# DP

## Content

- Introduction
- O-O aspects
- Selected Patterns

# DP

Introduction

## What is the DP?

Design patterns are recurring solutions to design problems you see over and over.

Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development.

# DP

Introduction

## What is the DP?

DP is coming for :

- How to write a new program feature?
- Have all possible avenues considered?
- A more methodologically way to do the job
- Develop the best solution to the problem.

DP

Introduction

## Patterns Types

- Creational
- Structural
- Behavioral

DP

Introduction

## Creational Types

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

# DP

Introduction

## Structural Types

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
-
- Proxy

DP

Introduction

## Behavioral Types

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

DP

Introduction

## Patterns Classification

1. what a pattern does ?

- Creational patterns - concern the process of object creation
- Structural patterns - deal with the composition of classes or objects
- Behavioral patterns - characterize the ways classes or objects interact

DP

Introduction

## Patterns Classification

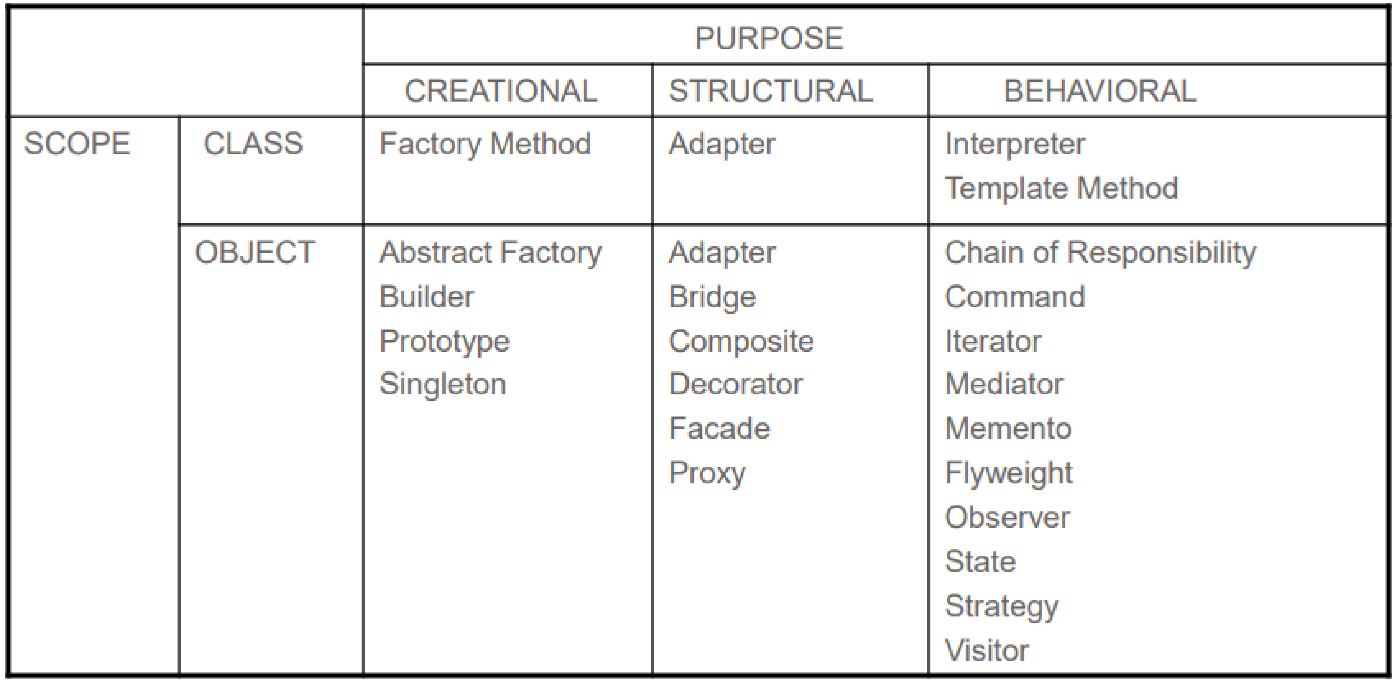
2. how the pattern is applied

- **Class patterns** - ‘is A’ relationships, static
- **Object patterns** - ‘has A’ relationships
- **Compound patterns** - deal with recursive object structures

DP

Introduction

## Patterns Scopes and Purpose

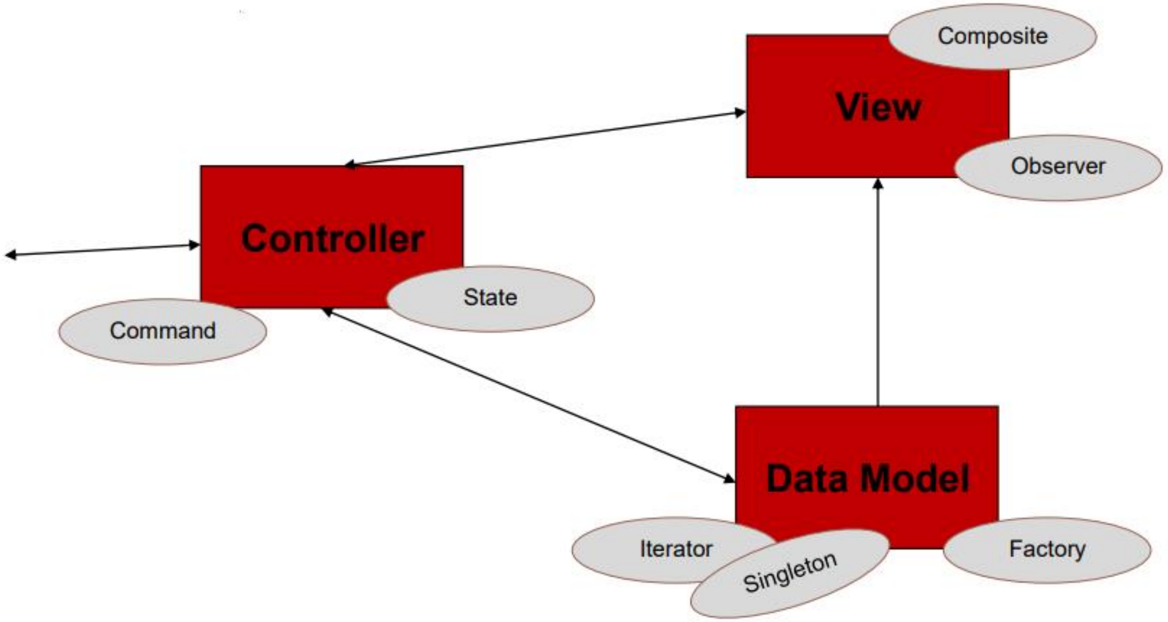


| | | PURPOSE | | |
|---|---|---|---|---|
| | | CREATIONAL | STRUCTURAL | BEHAVIORAL |
| SCOPE | CLASS | Factory Method | Adapter | Interpreter<br>Template Method |
| | OBJECT | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Flyweight<br>Observer<br>State<br>Strategy<br>Visitor |

DP

## Architecture VS. Design Patterns

Introduction



# DP

O-O aspects

## agenda

- Object Oriented approach
- Object composition
- Problems of redundancy
- Defining Object roles
- Inheritance vs. Composition

DP

O-O aspects

## Object Oriented approach

- Keeping classes separated
- Avoid “reinventing the wheel”
- There are several strategies that OO use to achieve separation, among them encapsulation and inheritance.

DP

O-O aspects

## Object Composition

- object composition- “The construction of objects that contain others, encapsulation of several objects inside another one.”
- Inheritance is not the solution of every problem

DP

O-O aspects

## Problems of Redundancy

**Problem: Many elements in a system will share similar structures and/or functionality**

**Common solutions:**

Subroutines and modules

Inheritance

- Define new structure and/or functionality then define a static relationship with “parents” to use theirs
- Implementation (operations and attributes)
- Interface (sub-type)

Composition

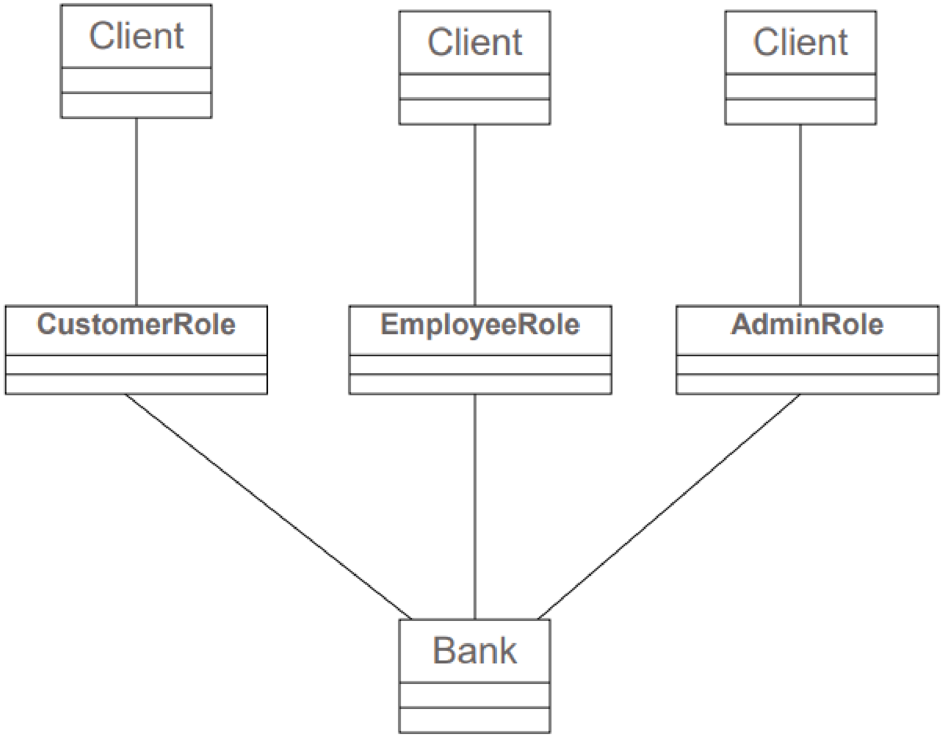
- Combine common “parts” then define new structure and/or functionality as necessary

DP

## Modules

Here, every new client type – forces a whole new module creation

O-O aspects

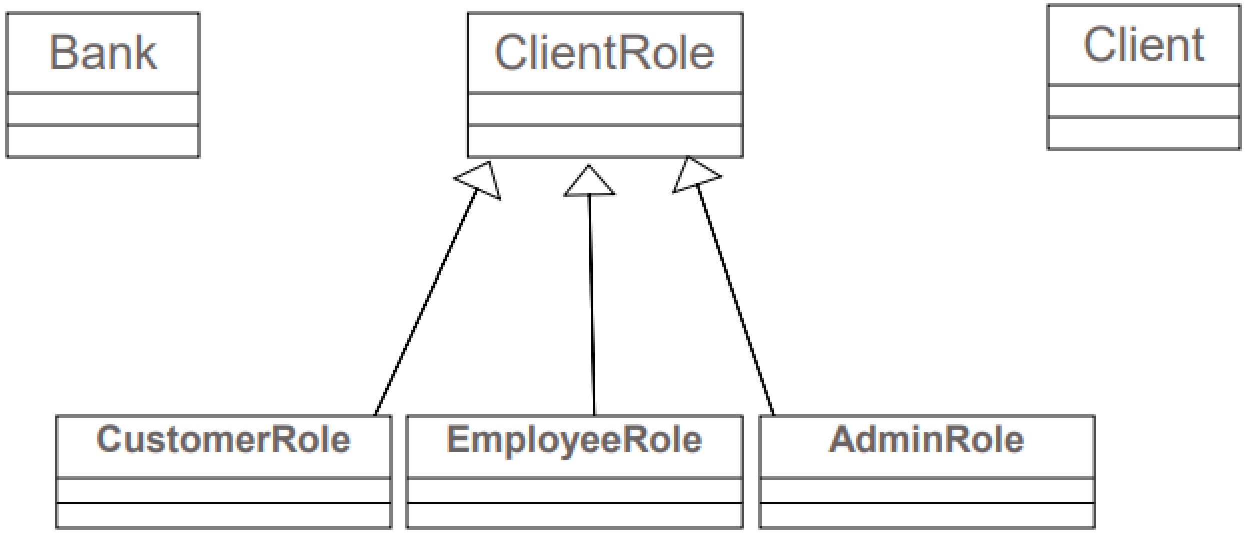


DP

## Inheritance

Here, every new client type is a new subtype of ClientRole

O-O aspects

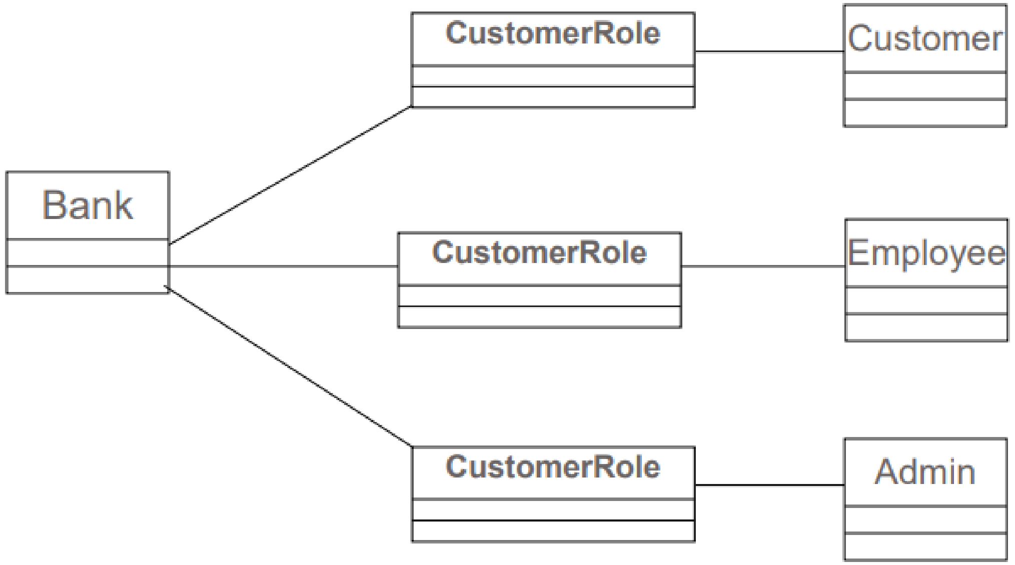


DP

O-O aspects

## Inheritance

Here, client types are visitors of a consistent CustomerRole class used by the Bank



DP

Selected Patterns

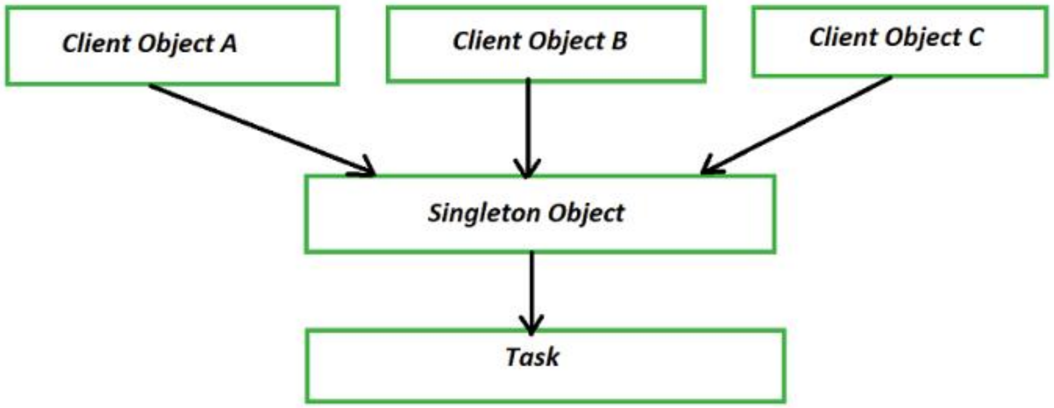
## Creational Patterns

- Singleton Pattern
- Prototype pattern
- The Abstract Factory Pattern
- The Factory Method pattern
- Builder pattern

DP

## The Singleton Pattern

Singleton

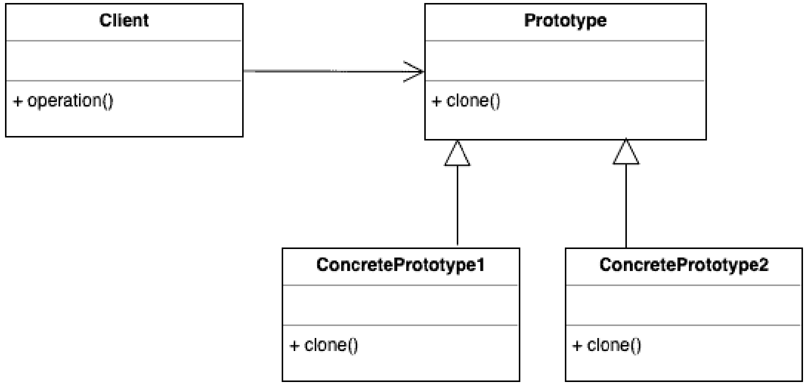


- One instantiation that is globally shared
- The class itself responsible for the creation and lifetime

DP

## The Protype Pattern

Protype

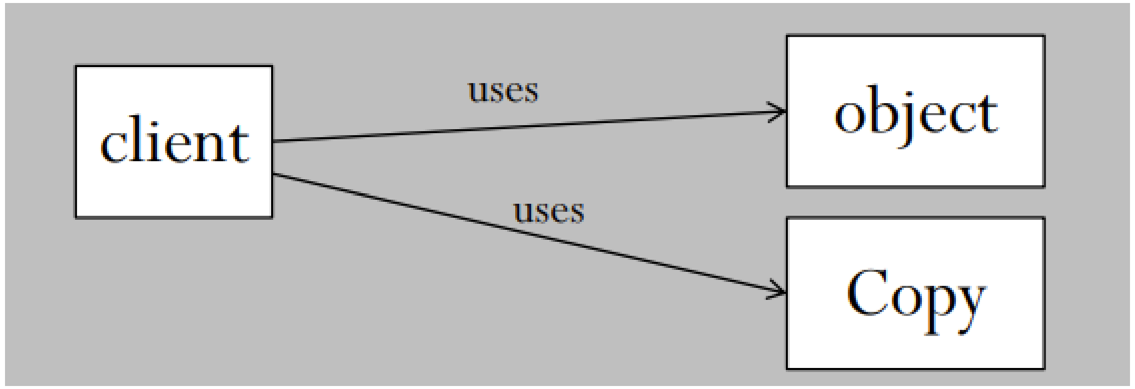


- Create copies of Objects in runtime without knowing their actual nature in advance
- Constructing new Object from a given Object – Clone

DP

Prototype

## The Problem

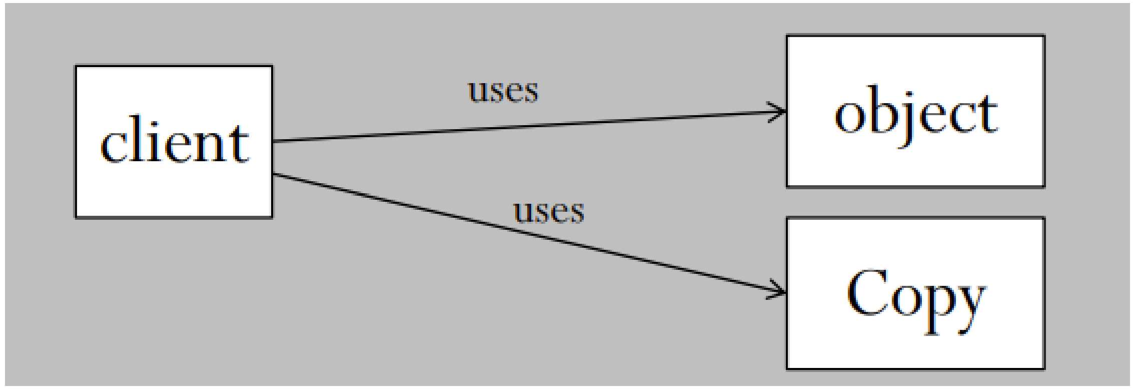


- You don’t know the actual nature / type of Object when you process it for cloning
- You need to create similar objects (or identical, e.g. clone) from a given object

DP

Prototype

## The Problem



Client is forced to :

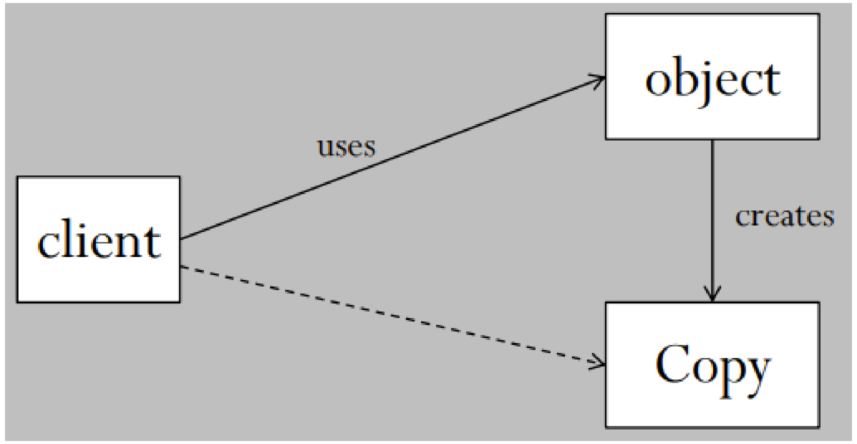
- check the origin object type (class)
- call the origin class constructor to instantiate a copy instance
- verify everything is well copies and populated in that copy

To much time, performance and tightly coupling design…

DP

Prototype

## The Solution



- Client uses the origin object in order to construct a copy
- Object becomes a Factory of its own replications

DP

Abstract Factory

## The Abstract Factory  Patterns

- Abstracts the object creation.
- Let's extensibility.
- Provides an interface for creating families of related or dependent objects.

DP

Abstract Factory

## The Abstract Factory  Patterns

### The Problem

An office needs to process different types of reports before entering them into a database.

Each report has unique fields and requires different processing steps.

Additionally, a third type of report is in planning, and more types may be added in the future.

The goal is to design a **flexible** and **scalable** system that can handle these reports efficiently.

DP

## The Abstract Factory  Patterns

**The Solution:** Abstract Factory Design Pattern

Abstract Factory

- The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- This makes it easier to add new report types and processing steps without altering the existing system.

DP

Abstract Factory

## The Abstract Factory  Patterns

**The Solution:**

1. **Define Abstract Product Interfaces-** Define an interface for each type of report and its processing steps.

```
// Report interface
public interface Report {
            void generateReport();
            void saveToDatabase();
           }
```

DP

## The Abstract Factory  Patterns

Abstract Factory

**The Solution:**

2. **Concrete Product Implementations** – implement the report interface for each specific report type

```
// Report Type A
public class ReportTypeA implements Report {

    @Override
    public void generateReport() {
            System.out.println("Generating Report Type A..."); }

    @Override
    public void saveToDatabase() {
            System.out.println("Saving Report Type A to database..."); }
}
```

DP

## The Abstract Factory  Patterns

Abstract Factory

**The Solution:**

3. **Abstract Factory Interface -** Define an abstract factory interface for creating reports.

```
public interface ReportFactory {
   Report createReport();
}
```

DP

# The Abstract Factory  Patterns

Abstract Factory

## The Solution:

4. **Concrete Factory Implementations-** implement the ReportFactory interface for each specific report type.

```
// Factory for Report Type A
public class ReportTypeAFactory implements ReportFactory {
          @Override public Report createReport() {
                              return new ReportTypeA(); }
          }

// Factory for Report Type B
public class ReportTypeBFactory implements ReportFactory {
          @Override public Report createReport() {
                              return new ReportTypeB(); }
}
```

DP

Abstract Factory

## The Abstract Factory  Patterns

### The Solution:

5. **Client Code -** Use the abstract factory to create and process reports without depending on the specific report types.

```
public class ReportProcessor {
  private ReportFactory reportFactory;

  public ReportProcessor(ReportFactory reportFactory) {
    this.reportFactory = reportFactory;}

  public void processReport() {
    Report report = reportFactory.createReport();
    report.generateReport();
    report.saveToDatabase();}
}
```

DP

Abstract Factory

## The Abstract Factory  Patterns

### The Solution:

5. **Main App-** demonstrate the use of the abstract factory pattern.

```
public class Main {
public static void main(String[] args) {
ReportFactory reportTypeAFactory =
                              new ReportTypeAFactory();
ReportProcessor processorA =
                              new ReportProcessor(reportTypeAFactory);
processorA.processReport();
ReportFactory reportTypeBFactory =
                              new ReportTypeBFactory();
ReportProcessor processorB =
                    new ReportProcessor(reportTypeBFactory);
processorB.processReport(); }}
```

DP

Abstract Factory

## The Abstract Factory  Patterns

**Benefits:**

- **Modularity**: Each report type is encapsulated in its own class, making the codebase more modular and easier to maintain.
- **Scalability**: Adding a new report type involves creating a new concrete product class and a corresponding factory class, with no changes to the existing code.
- **Flexibility**: The client code works with factories and reports through their abstract interfaces, making it easy to extend the system with new report types.

DP

Abstract Factory

## The Abstract Factory  Patterns

**Extending the System:**

To add a new report type (e.g., ReportTypeC):

1. Create a new concrete report class that implements the Report interface.

2. Create a new factory class that implements the ReportFactory interface and returns an instance of the new report class.

3. Update the client code to use the new factory class when needed.

DP

Factory Method

## The Factory Method  Patterns

- The Factory Method pattern is a creational design pattern
- It’s providing an interface for creating objects in a superclass.
- It’s allowing subclasses to alter the type of objects that will be created.
- It’s done by defining a method for creating objects, which subclasses can override to instantiate the appropriate class.

DP

## Structural Patterns

Selected Patterns

- Adapter pattern
- Facade Pattern
- The Bridge Pattern
- Composite Pattern
- Proxy Pattern
- Decorator pattern

DP

Adapter

## The Adapter Patterns

- Wrap up existing classes inside a new target interface.
- Helps in the reuse of existing class with a different interface.
- Achieving it by converting an existing interface to a new interface.

DP

Facade

## The Facade Patterns

- Facade defines a higher-level interface that makes the subsystem easier to use.
- Provides a simplified interface and minimizes the dependency between subsystem.

DP

Composite

## The Composite Patterns

- Compose objects into tree structures
- Let's clients treat individual and compositions of object uniformly
- Complex object out of elementary objects
- Explains the context and forces when a pattern can be applied

DP

Proxy

## The Proxy Patterns

- Proxy Provide a surrogate or placeholder for another object to control access to it.
- A proxy controls access to another object.
- Can also defer the full cost of creation.
- Types:
  - Cache Proxy
  - Count Proxy
  - Protection Proxy
  - Remote Proxy
  - Virtual Proxy

DP

Selected Patterns

## Behavioral Patterns

- Iterator Pattern
- Strategy Pattern
- State Pattern
- Command Pattern
- Mediator Pattern
- Observer Pattern
- Visitor Pattern
- Chain of responsibility

DP

## The Iterator Patterns

Iterator

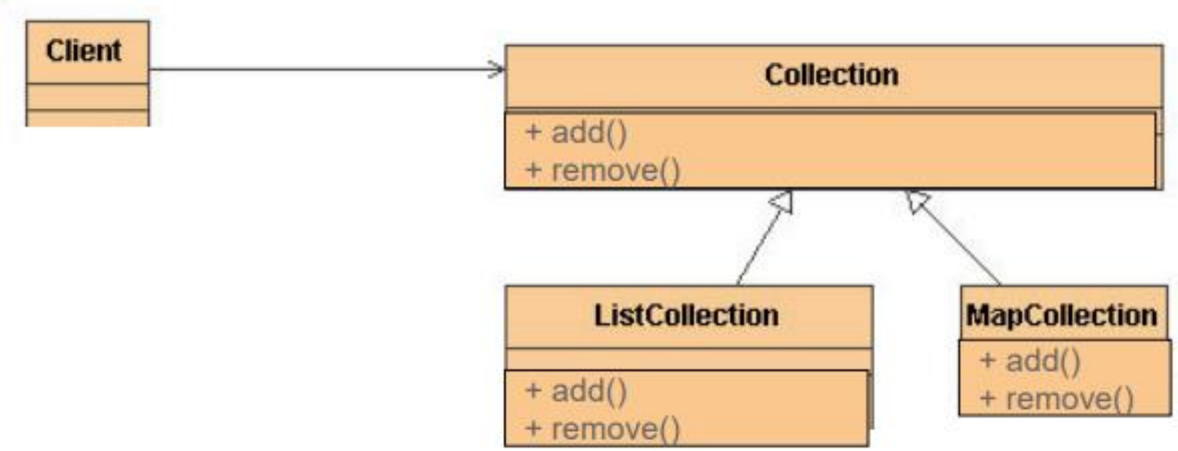
- Iterator is browsing a collection of entities without dealing with its implementation.
- Accesses the elements of an object collection sequentially.
- Done without exposing its underlying representation.

DP

Iterator

## The Problem

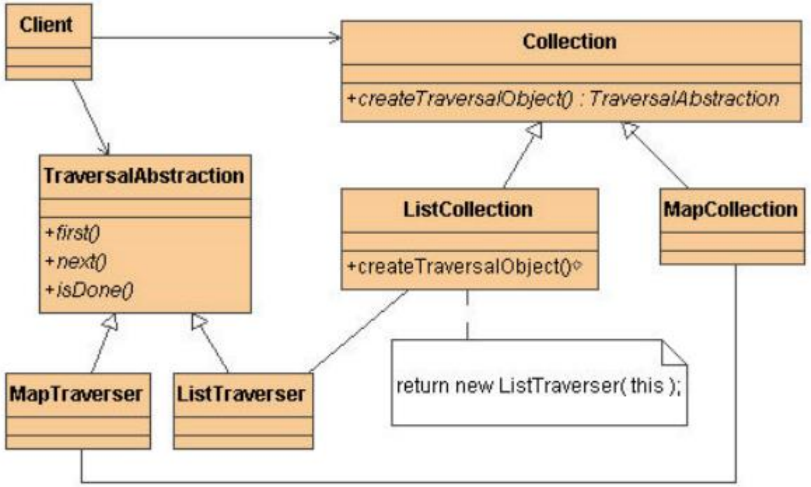
- Clients needs to be familiar with code and logic



DP

## The Solution

Iterator



DP

State

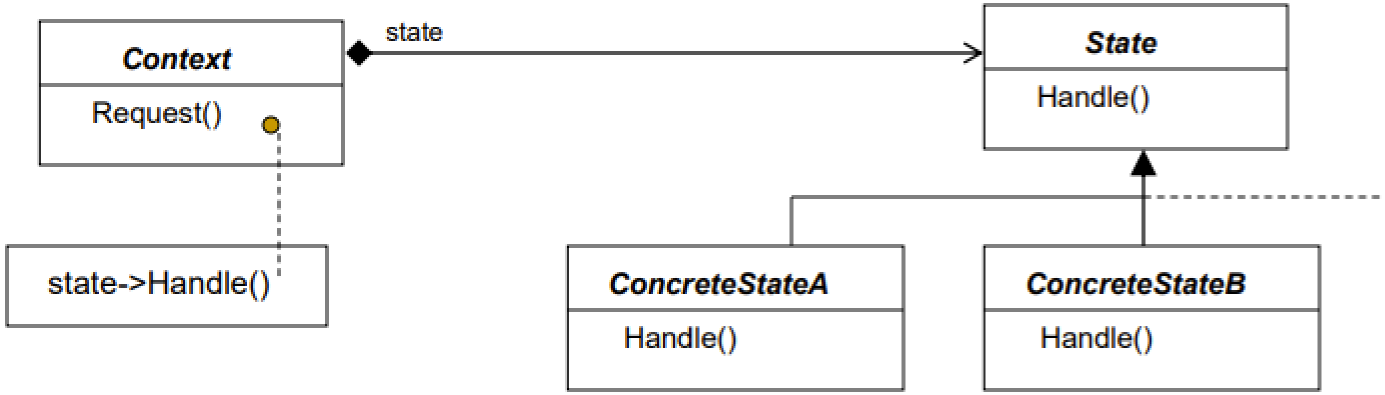
## The State Patterns

- Allow an object to alter its behavior when its internal state changes , The object which appear to change its class.
- “Implementing discrete object states using explicit classes“
- An object’s behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object’s state

DP

State

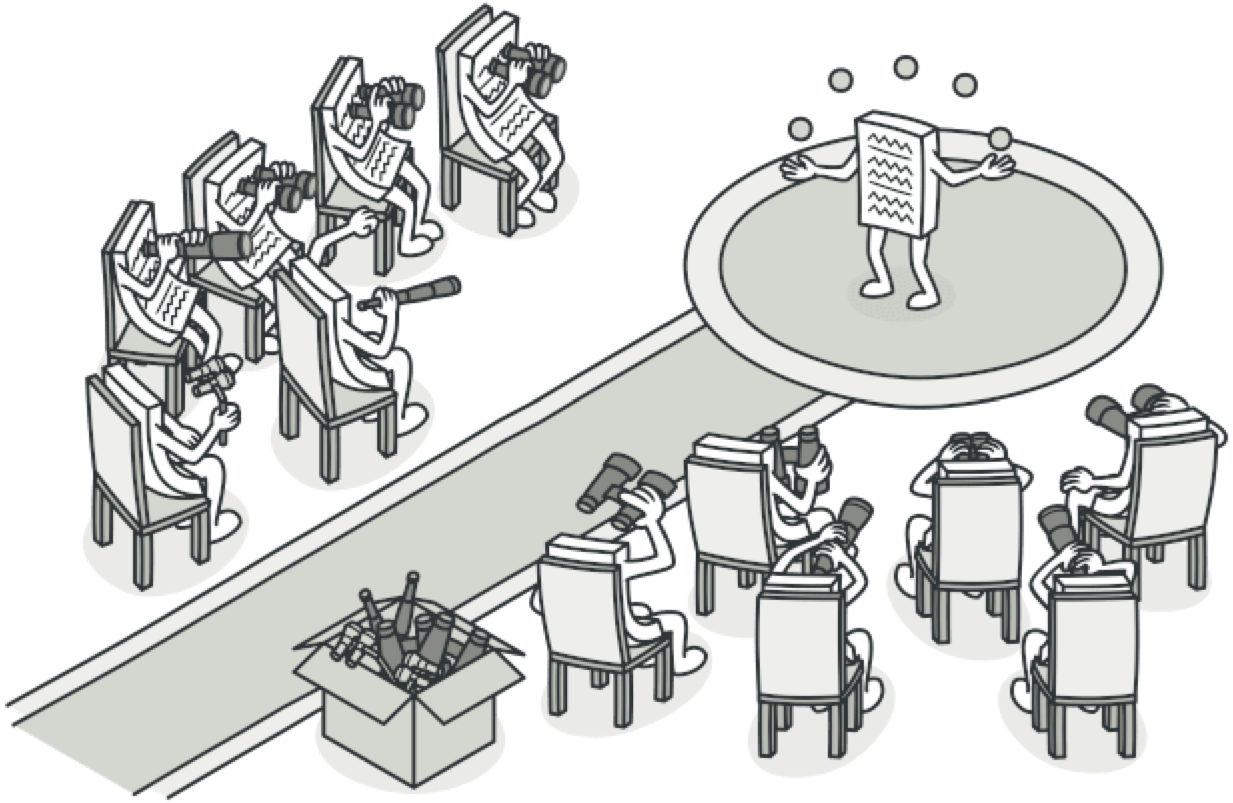
## The State Patterns



DP

## The Observer Patterns

observer



DP

observer

## The Observer Patterns

- Creates a publish-subscribe relationship.
- Observers can register to receive events from the subject.
- Define a one-to-many dependency between objects
- When one object changes state, all its dependents are notified and updated automatically

DP

Thank You !!