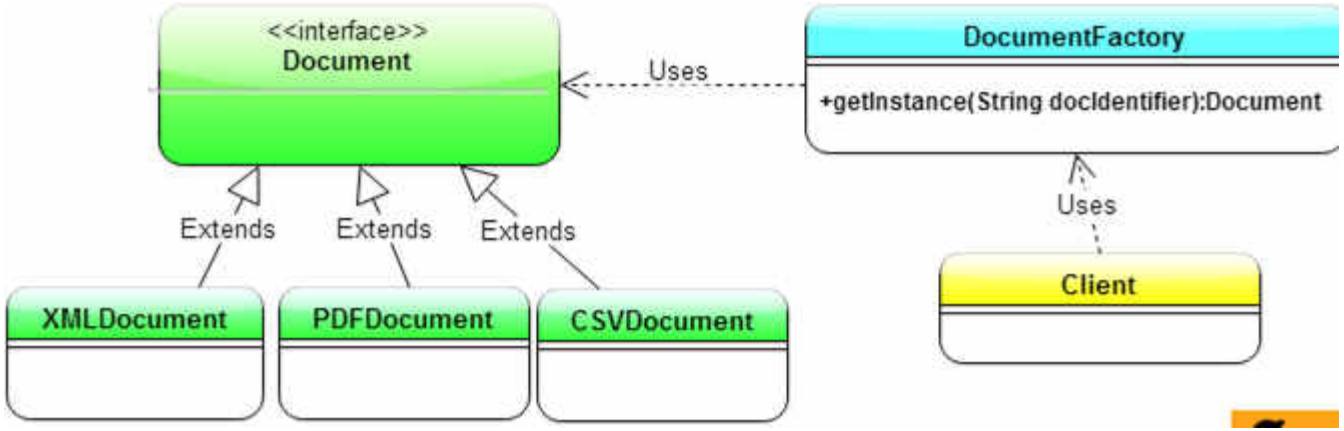
Intent

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.



Factory: Implementation

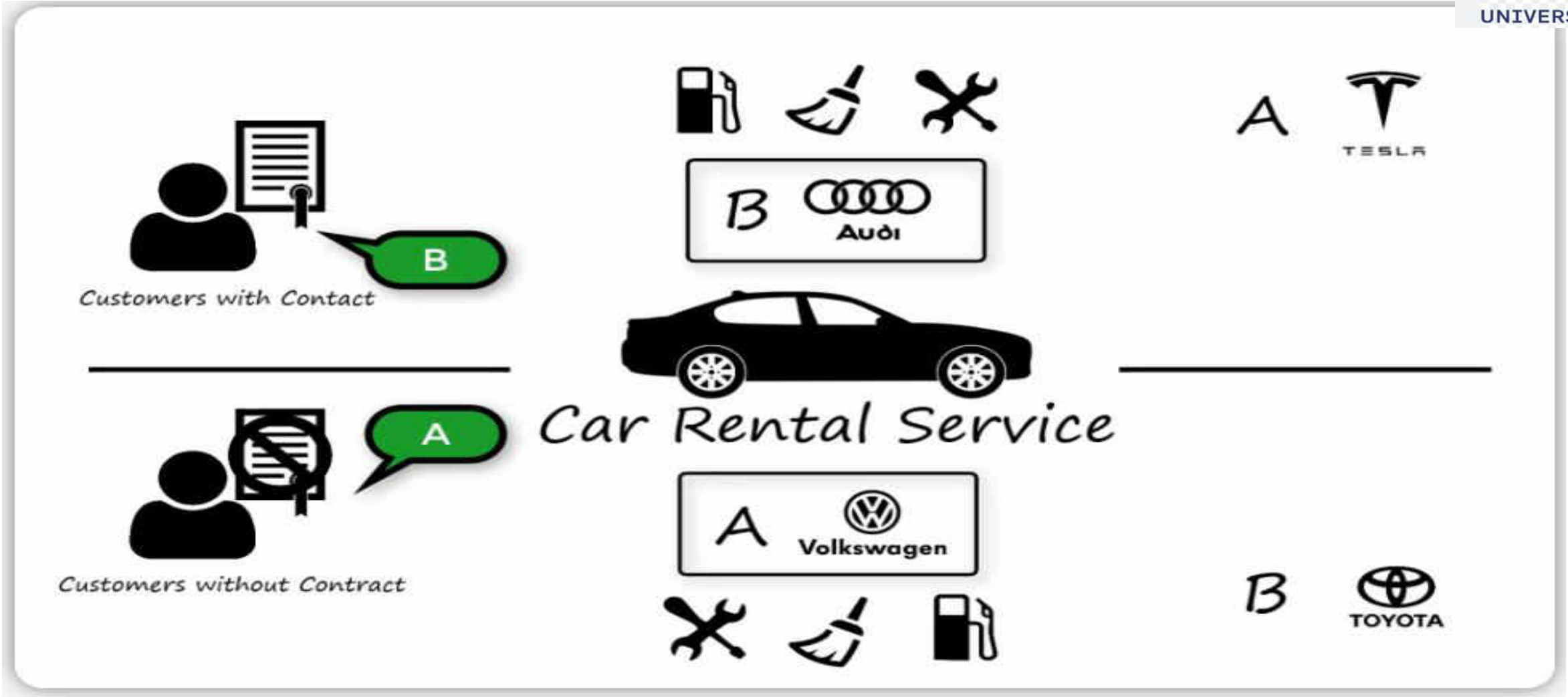
UML class diagram for the Factory Pattern



Object Oriented Analysis and Design with 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.

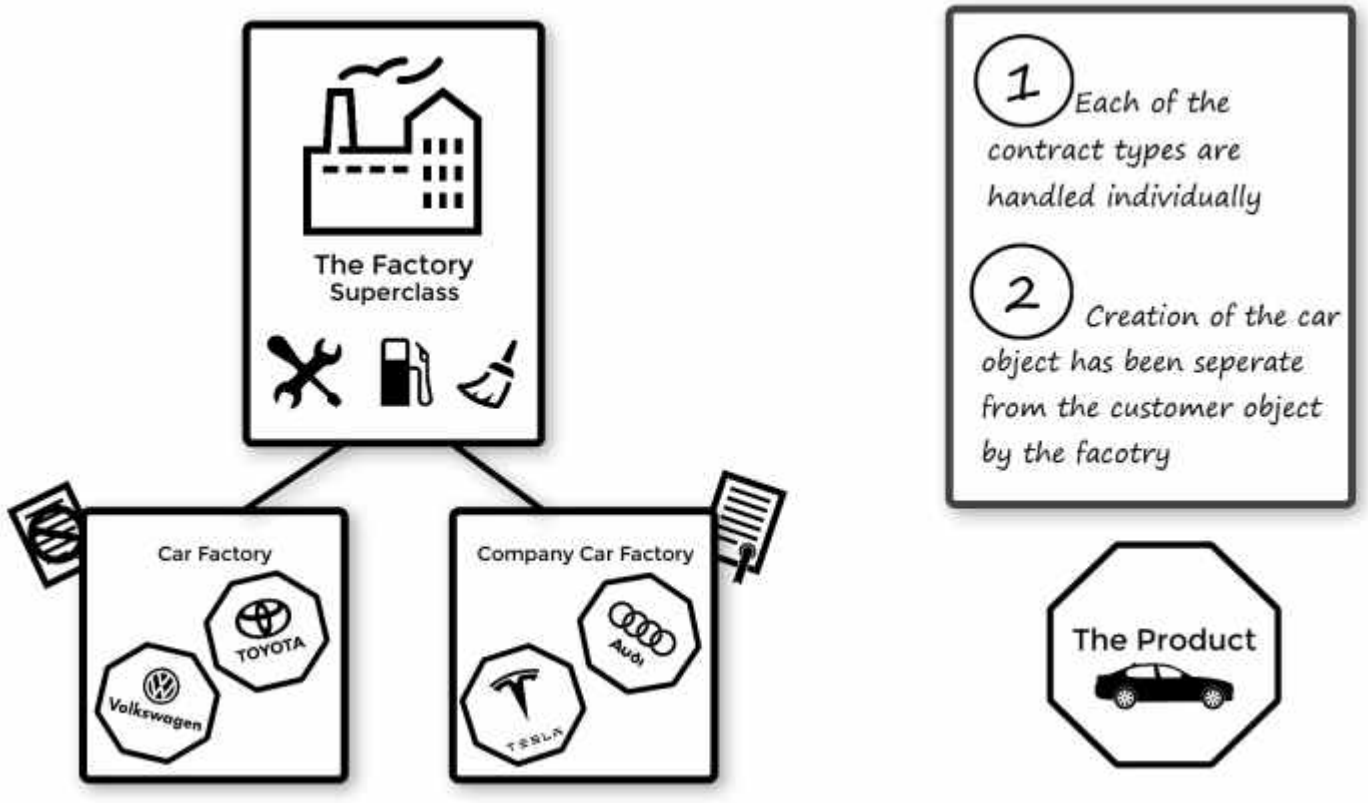
All of the above mentioned steps have all be accomplished inside the main-method of our system, as can be seen from the code sample below.

Solution

```
public static void main(String[] args) {
        }else {
            switch (customerOne.getGradeRequest()){
                case "A":
                    carOne = new Volkswagen();
                    break;
                case "B":
                    carOne = new Toyota();
                    break;
                default:
                    System.out.println("The requested car was unfortunately not available.");
                    carOne = null;
            }
        }
        carOne.clean();
        carOne.mechanicCheck();
        carOne.fuelCar();
        carOne.startEngine();
    }
```

Solution

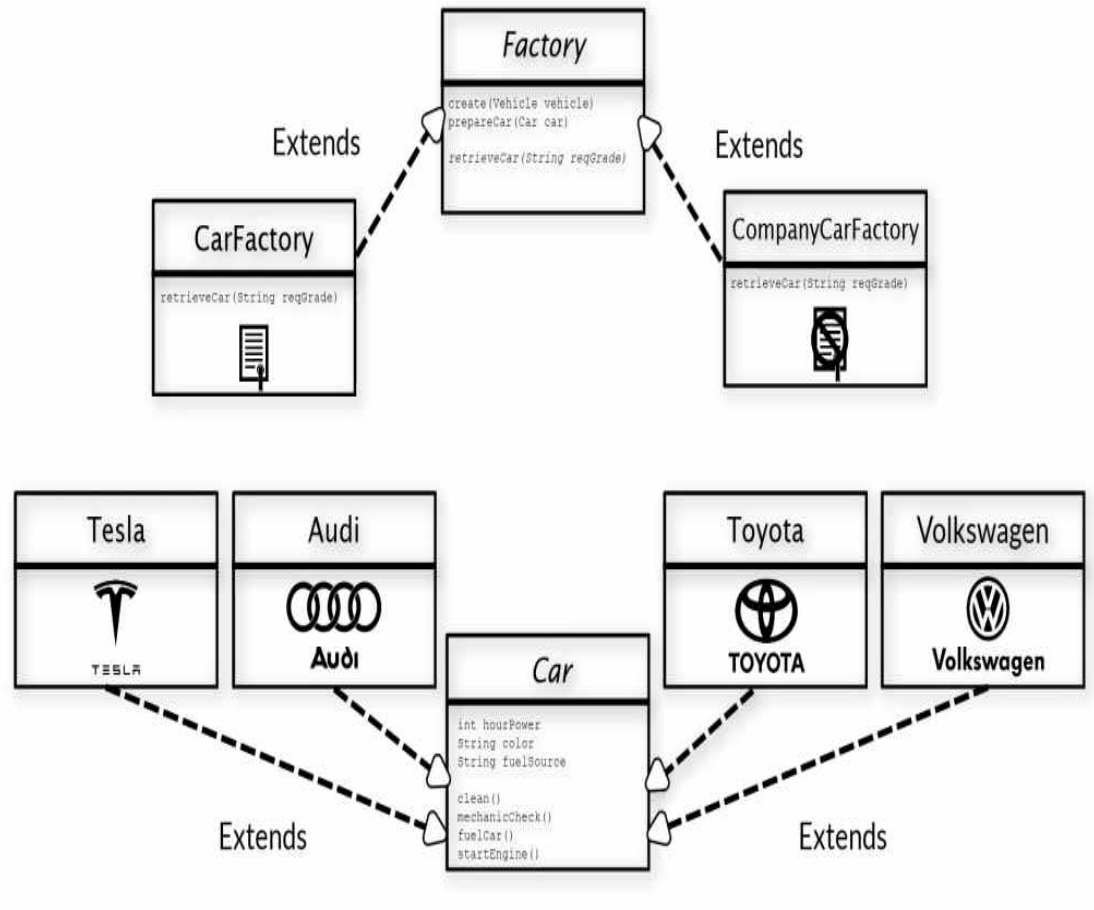
Concept Of The Factory Pattern



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/FactoryDesignPatternDemo (it's a eclipse source file)

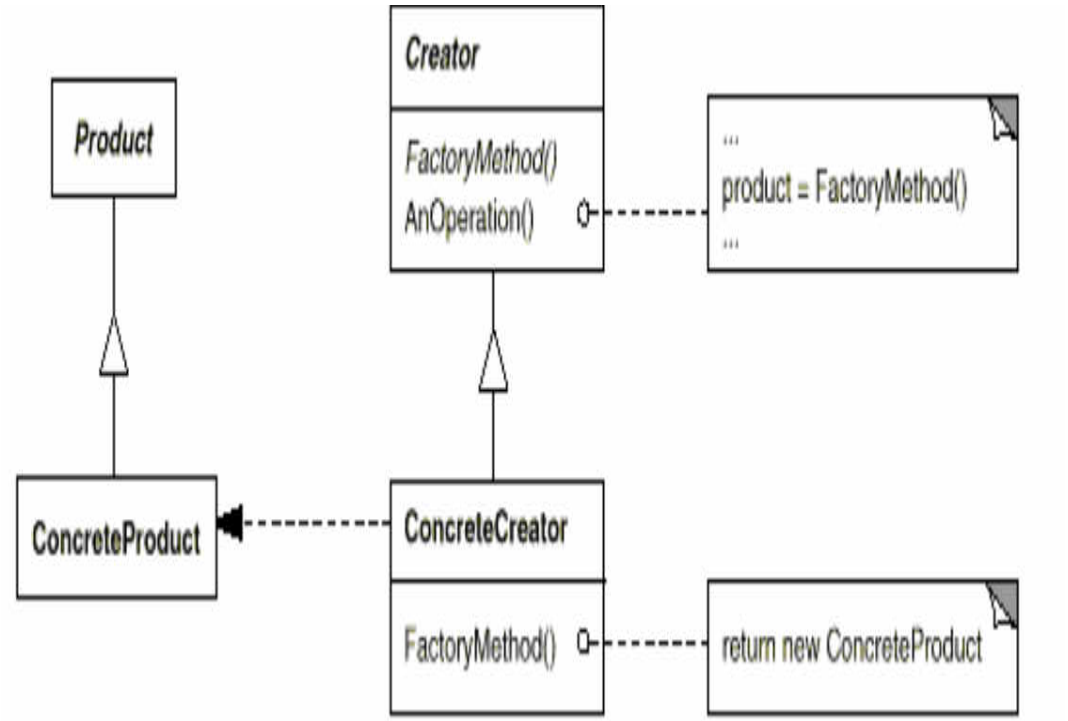
Applicability

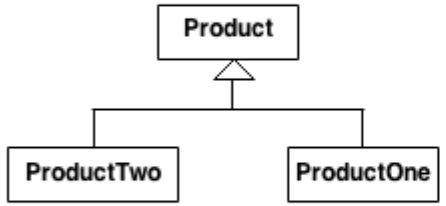
Use the Factory pattern when

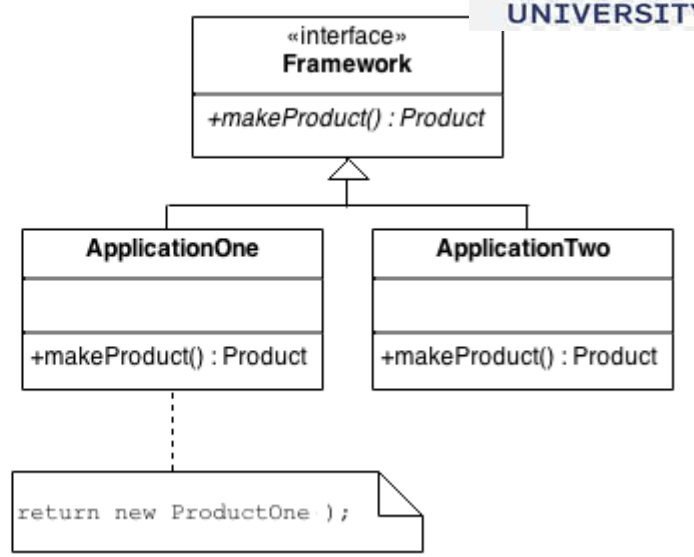
- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

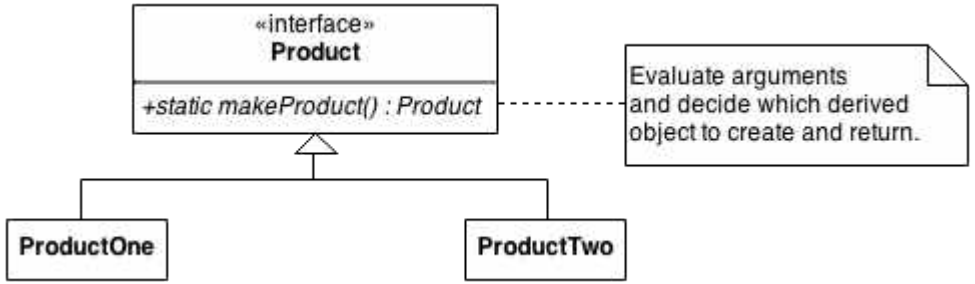
Object Oriented Analysis and Design with Java

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.

Relationship with other pattern

⇄ Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.

- You can use **Factory Method** along with **Iterator** to let collection subclasses return different types of iterators that are compatible with the collections.

- **Prototype** isn't based on inheritance, so it doesn't have its drawbacks. On the other hand, *Prototype* requires a complicated initialization of the cloned object. **Factory Method** is based on inheritance but doesn't require an initialization step.

- **Factory Method** is a specialization of **Template Method**. At the same time, a *Factory Method* may serve as a step in a large *Template Method*.

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.

Object Oriented Analysis and Design with Java

Demo

Link for java program

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 Nivedita Kasturi

Department of Computer Science and Engineering

niveditak@pes.edu

Object Oriented Analysis and Design with Java

UE20CS352

Prof Nivedita Kasturi
Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design with Java

OO Design Patterns

Prof Nivedita Kasturi

Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design with Java

Unit-4

Creational Patterns – Builder

Agenda

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

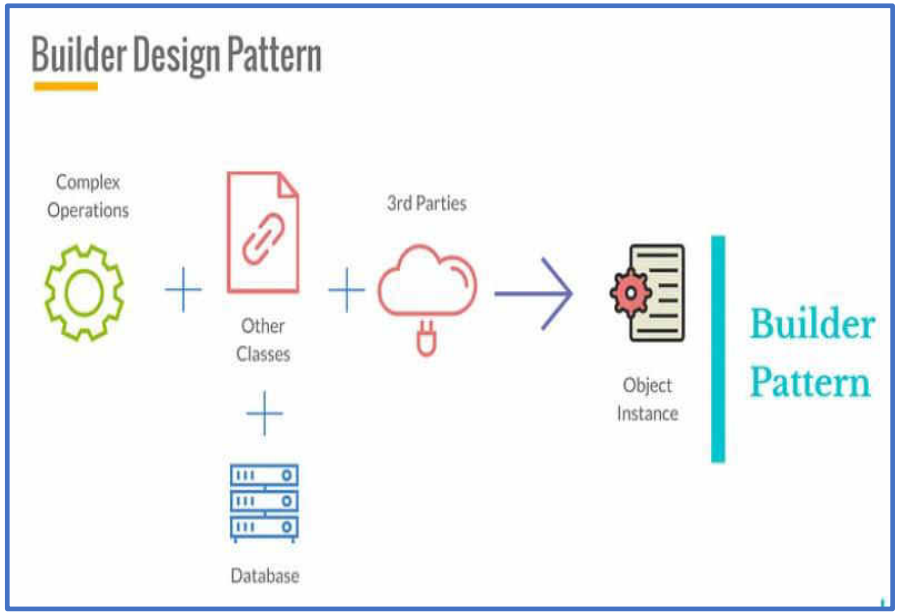
Introduction

- **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

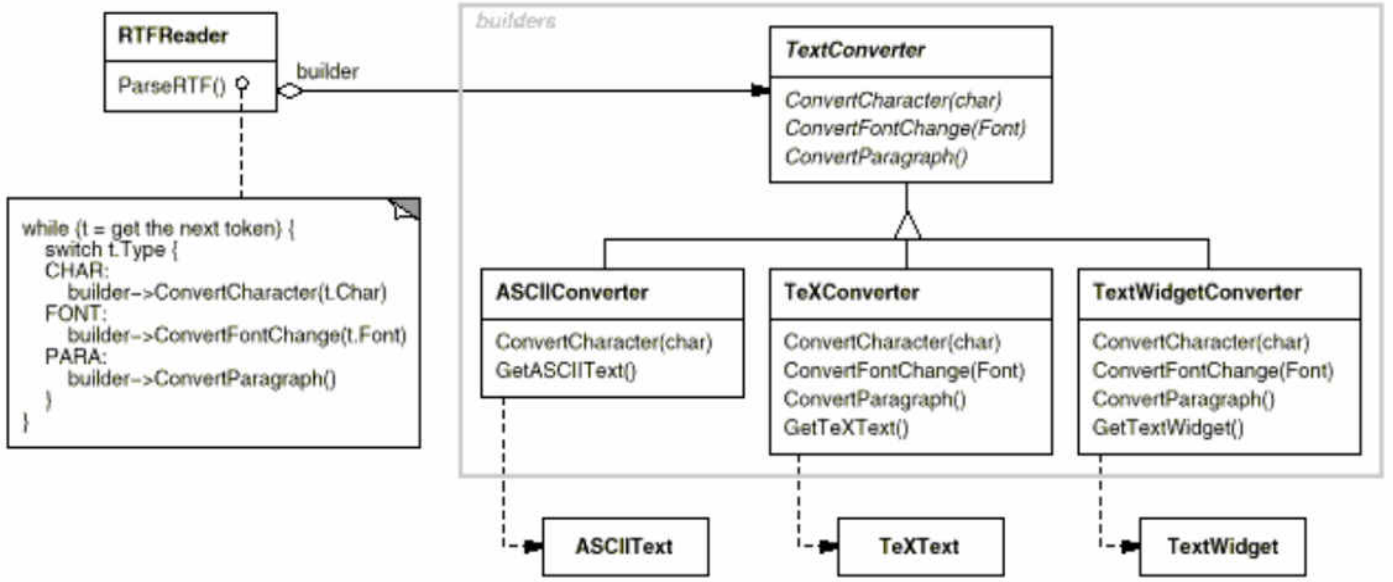
Motivation

- 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.

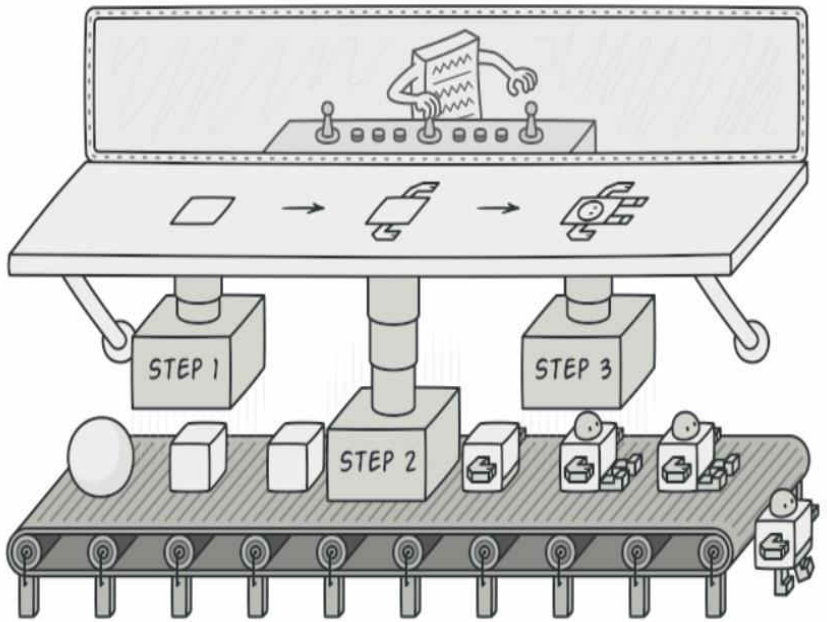
- The Builder pattern captures all these relationships. **Each converter class is called a builder** in the pattern, and the **reader is called the director.**



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



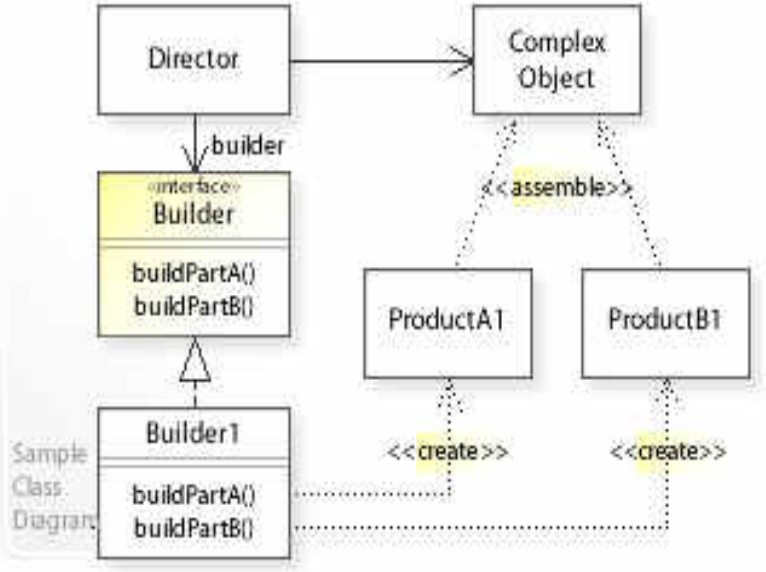
Think of a car factory

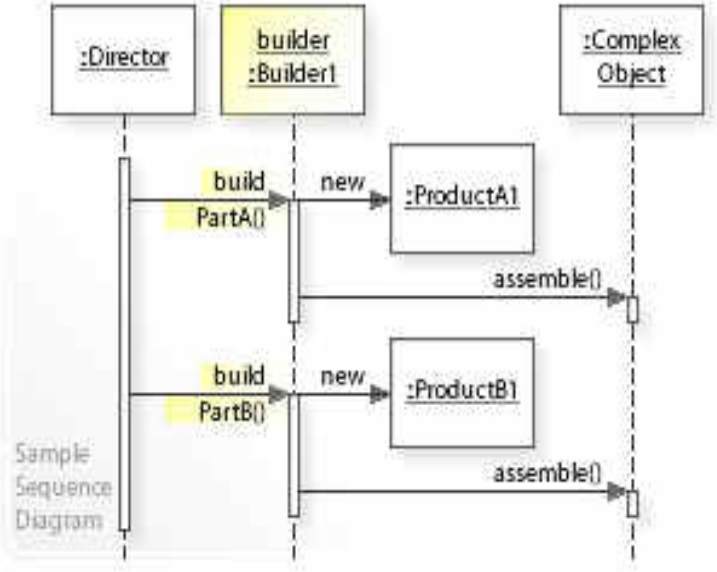
Boss tells workers (or robots) to build each part of a car
Workers build each part and add them to the car being constructed

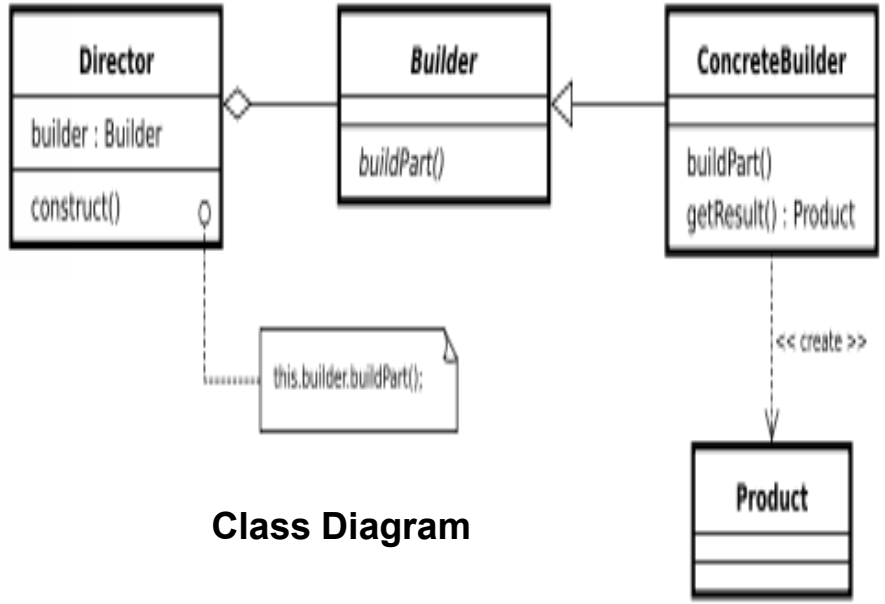
Object Oriented Analysis and Design with Java

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.

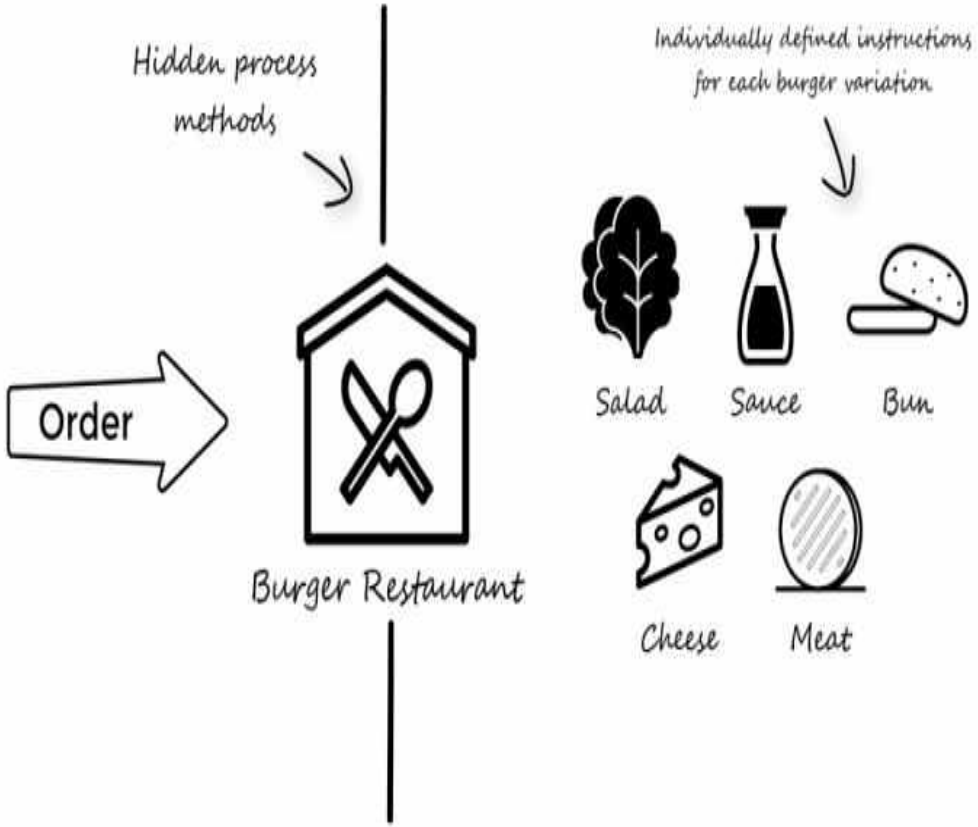
Builder : Implementation example-1

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.

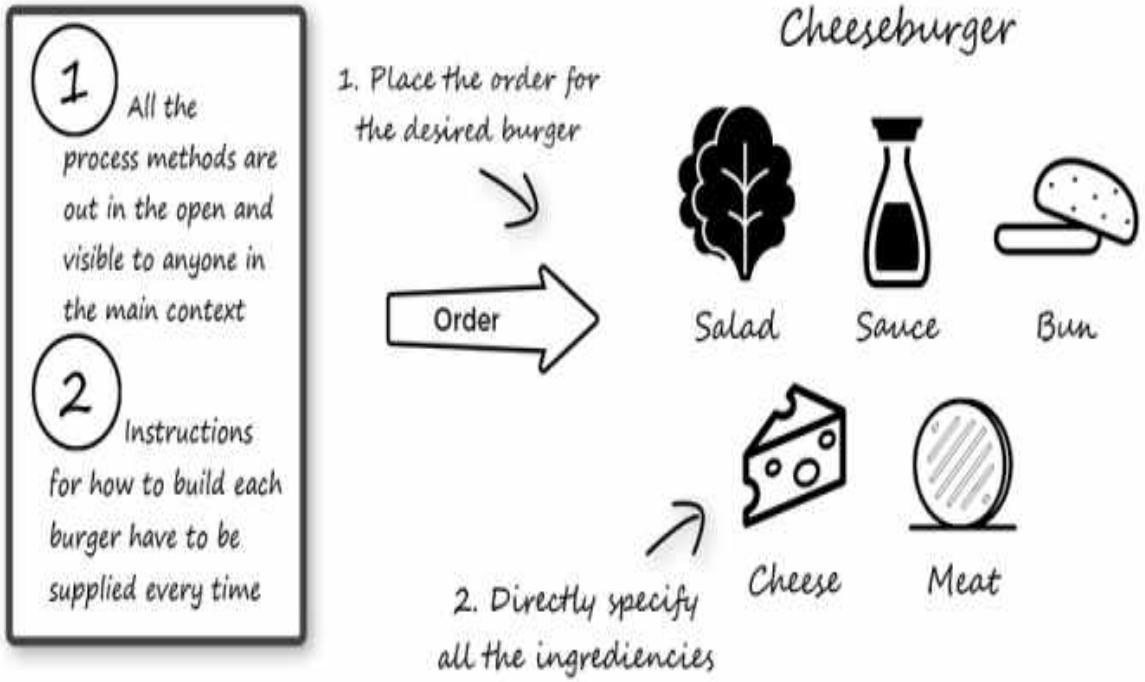
- Before, how the builder design pattern will be able to solve these issues, we will start by seeing how this system could be build based on construction of the burger object in the main context.



Object Oriented Analysis and Design with Java

Builder : Implementation example-1

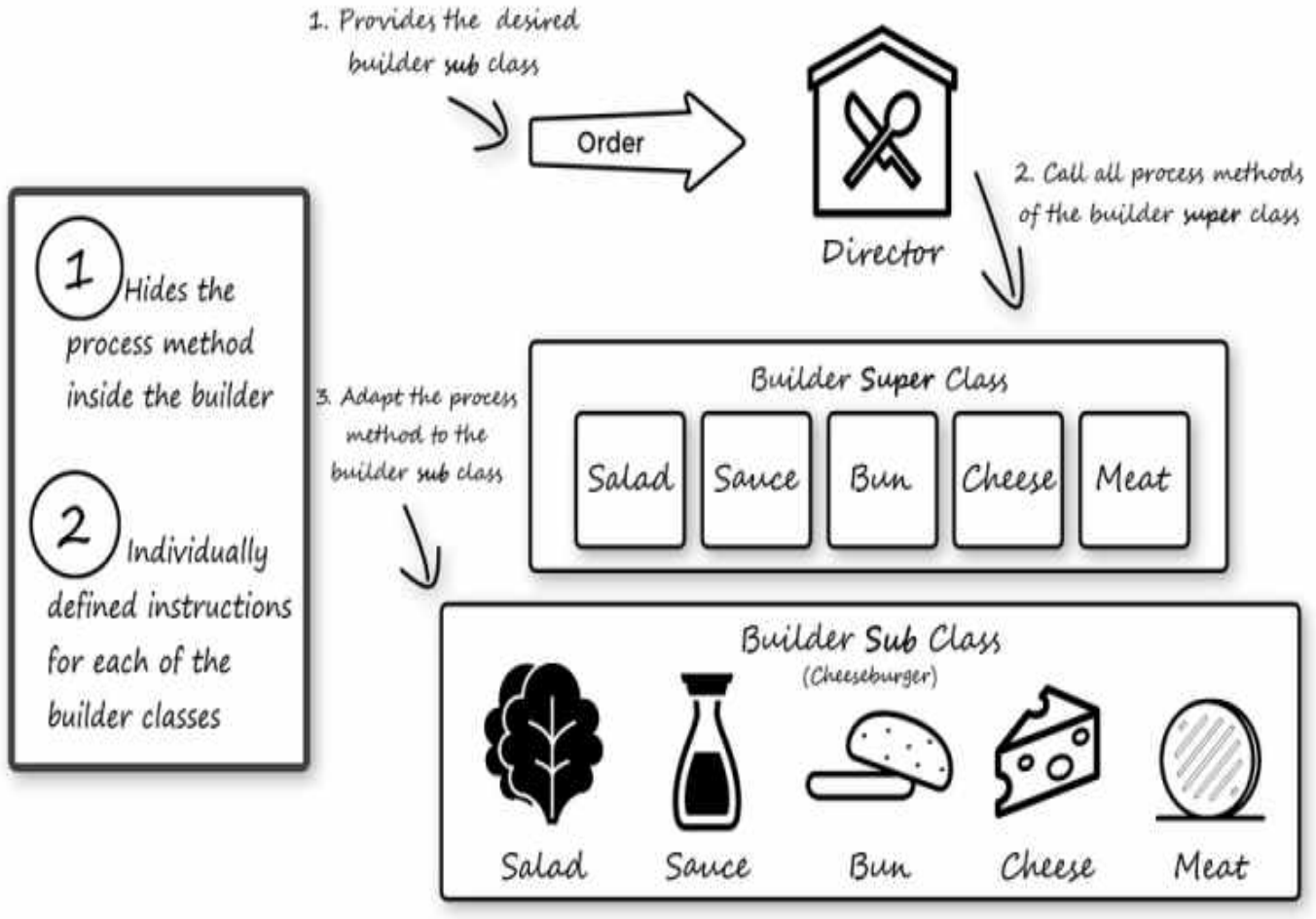
Solution: Construction In The Main Context




```
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();
    }
```

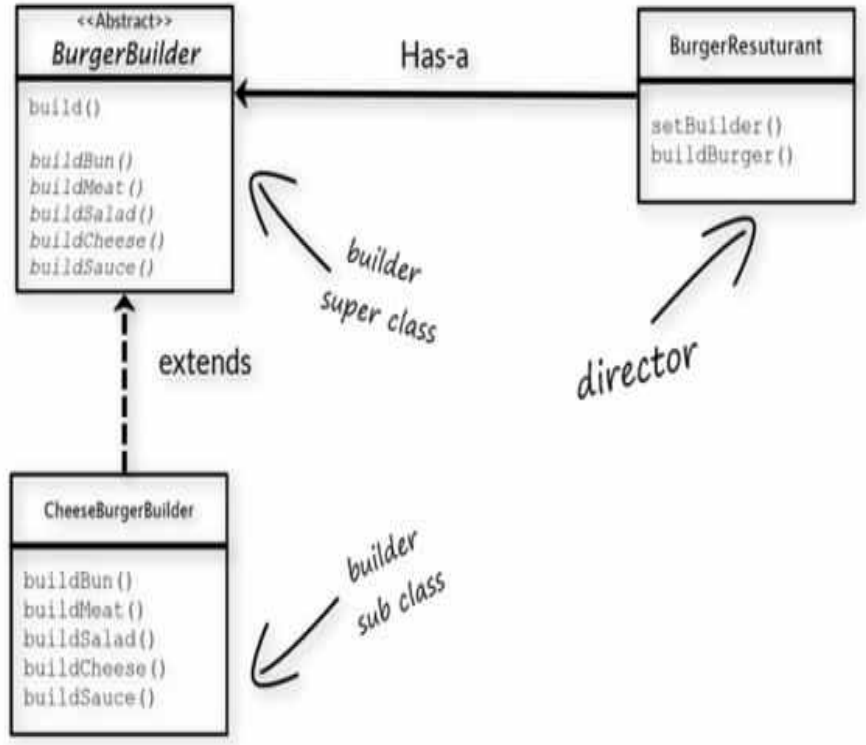
Solution: Concept Of The Builder Pattern

- 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

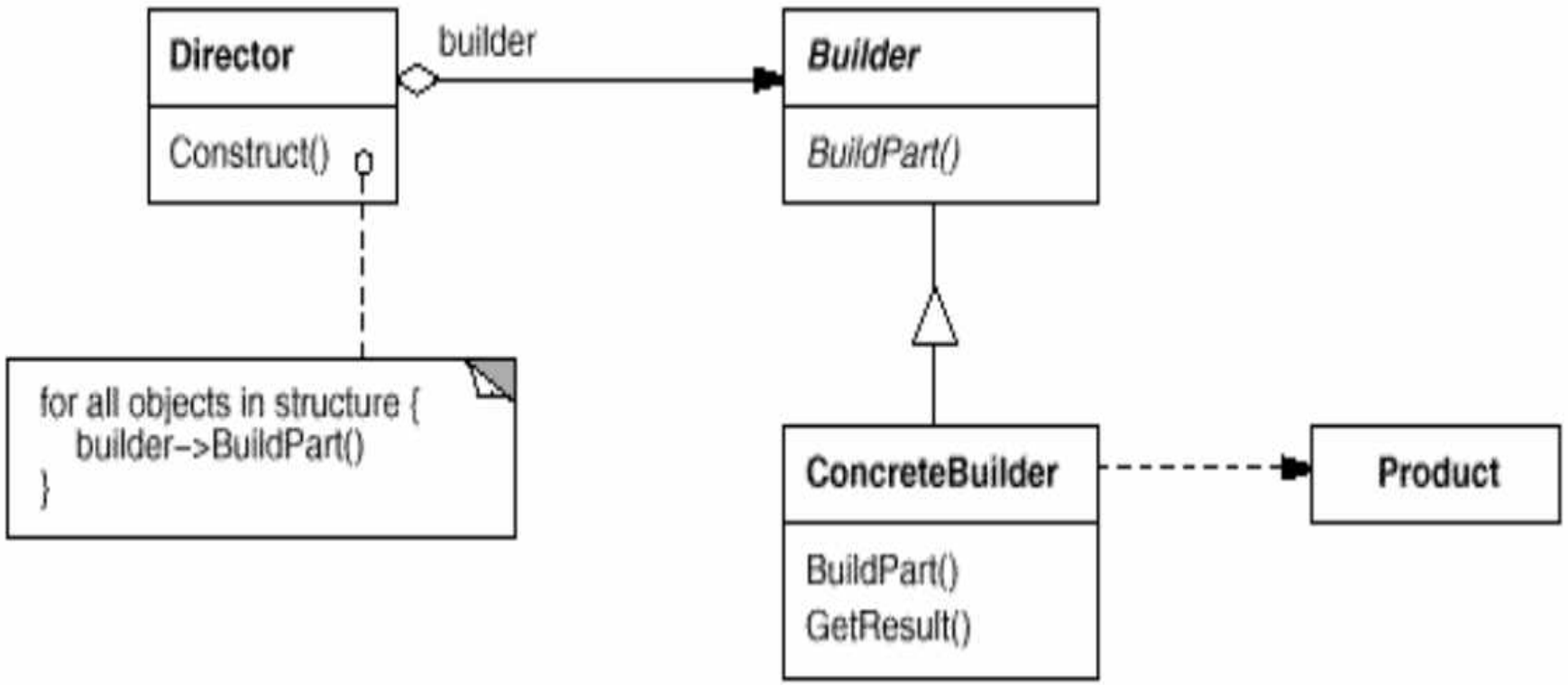
Note: Java files in the src/builderPatternDemo (it's a eclipse source file)

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). ·

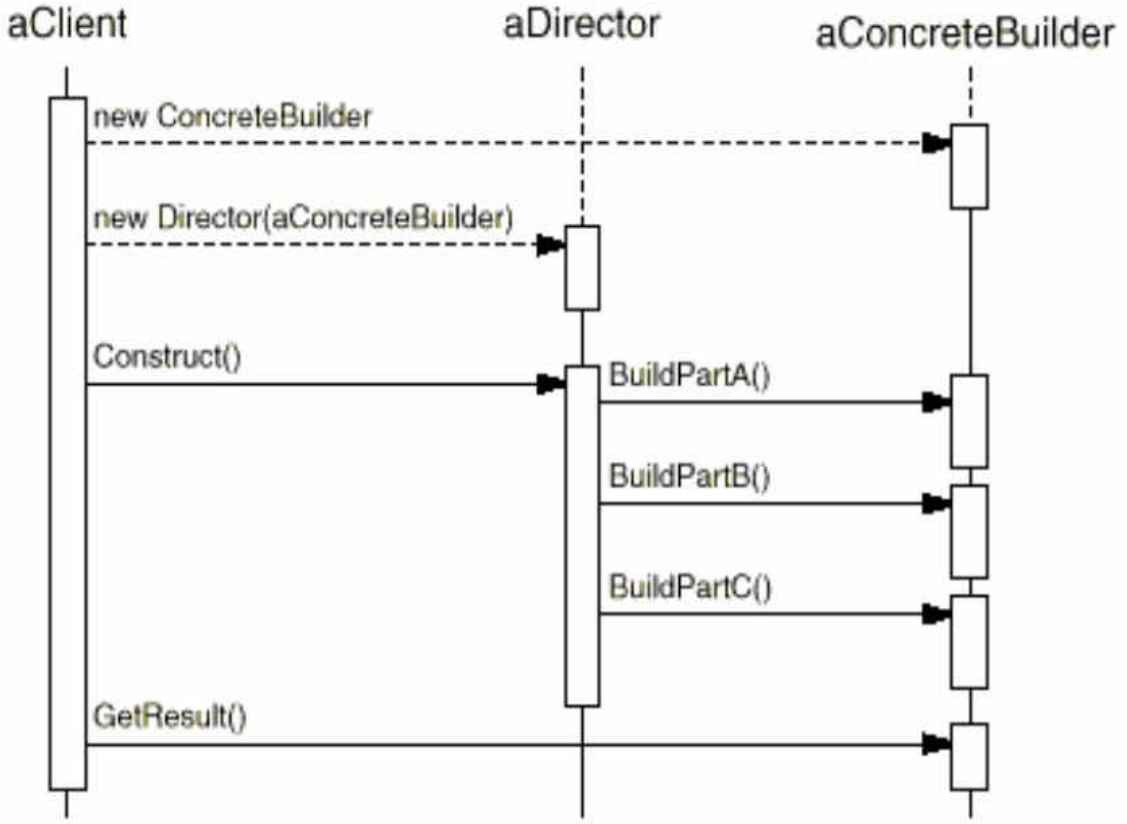
Director (RTFReader)
o constructs an object using the Builder interface.

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.

Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).

- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. *Abstract Factory* returns the product immediately, whereas *Builder* lets you run some additional construction steps before fetching the product.

- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively.

- You can combine **Builder** with **Bridge**: the director class plays the role of the abstraction, while different builders act as implementations.

- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

Demo

Link for builder demo

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

Prof Nivedita Kasturi

Department of Computer Science and Engineering

niveditak@pes.edu

Object Oriented Analysis and Design with Java

UE20CS352

Prof Nivedita Kasturi
Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design with Java

OO Design Patterns

Prof Nivedita Kasturi

Department of Computer Science and Engineering

UE20CS352: Object Oriented Analysis and Design with Java

Unit-4

Creational Patterns – Prototype

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

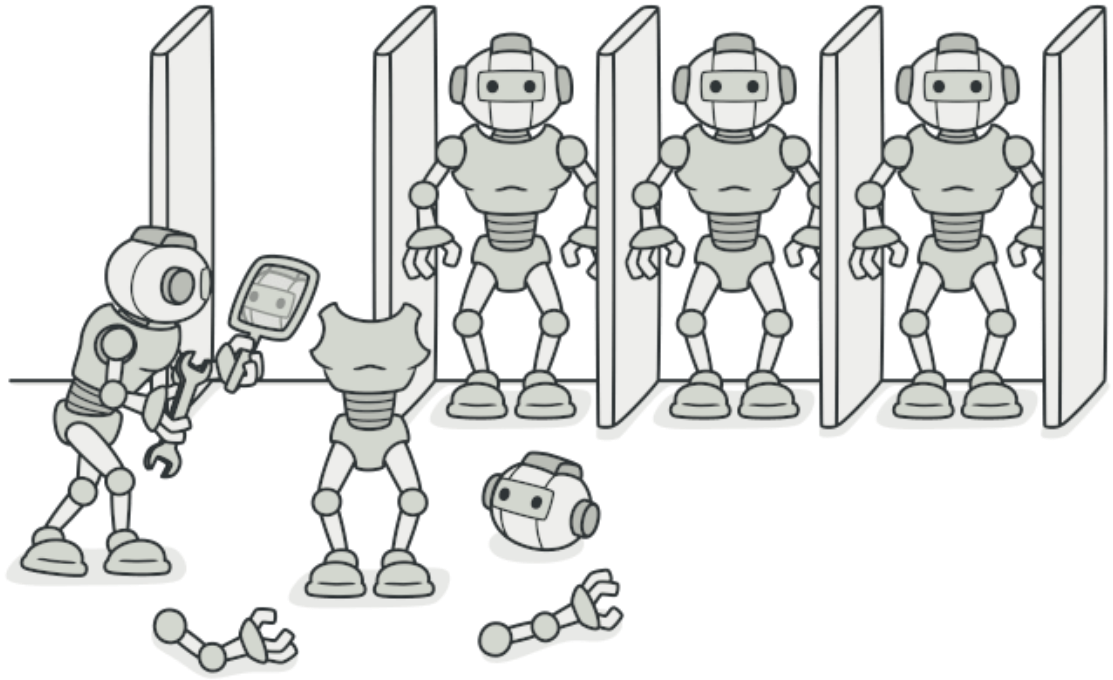
Prototype : Class, Object Structural

Intent

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

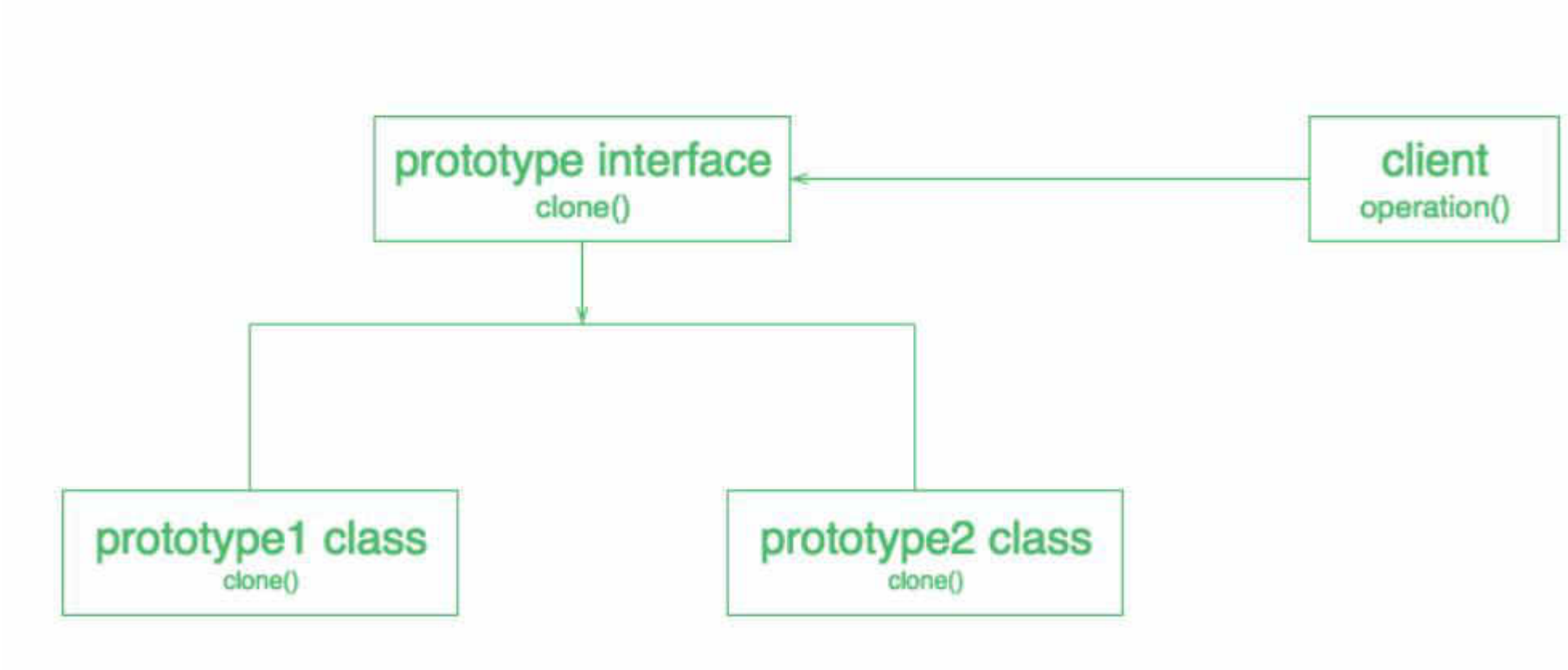
Usage examples: The Prototype pattern is available in Java out of the box with a Cloneable interface.
Any class can implement this interface to become cloneable.

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

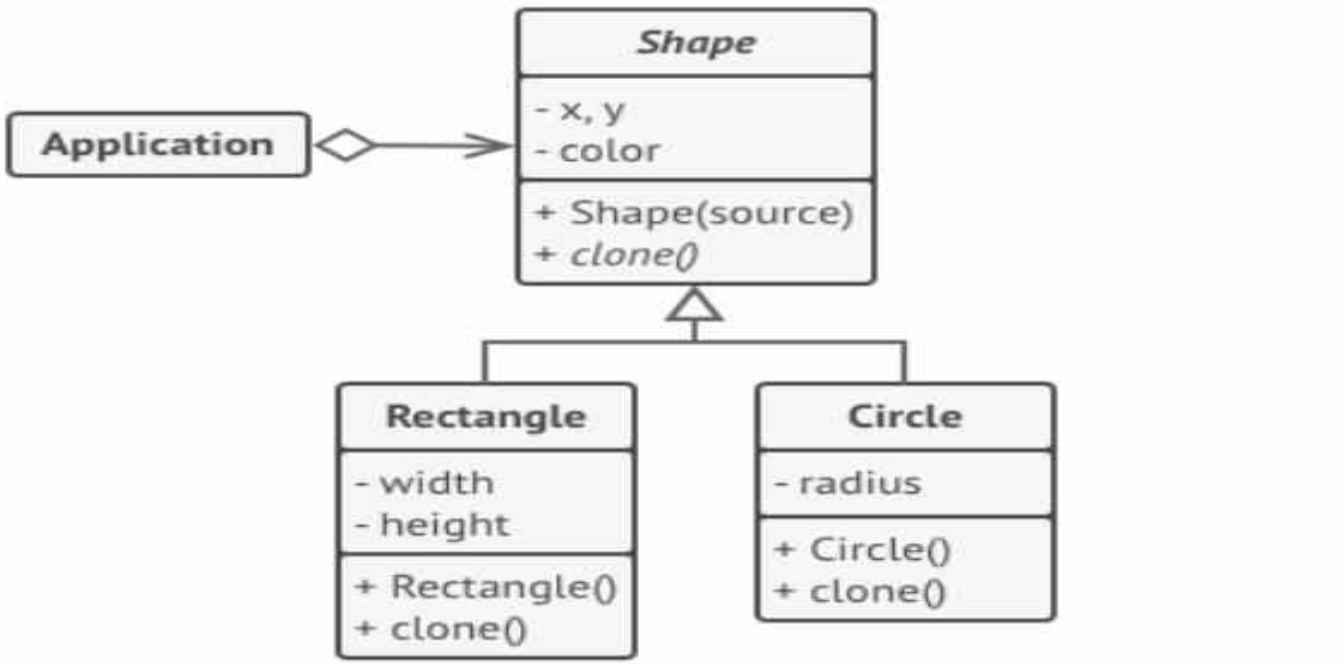
UML class diagram for the Prototype Pattern



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.

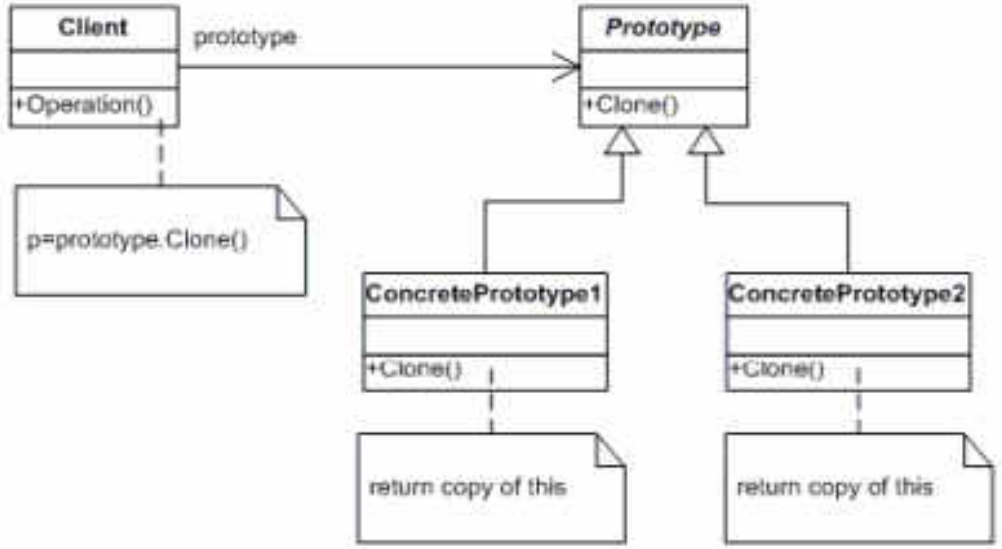
Structure

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

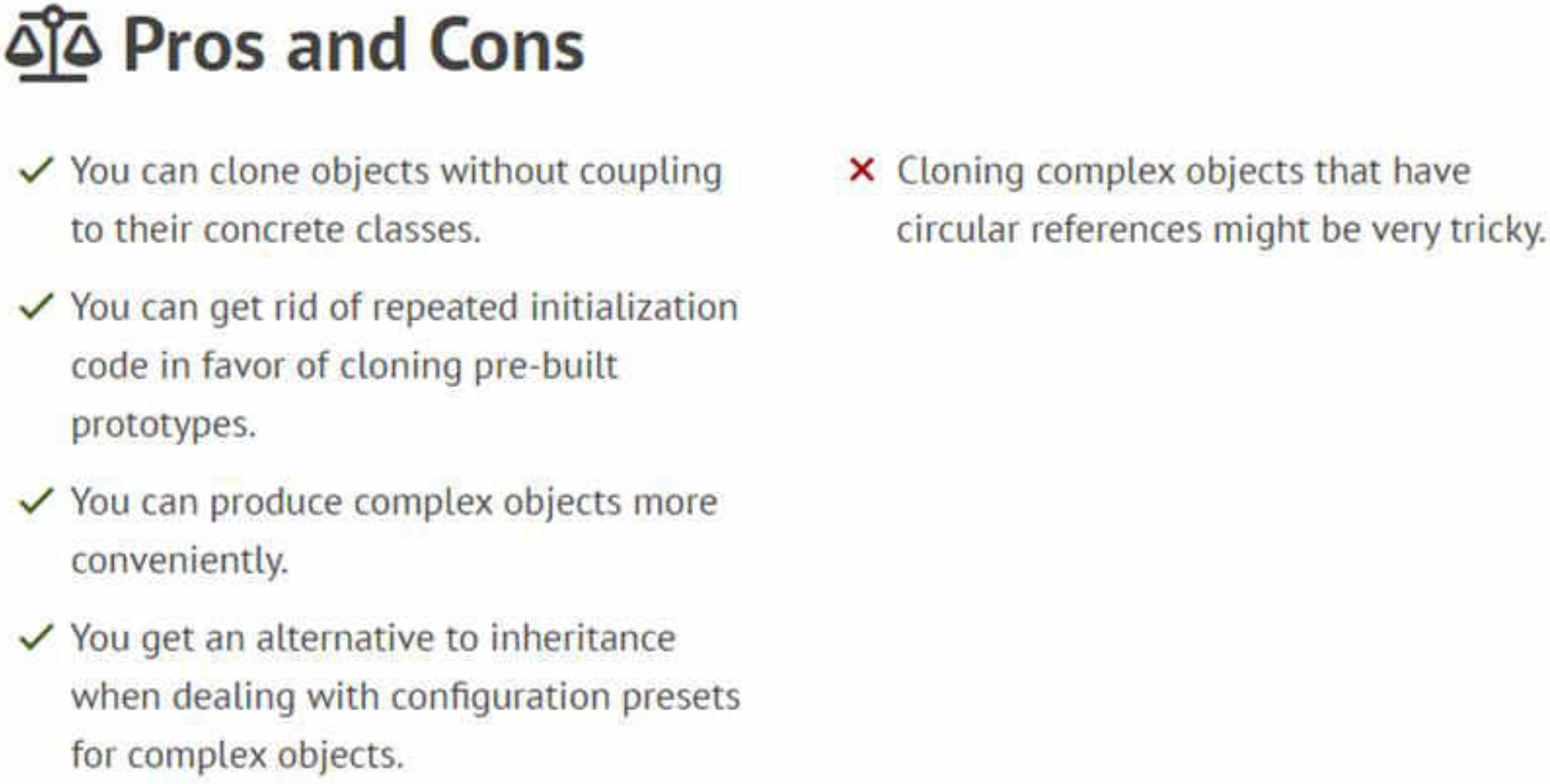
Consider the following issues when implementing prototypes:

- **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

Prof Nivedita Kasturi

Department of Computer Science and Engineering

niveditak@pes.edu

Object Oriented Analysis and Design with Java

UE20CS352

Dr. Sudeepa Roy Dey & Saranya devi B
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

Unit 4
Structural Patterns – Adapter

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

Agenda

- ❑ Introduction to Structural design patterns
- ❑ Types of Structural Design Patterns.
- ❑ Adapter-definition
- ✔ Motivation
- ✔ Intent
- ✔ Implementation
- ✔ Applicability
- ✔ Structure-Consequence
- ✔ Issues

Introduction to Structural Design Pattern

- Structural design patterns are concerned with how classes and objects can be composed, to form larger structures.
- The structural design patterns simplifies the structure by identifying the relationships.
- These patterns focus on, how the classes inherit from each other and how they are composed from other classes.
- Structural *class* patterns use inheritance to compose interfaces or implementations.
- Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

Types of Structural 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)

Adapter: Class, Object Structural

Motivation

- The adapter pattern is adapting between classes and objects.
- Like any adapter in the real world it is used to be an interface, a bridge between two objects. Ex: In real world we have adapters for power supplies, adapters for camera memory cards, and so on.
- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Advantage of Adapter Pattern

- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

Adapter: Class, Object Structural

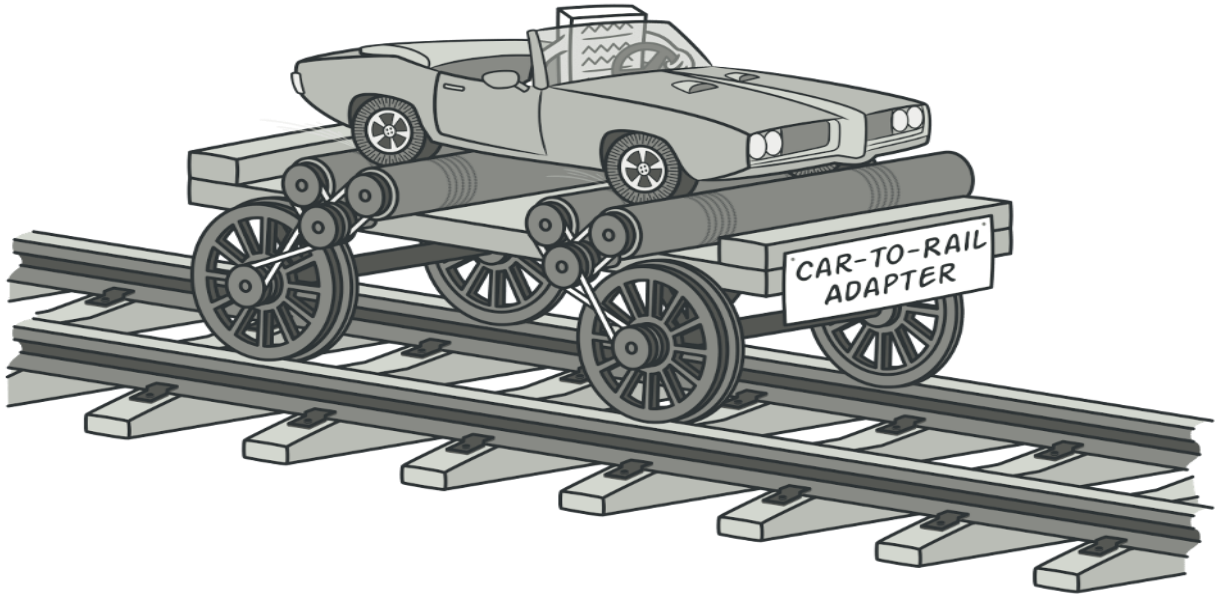
Intent

- Adapter lets classes work together that couldn' t otherwise because of incompatible interfaces.

- An Adapter Pattern says that "convert the interface of a class into another interface that a client wants". In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

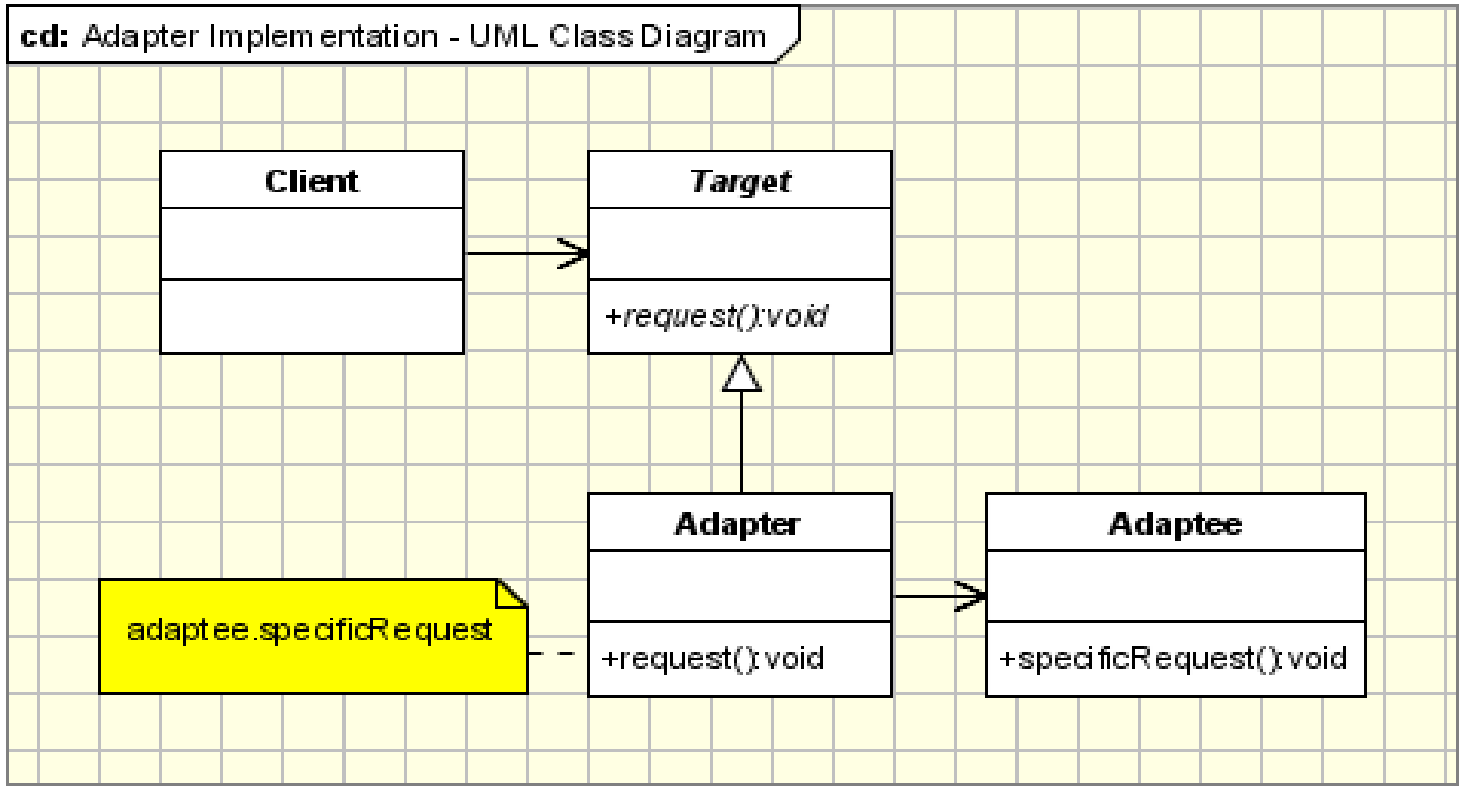
- The Adapter Pattern is also known as Wrapper.

Adapter is a structural design pattern that allows classes with incompatible interfaces to collaborate.



Adapter: Implementation

UML class diagram for the Adapter Pattern



The classes/objects participating in adapter pattern:

1. Target - defines the domain-specific interface that Client uses.
2. Adapter - adapts the interface Adaptee to the Target interface.
3. Adaptee - defines an existing interface that needs adapting.
4. Client - collaborates with objects conforming to the Target interface.

Adapter class: This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.

Object Oriented Analysis and Design with Java

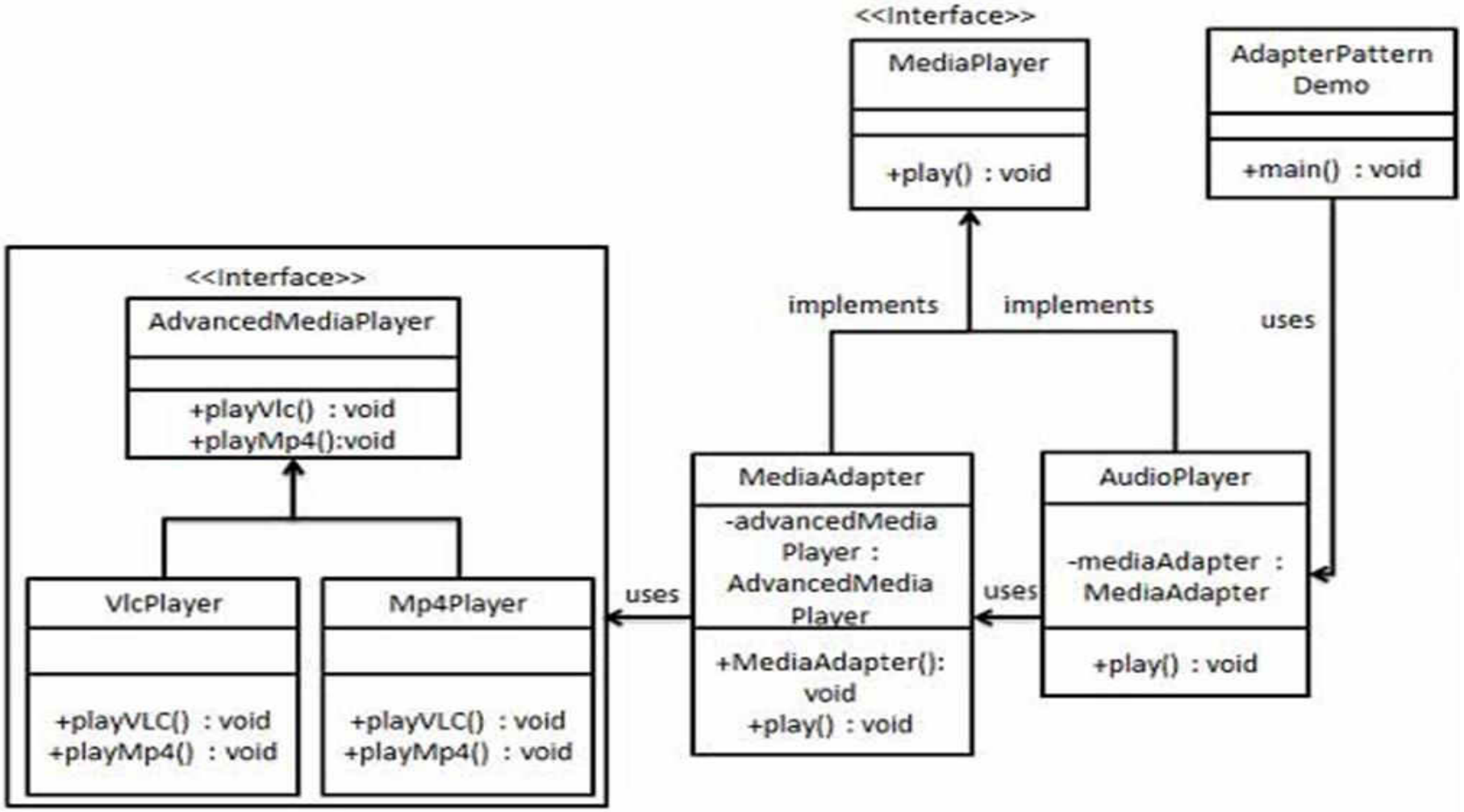
Adapter: Implementation example-1

Problem Statement: We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default. We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files. We want to make AudioPlayer to play other formats as well.

Solution: To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format. AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format. AdapterPatternDemo, our demo class will use AudioPlayer class to play various formats.

Demo….>refer to AdapterPatternDemo.java

Design Solution



Design Solution

Adapter Pattern

Modules :

Module Type	Module Example
Adapter (concrete of AbstractA)	*MediaAdapter*
Adaptee (concrete of AbstractA)	*AudioPlayer*
AbstractB	*AdvanceMediaPlayer*
ConcreteB	*VLCplayer, MP4player*
AbstactA	*MediaPlayer*
Demo	*AdapterPatternDemo*

Adapter: Scenario example-2

Suppose you have a Bird class with fly() and makeSound()methods. And also a ToyDuck class with squeak() method. Let's assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can't use them directly.

Solution: So, we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.(refer to demo main.java

The adapter pattern we have implemented above is called Object Adapter Pattern because the adapter holds an instance of adaptee.
What a Object adapter??? Next slides we will explain in detail

Adapter: Scenario example-2

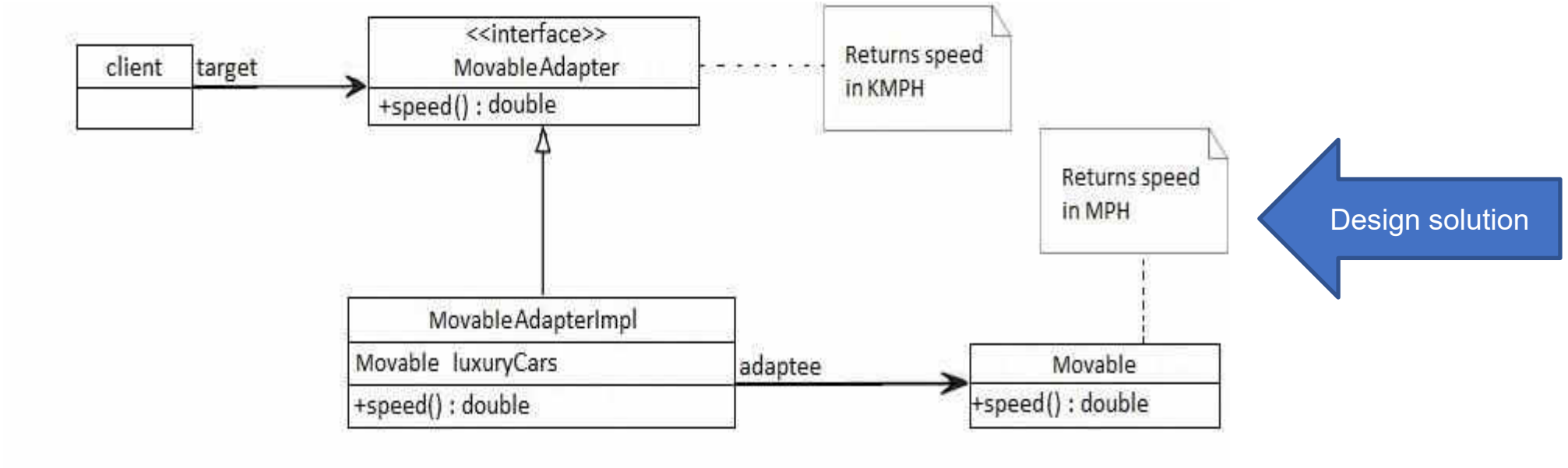
Explain: Here we have created a BirdAdapter class implementing inteface ToyDuck to convert Bird 's makeSound() method to makeSound() 's squeak() ToyDuck . This way, we can still call the squeak() of Bird objects through BirdAdapter

Suppose we have a bird that can makeSound(), and we have a plastic toy duck that can squeak(). Now suppose our client changes the requirement and he wants the toyDuck to makeSound then ?

Simple solution is that we will just change the implementation class to the new adapter class and tell the client to pass the instance of the bird(which wants to squeak()) to that class.
Before : ToyDuck toyDuck = new PlasticToyDuck();
After : ToyDuck toyDuck = new BirdAdapter(sparrow);
You can see that by changing just one line the toyDuck can now do Chirp Chirp !!

Adapter: Scenario example-3

Consider a scenario in which there is an app that's developed in the US which returns the top speed of luxury cars in miles per hour (MPH). Now we need to use the same app for our client in the UK that wants the same results but in kilometers per hour (km/h).
To deal with this problem, we'll create an adapter which will convert the values and give us the desired results:



(Students can Try coding with the given design solution)

Applicability

Use the Adapter pattern when

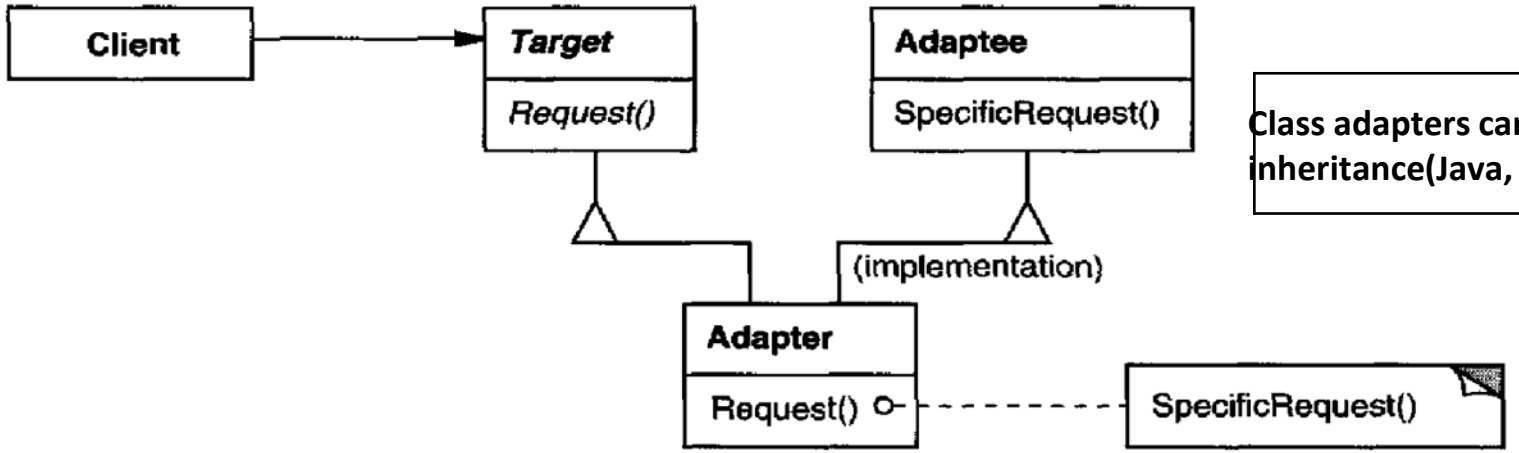
- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (object adapter only) you need to use several existing subclasses, but it's unpractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Software Examples of Adapter Patterns: Wrappers used to adopt 3rd parties libraries and frameworks - most of the applications using third party libraries use adapters as a middle layer between the application and the 3rd party library to decouple the application from the library.

Object Oriented Analysis and Design with Java

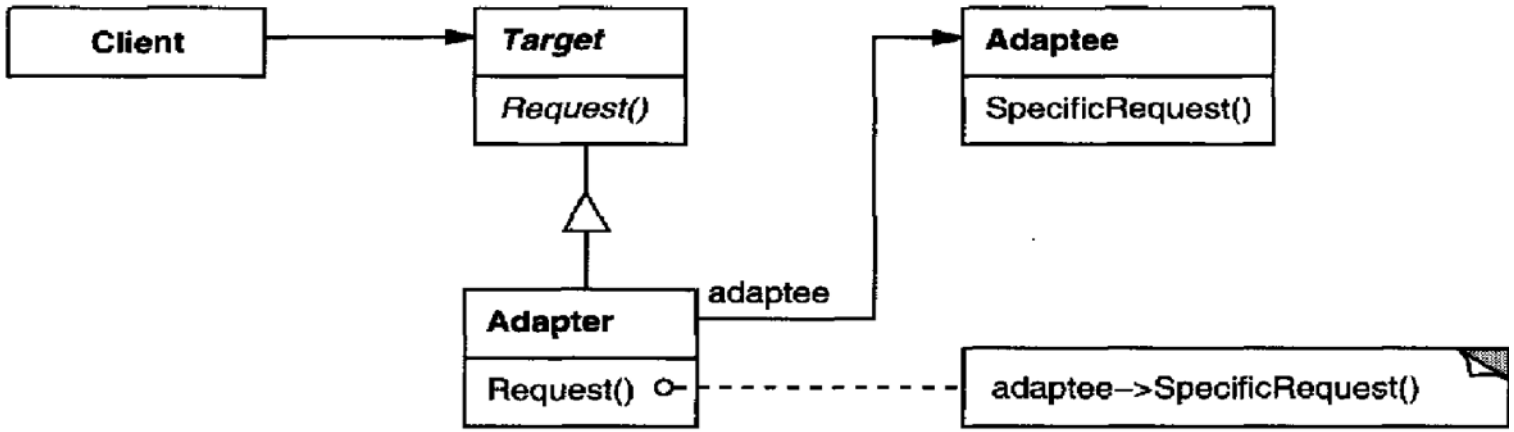
Structure

A class adapter uses multiple inheritance to adapt one interface to another:



Class adapters can be implemented in languages supporting multiple inheritance(Java, C# or PHP does not support multiple inheritance)

An object adapter relies on object composition: Based on delegation



Consequence

Class and object adapters have different trade-offs.

A class adapter

- adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- let's Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

- **let's a single Adapter work with many Adaptees**—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Difference: Class and Object structure pattern

- Objects Adapters uses composition, the Adaptee delegates the calls to Adaptee (opossed to class adapters which extends the Adaptee).
- The main advantage is that the object Adapter adapts not only the Adaptee but all its subclasses. All it's subclasses with one "small" restriction: all the subclasses which don't add new methods, because the used mechanism is delegation. So for any new method the Adapter must be changed or extended to expose the new methods as well.
- The main disadvantage is that it requires to write all the code for delegating all the necessary requests to the Adaptee.

- Class adapter uses inheritance instead of composition. It means that instead of delegating the calls to the Adaptee, it subclasses it. In conclusion it must subclass both the Target and the Adaptee.
- There are advantages and disadvantages: It adapts the specific Adaptee class. The class it extends. If that one is subclassed it can not be adapted by the existing adapter.
- It doesn't require all the code required for delegation, which must be written for an Object Adapter.
- If the Target is represented by an interface instead of a class then we can talk about "class" adapters, because we can implement as many interfaces as we want.

Issues to consider when using the Adapter pattern

- *How Much the Adapter Should Do?*

It should do how much it has to in order to adapt. It's very simple, if the Target and Adaptee are similar then the adapter has just to delegate the requests from the Target to the Adaptee. If Target and Adaptee are not similar, then the adapter might have to convert the data structures between those and to implement the operations required by the Target but not implemented by the Adaptee.

- *Using two-way adapters to provide transparency.*

A potential problem with adapters is that they aren't transparent to all clients. An adapted object no longer conforms to the Adaptee interface, so it can't be used as is wherever an Adaptee object can. Two-way adapters can provide such transparency. Specifically, they're useful when two different clients need to view an object differently.

References

Text Reference

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

Web Reference

- https://www.javatpoint.com/adapter-pattern
- https://www.oodesign.com/adapter-pattern.html
- https://www.geeksforgeeks.org/adapter-pattern/
- https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm
- https://itzone.com.vn/en/article/adapter-design-pattern-in-java

THANK YOU

Dr. Sudeepa Roy Dey and Saranya devi b

Department of Computer Science and Engineering

sudeepar@pes.edu

Object Oriented Analysis and Design with Java

UE20CS352

Sindhu R Pai, Saranya Devi B

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

UE20CS352 : Object Oriented Analysis and Design with Java

Structural Patterns : Facade pattern

Prof. Sindhu R Pai
Department of Computer Science and Engineering

Agenda

- What is façade?
- Why Façade?
- Pictorial Representation
- Applicability
- Advantages
- Known Uses
- Implementation
- References

What is facade?

Meaning:

- The principal front of a building, that faces on to a street or open space.

  "the house has a half-timbered façade"

- A deceptive outward appearance.

  "her flawless public façade masked private despair"

Facade in Software

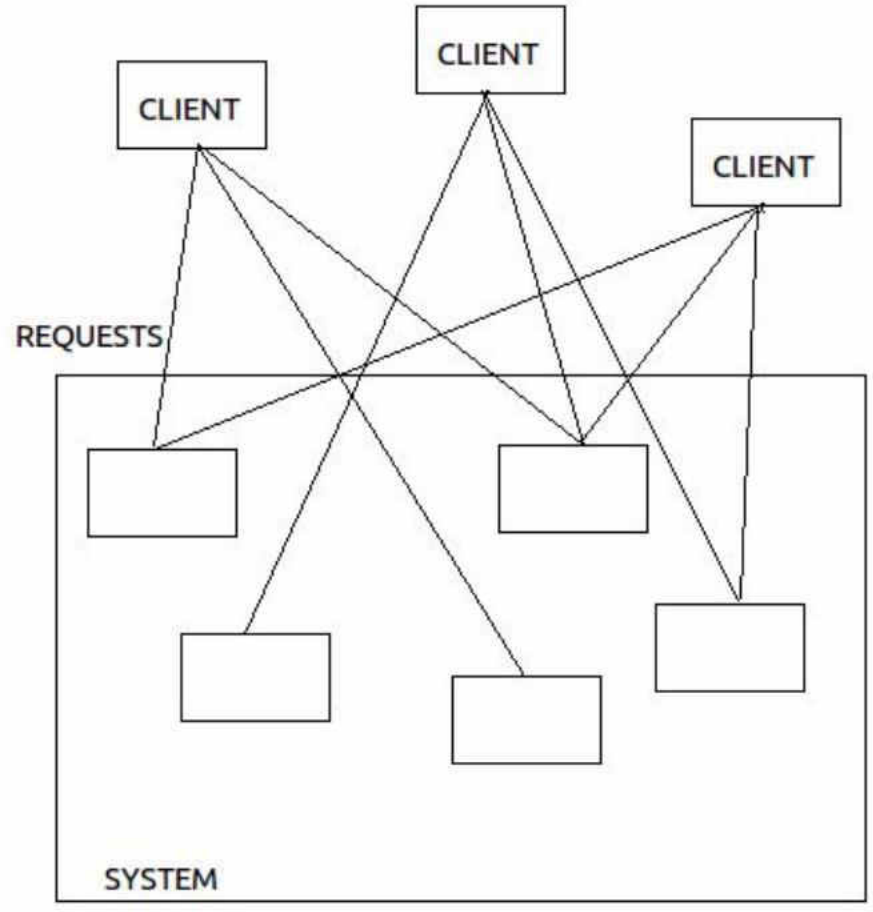
- An object that provides a **simplified interface to a larger body** of code, a library, a framework, or any other complex set of classes.
- The main intent of Façade is to provide a **unified interface to a set of interfaces in a subsystem**. Façade defines a **higher-level interface** that makes the subsystem easier to use.
- Example: fopen is an interface to open and read system commands.
- A structural design pattern, which **wraps a complicated subsystem with a simpler interface.**
- If the Facade is the only access point for the subsystem, it will limit the features and flexibility that **"power users**" may need

Why Facade?

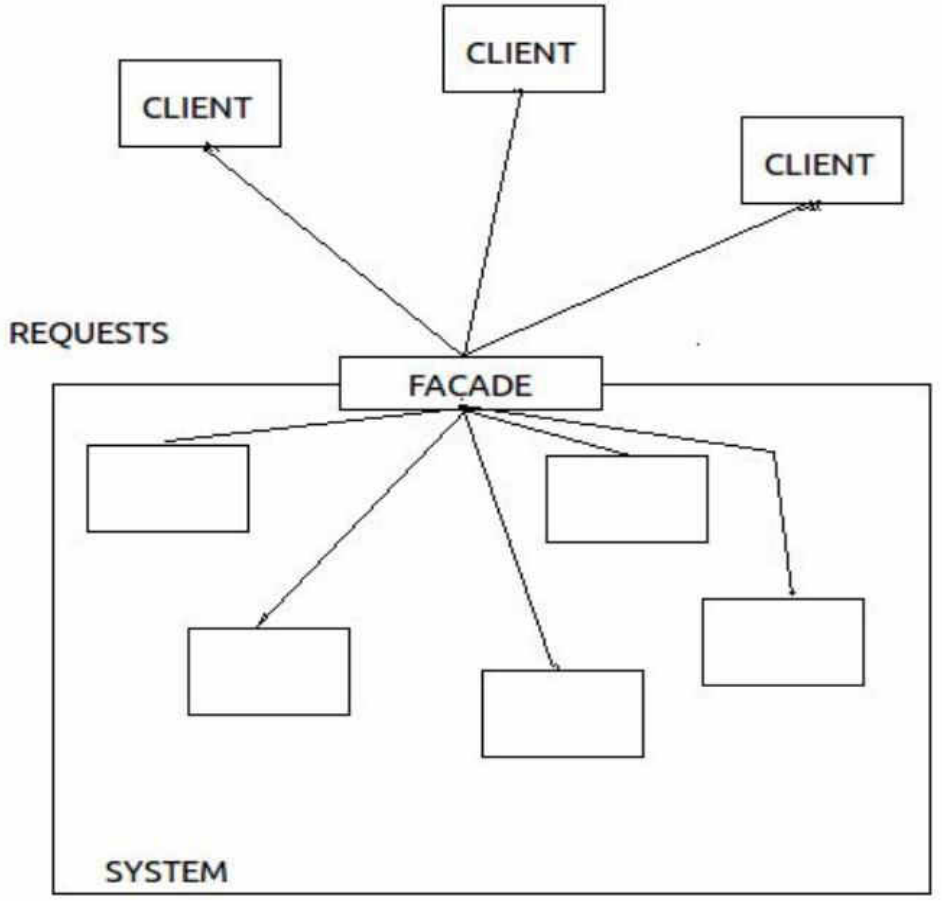
- Structuring a system into subsystems **helps reduce complexity**.

- A common design goal is to **minimize the communication and dependencies between subsystems**. One way to achieve this goal is to introduce a façade object that provides a single, simplified interface to the more general facilities of a subsystem.

Pictorial representation



Without facade



With facade

Applicability

- To provide a **simple interface to a complex subsystem**. A façade can provide **a simple default view of the subsystem** that is good enough for most clients.
- Introduce a façade to **decouple the subsystem from clients and other subsystems**, thereby promoting **subsystem independence and portability**.
- Use a façade to **define an entry point to each subsystem** level.
- To **simplify the dependencies between subsystems** by making them communicate with each other solely through their façades.

Advantages

- It **shields clients from subsystem components**, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- It **promotes weak coupling between the subsystem and its clients**. Often the components in a subsystem are strongly coupled.
- This can **eliminate complex or circular dependencies**. This can be an important consequence when the client and the subsystem are implemented independently.
- Reducing compilation dependencies with façades **can limit the recompilation needed for a small change** in an important subsystem.

Known uses

- In the **ET++ application framework**, an application that can have built-in browsing tools for inspecting its objects at run-time. These browsing tools are implemented in a separate subsystem that includes a façade class called "ProgrammingEnvironment." This façade defines operations such as InspectObject and InspectClass for accessing the browsers.

- The **Choices operating system** uses façade to compose many frameworks into one. The key abstractions in Choices are processes, storage, and address spaces. For each of these abstractions there is a corresponding subsystem, implemented as a framework, that supports porting Choices to a variety of different hardware platforms.

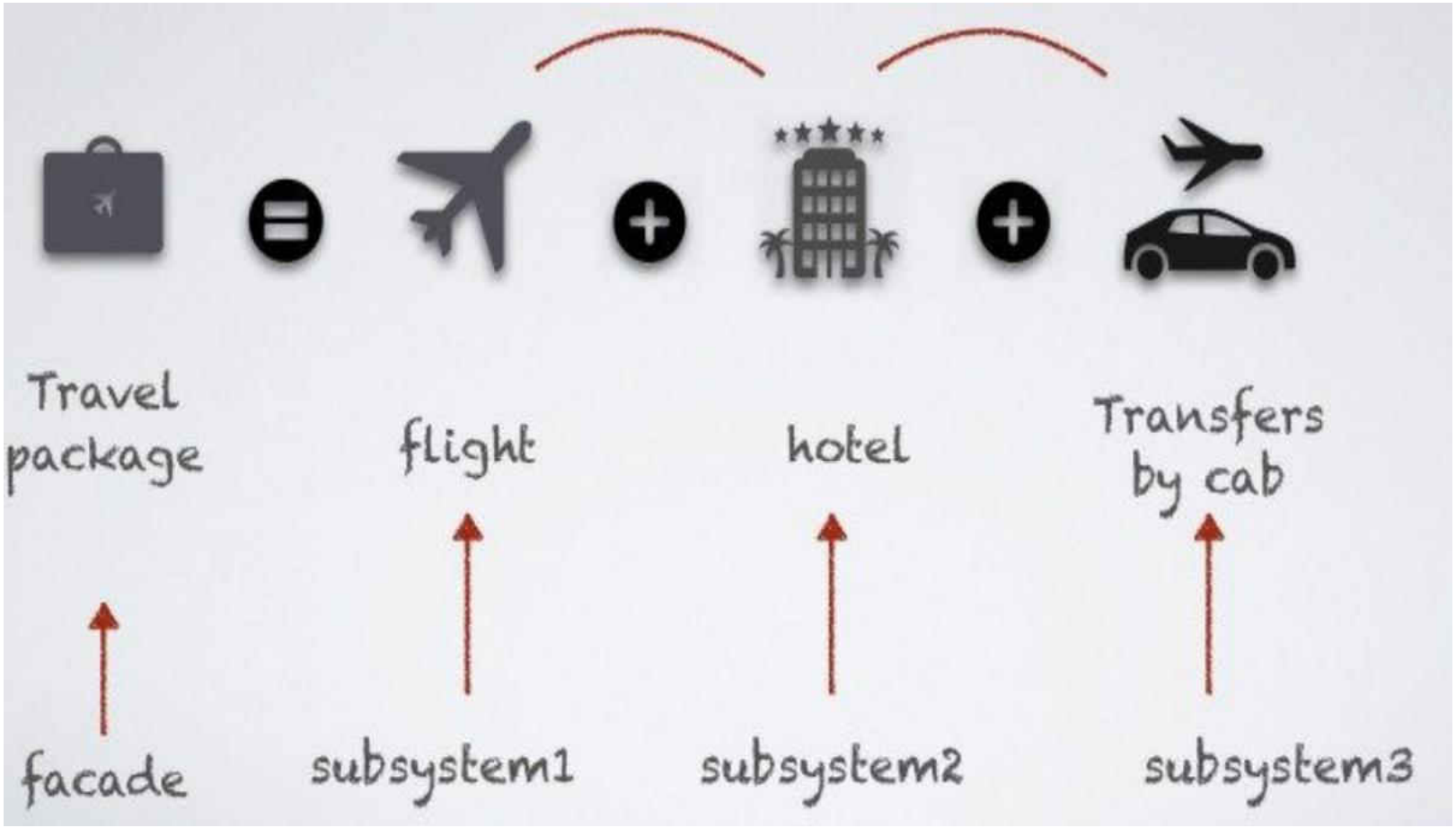
Implementation

• **Reduce client-subsystem coupling**. This can be done by simply **creating a "Façade Abstract Class" and concrete subclasses for the implementation of the subsystem**. Now the client class can communicate with the subsystem through the "**Abstract Façade Class**“.

An alternative approach is to configure a façade object with different subsystem objects

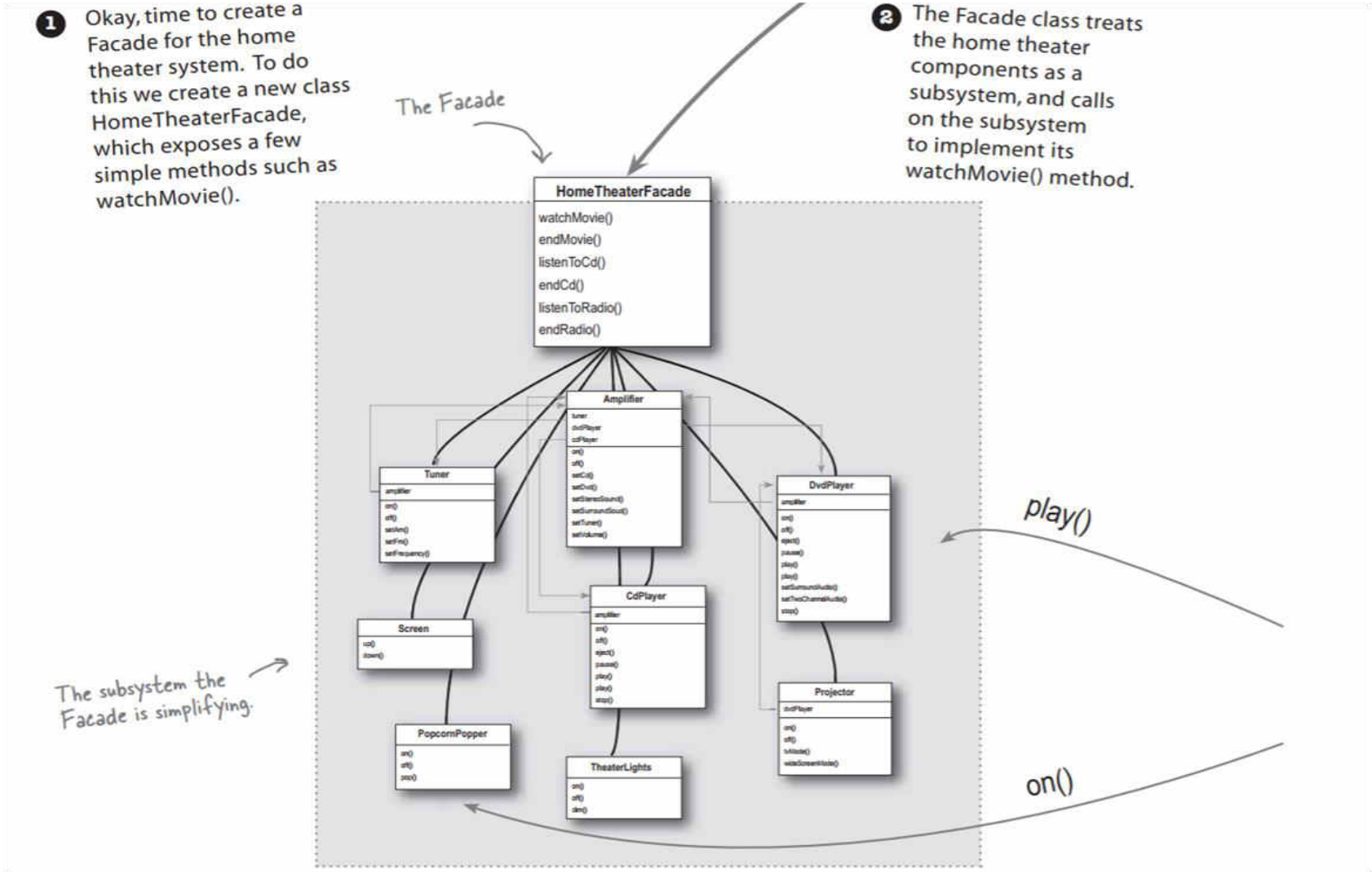
• Public versus Private Interfaces. Classes and Subsystems are similar. Both encapsulate something. Both have private and public interfaces. The **public interface consists of classes accessible by all Clients** whereas the private interface is only for subsystem extenders. **Façade is part of the public interface**

Example -1: Booking a package



Booking a package

Example -2: Designing Hometheater



References

- Facade Pattern from Head First Design Patterns (javaguides.net)
- Facade Design Pattern (sourcemaking.com)
- https://refactoring.guru/design-patterns/java

THANK YOU

Prof. Sindhu R Pai and Saranya devi

Department of Computer Science and Engineering

sindhurpai@pes.edu

Object Oriented Analysis and Design with Java

UE20CS352

Prof. Spurthi N Anjan and Saranya devi B
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

Unit 4
Structural Patterns - Proxy

Department of Computer Science and Engineering

Proxy

- In proxy design pattern, a proxy object provide a surrogate or placeholder for another object to control access to it.
- A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Proxy design pattern - Intent

- Proxy is heavily used to implement lazy loading related usecases where we do not want to create full object until it is actually needed.
- Using the proxy pattern, a class represents the functionality of another class.

Proxy design pattern - Intent

In the Proxy Design Pattern, a client does not directly talk to the original object, it delegates calls to the proxy object which calls the methods of the original object. Moreover, the important point is that the client does not know about the proxy. The proxy acts as an original object for the client.

There are three main variations of the Proxy Pattern:

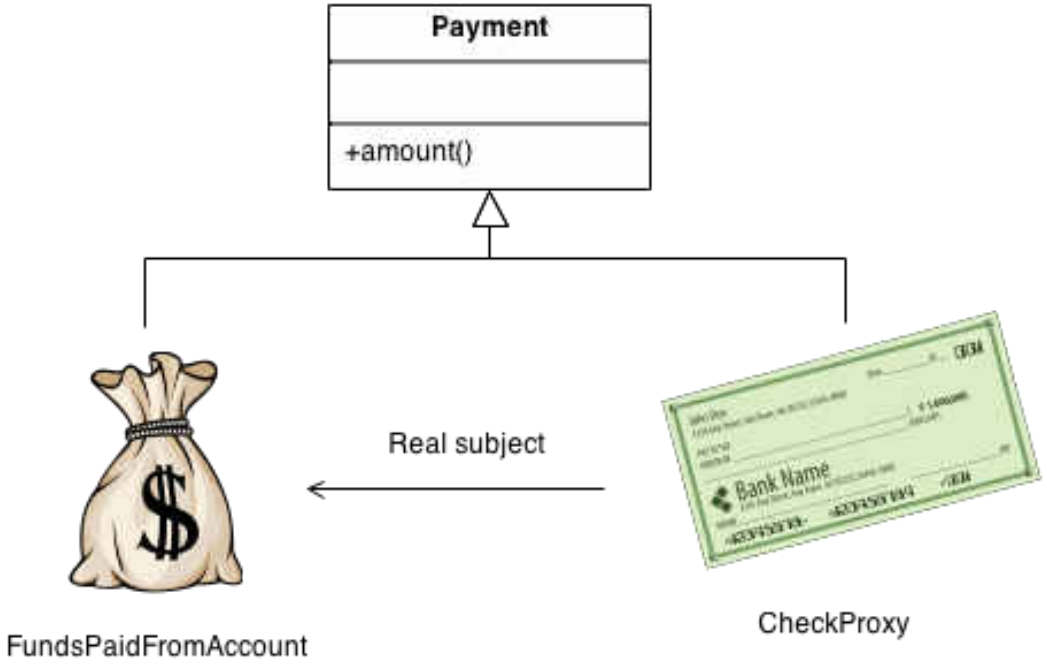
- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand.
- A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

Design participants

- **Subject –** is an interface which expose the functionality available to be used by the clients.
- **Real Subject –** is a class implementing Subject and it is concrete implementation which needs to be hidden behind a proxy.
- **Proxy –** hides the real object by extending it and clients communicate to real object via this proxy object. Usually frameworks create this proxy object when client request for real object.

Example

A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



Real-world example

- In corporate networks, internet access is guarded behind a network proxy. All network requests goes through proxy which first check the requests for allowed websites and posted data to network. If request looks suspicious, proxy block the request – else request pass through.

When to use proxy design pattern- Applicability

The Proxy Pattern can be used in scenarios like, when data can be cached to improve responsiveness, the Real Subject is in a remote location and requires wrapping and unwrapping of requests and making remote calls. The Real Subject should be protected from unauthorized access, or they are expensive to create. A few of the different types of Proxy Pattern implementation are discussed below:

- **Cache Proxy(**Caching request results)

  The Cache Proxy **improves the responsiveness by caching relevant data** (that does not change frequently or is expensive to create) at the Proxy object level, when a client sends a request, the Proxy checks if the requested data is present with it if the data is found, it is returned to the client without sending the request to the Real Object.

- **Remote Proxy(**Local execution of a remote service)

  The Remote Proxy provides a **proxy object at a local level which represents an object present at another location**.

When to use proxy design pattern

- **Protection Proxy(**Access control)

  As its name suggests, the Protection Proxy is used to provide some **security for Real Subject at the proxy level**

- **Virtual Proxy(**Lazy initialization)

  Virtual Proxy is used in a scenario when the **Real Object is a complex object or expensive to create**. This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.

Advantages of Proxy Design Pattern

- A few of the advantages of the Proxy Design Pattern are as follows:

  The proxy works even if the service object isn't ready or is not available.

  Open/Closed Principle - new proxies can be introduced without changing the service or clients.

Disadvantages of Proxy Design Pattern

- A few of the disadvantages of the Proxy Design Pattern are as follows:

  The Proxy Design Pattern introduces an additional layer between the client and the Real Subject, this contributes to the code complexity.

  The response from the service might get delayed.

  Additional request forwarding is introduced between the client and the Real Subject.

Usage examples:

1.Image viewer program that lists and displays high resolution photos. The program has to show a list of all photos however it does not need to display the actual photo until the user selects an image item from a list.

2.The same for document editor that can embed graphical objects in a document. It isn't necessary to load all pictures when the document is opened, because not all of these objects will be visible at the same time.

3.The protective proxy acts as an authorization layer to verify if the actual user has access to appropriate content. An example can be thought about the proxy server which provides restrictive internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.

4.Maybe used also for adding a thread-safe feature to an existing class without changing the existing class's code.

Example

Let's take a scenario where the real image contains a huge size data which clients needs to access. To save our resources and memory the implementation will be as below:

– Create an interface which will be accessed by the client. All its methods will be implemented by the ProxyImage class and RealImage class.

– RealImage runs on the different system and contains the image information is accessed from the database.

– The ProxyImage which is running on a different system can represent the RealImage in the new system. Using the proxy we can avoid multiple loading of the image.

Example

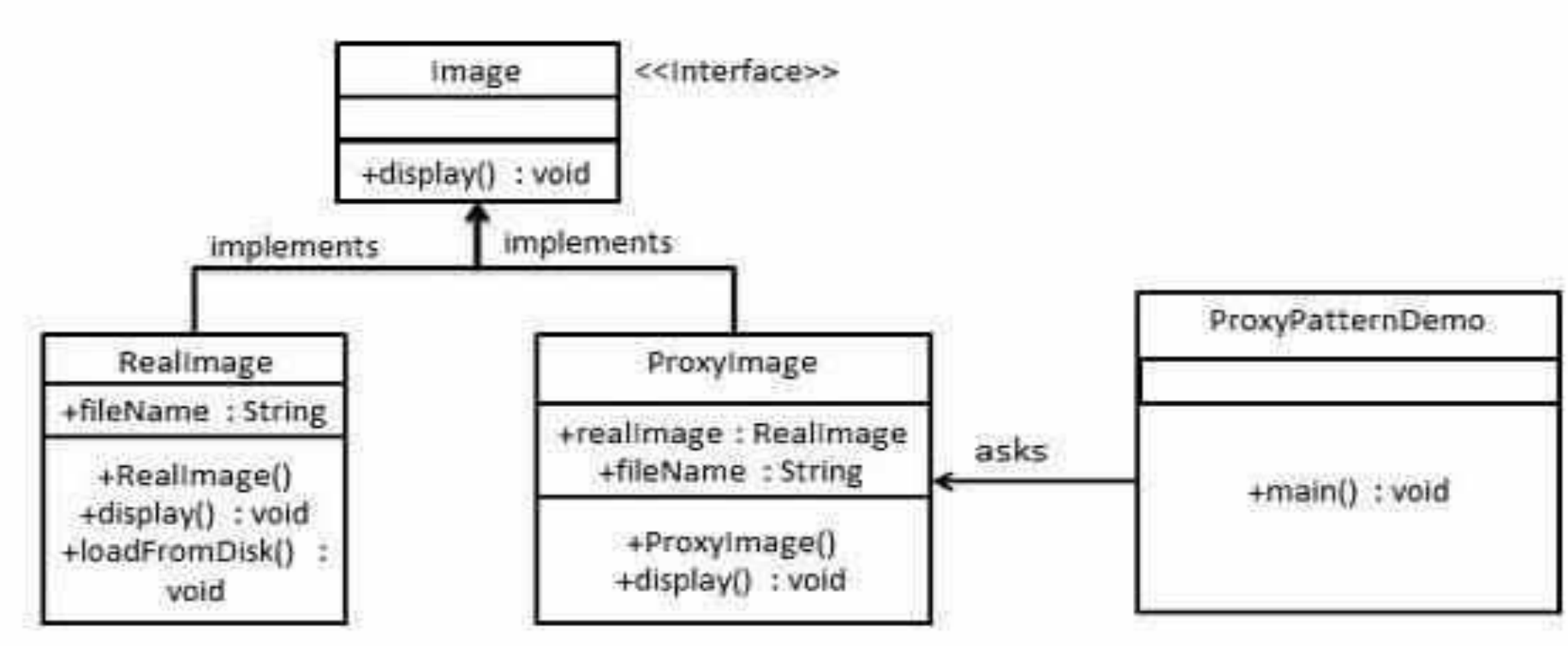
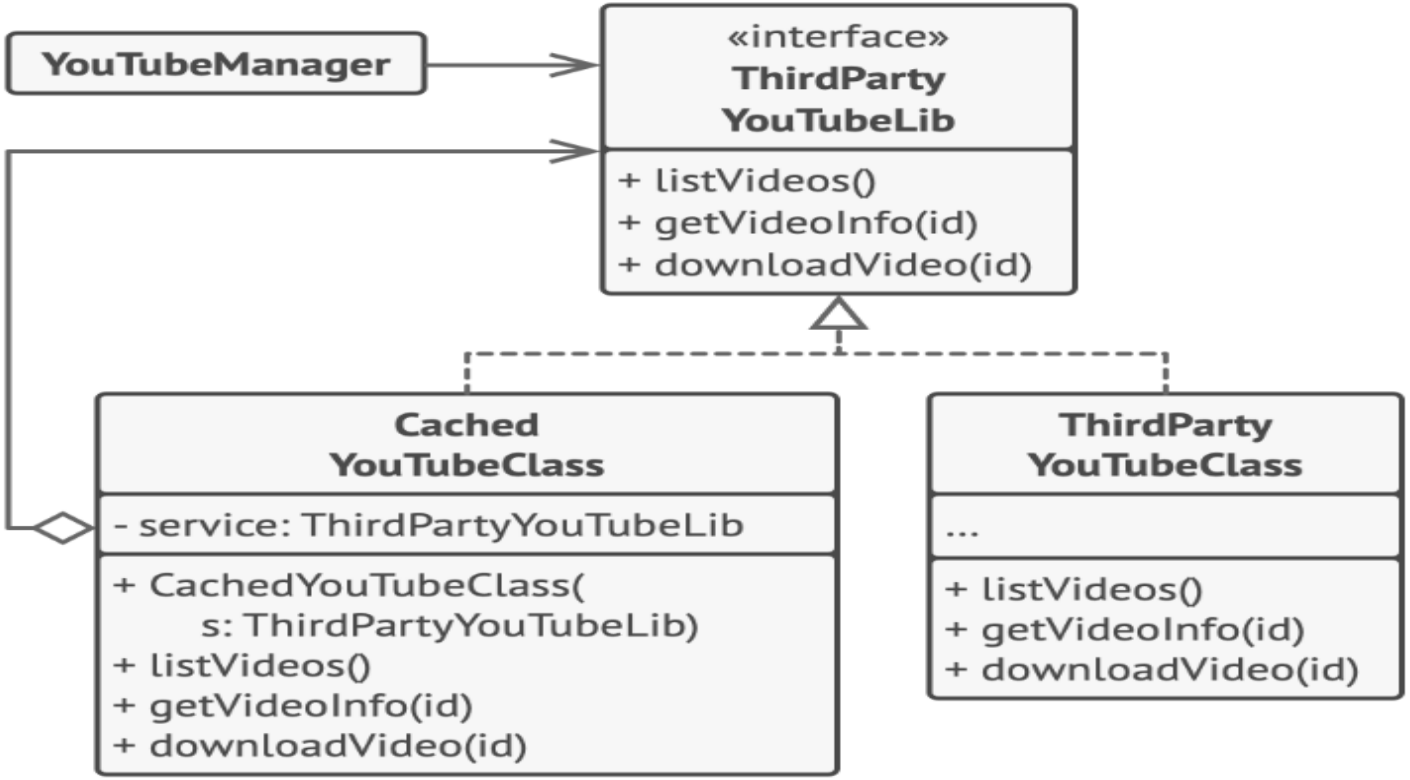


Image
<<Interface>>
+display() : void
implements
implements
RealImage
+fileName : String
+RealImage()
+display() : void
+loadFromDisk() : void
ProxyImage
+realImage : RealImage
+fileName : String
+ProxyImage()
+display() : void
asks
ProxyPatternDemo
+main() : void

Object Oriented Analysis and Design with Java

Example

This example illustrates how the **Proxy** pattern can help to introduce lazy initialization and caching to a 3rd-party YouTube integration library.



Caching results of a service with a proxy.

Example

The library provides us with the video downloading class. However, it's very inefficient. If the client application requests the same video multiple times, the library just downloads it over and over, instead of caching and reusing the first downloaded file.

The proxy class implements the same interface as the original downloader and delegates it all the work. However, it keeps track of the downloaded files and returns the cached result when the app requests the same video multiple times.

THANK YOU

Prof. Spurthi N Anjan and
Saranya devi B

Department of Computer Science and Engineering

Object Oriented Analysis and Design with Java

UE20CS352

Prof. Saranya devi B
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

Unit 4
Structural Patterns - Flyweight

Department of Computer Science and Engineering

Agenda

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

Flyweight design pattern - Introduction

- The flyweight pattern is for sharing objects, where each instance does not contain its own state but stores it externally.
- This allows efficient sharing of objects to save space when there are many instances but a few different types.

Flyweight design pattern - Intent

- Use sharing to support large number of fine-grained objects efficiently.
- Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Flyweight Pattern- Motivation

- An application uses many objects.
- A flyweight is a shared object that can be used in multiple contexts simultaneously.
- It acts as an individual object in each context.
- The key concept is the distinction between intrinsic and extrinsic state.

Problem

Placing the same information in many different place-

1. Mistakes can be made in copying the information.
2. If the data changes, all occurrences of the information must be located and changed. This makes maintenance of the document difficult.
3. Documents can become quite large if the same information is repeated over and over again.

Issues –

1. RAM overhead associated with each Instance.
2. CPU overhead associated with Memory management.

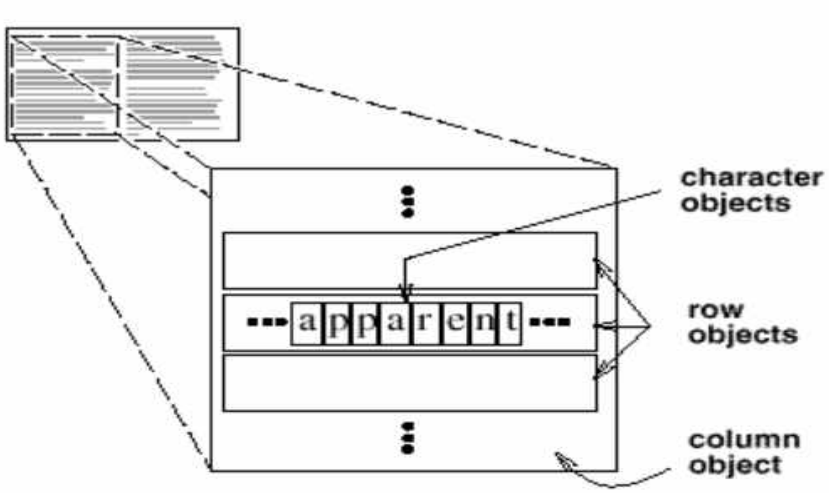
Solution

Create Flyweight Objects for Intrinsic state

- **Intrinsic State** – Information independent of the object's context, sharable(e.g state might include name, postal abbreviation, time zone etc.). This is included in the flyweight and instead of storing the same information n times for n objects, it is stored only once.
- **Extrinsic state** – Information dependent of the object's context, un shareable, stateless, having no stored values, but values that can be calculated on the spot(e.g. state might need access to region). This is excluded from the Flyweight.

Example: OO Document Editor

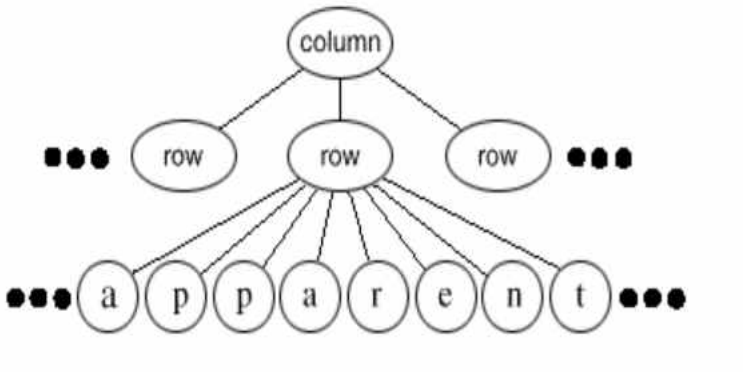
- Using an object for each character in the document:
- Promotes flexibility at the finest levels in the application.
- Character and embedded elements can be treated uniformly with respect to how they are drawn and formatted.
- The application could be extended to support new character sets without disturbing other functionalities.
- The application's object structure could mimic the document's physical structure.



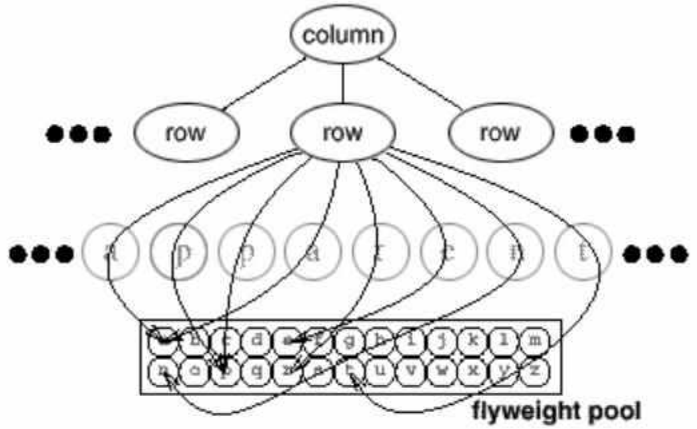
Object Oriented Analysis and Design with Java

Example: OO Document Editor

Logically



physically



Creating a flyweight for each letter of the alphabet:

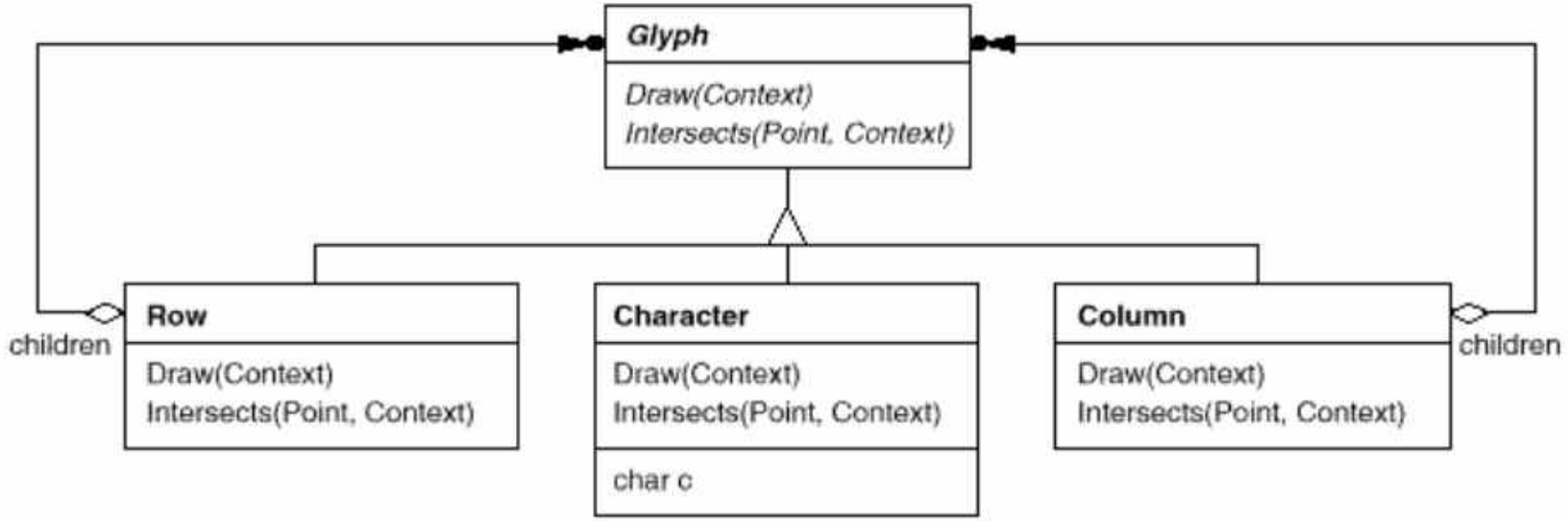
Intrinsic state: A character code

Extrinsic state: Coordinate position in the document

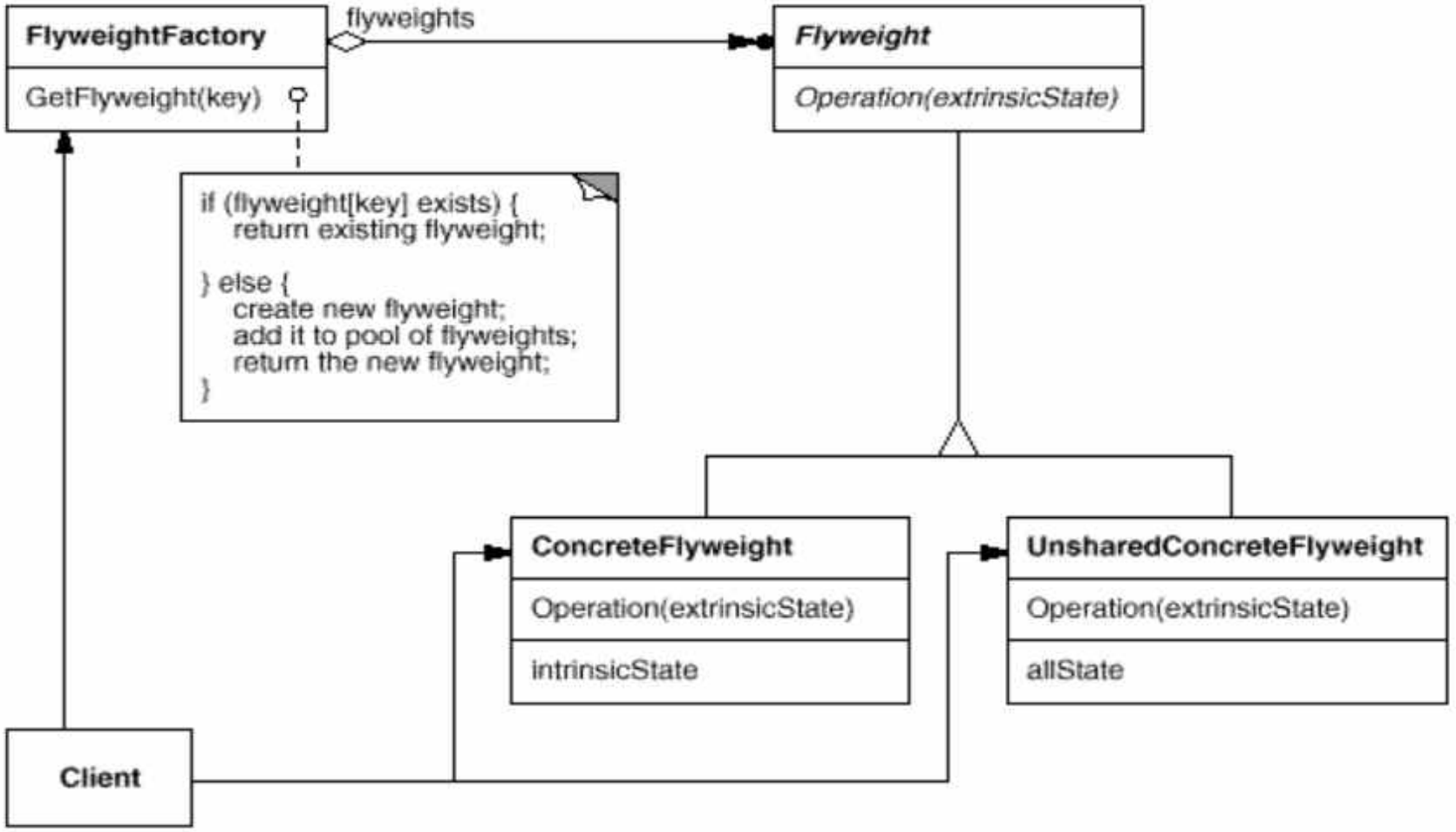
Typographic style(font, color)

Is determined from the text layout algorithms and formatting commands in effect wherever the character appears.

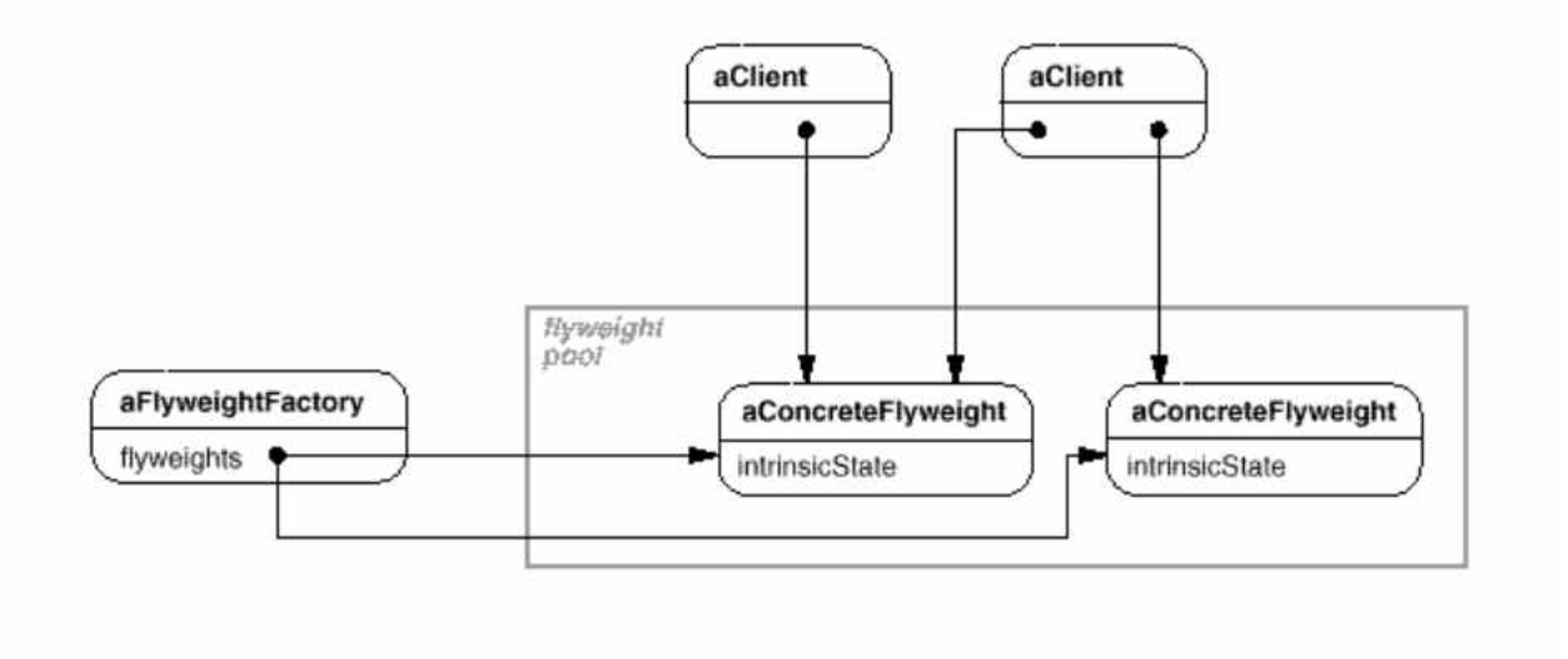
Class Structure



Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Sharing of flyweights



Flyweight Participants

- **Flyweight**
  - declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight** (Character)
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight** (Row, Column)
  - not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
- **Client**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

Applicability

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Consequences

- Disadvantage: Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.
- Advantage: Storage savings are a function of several factors:
- The reduction in the total number of instances that comes from sharing.
- The amount of intrinsic state per object
- Whether extrinsic state is computed or stored.

Implementation

- Removing extrinsic state - The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing.

- Managing shared objects - Because objects are shared, clients shouldn't instantiate them directly. FlyweightFactory lets clients locate a particular flyweight. FlyweightFactory objects often use an associative store to let clients look up flyweights of interest.

Examples:

•Suppose we have a **pen** which can exist with/without **refill**. A refill can be of any color thus a pen can be used to create drawings having N number of colors.
Here Pen can be flyweight object with refill as extrinsic attribute. All other attributes such as pen body, pointer etc. can be intrinsic attributes which will be common to all pens. A pen will be distinguished by its refill color only, nothing else.
All application modules which need to access a red pen – can use the same instance of red pen (shared object). Only when a different color pen is needed, application module will ask for another pen from flyweight factory.
•In programming, we can see **java.lang.String** constants as flyweight objects. All strings are stored in string pool and if we need a string with certain content then runtime return the reference to already existing string constant from the pool – if available.
•In browsers, we

• use an image in multiple places in a webpage. Browsers will load the image only one time, and for other times browsers will reuse the image from cache. Now image is same but used in multiple places. It's URL is intrinsic attribute because it's fixed and shareable. Images position coordinates, height and width are extrinsic attributes which vary according to place (context) where they have to be rendered.

References

- https://www.cs.unc.edu/~stotts/GOF/hires/pat4ffso.htm
- https://refactoring.guru/design-patterns/flyweight
- https://howtodoinjava.com/design-patterns/structural/flyweight-design-pattern/
- Design Patterns: Elements of Reusable Object-Oriented SoftwareErich Gamma, Ralph Johnson, Richard Helm · 1995

THANK YOU

Prof. Saranya devi B

Department of Computer Science and Engineering