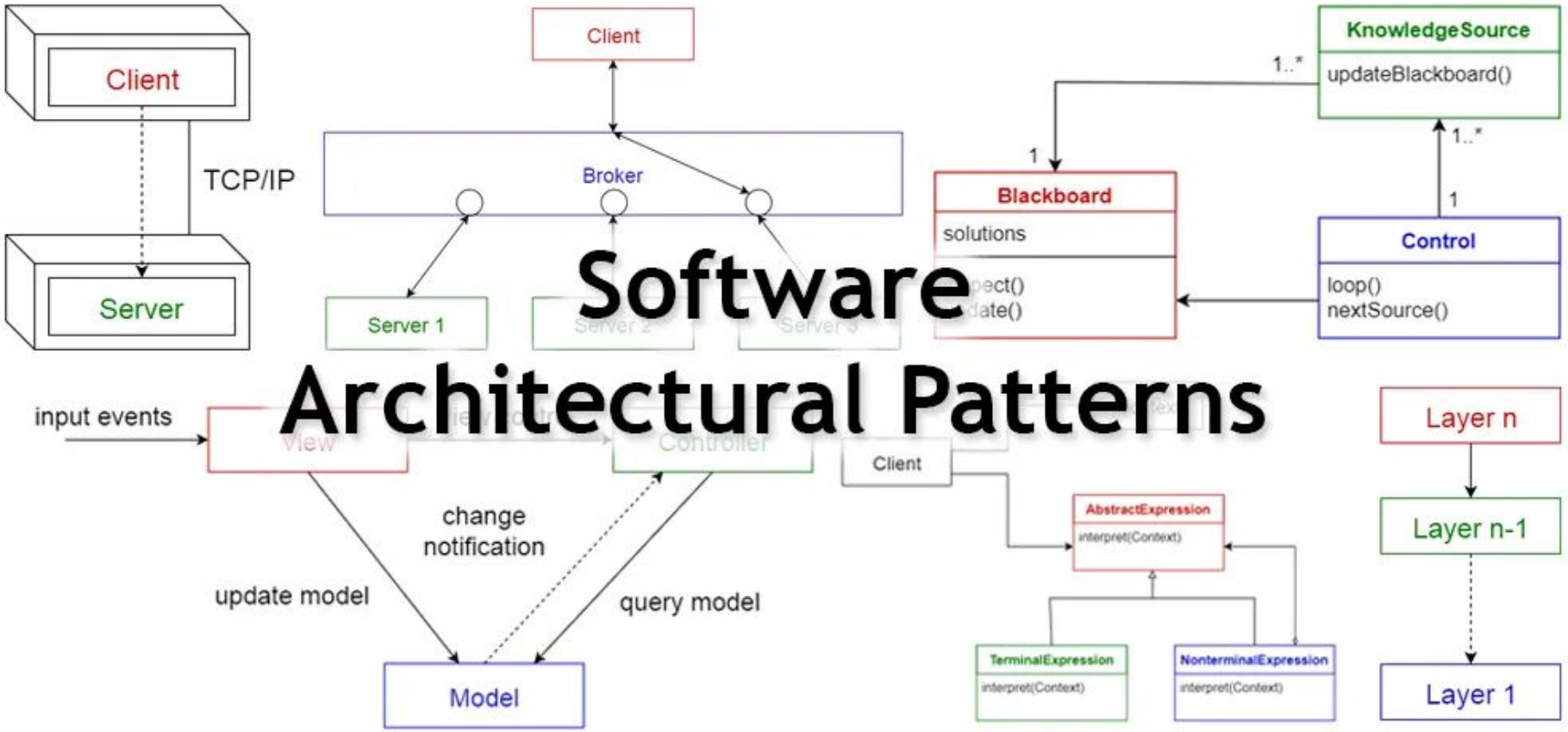


Software Architectural Patterns

Architectural Pattern

Architectural patterns describe the fundamental structure of a software system and are often based on design patterns.

An idiom is an implementation of an architecture or design pattern in a concrete programming language.

Software Architectural Pattern

1. **Layered pattern**
2. **Client-server pattern**
3. **Master-slave pattern**
4. **Pipe-filter pattern**
5. **Broker pattern**
6. **Peer-to-peer pattern**
7. **Event-bus pattern**
8. **Model-view-controller pattern**
9. **Blackboard pattern**
10. **Interpreter pattern**

Scope	Creational Patterns	Structural Patterns	Behavioral Patterns
Class Patterns	▪ **Factory method**	▪ **Adapter**	▪ Interpreter ▪ **Template method**
Object Patterns	▪ Abstract factory ▪ Builders ▪ Prototype ▪ **Singleton**	▪ **Adapter** ▪ **Bridge** ▪ **Composite** ▪ **Decorator** ▪ **Facade** ▪ **Proxy** ▪ Flyweight	▪ Chain of responsibility ▪ Command ▪ **Iterator** ▪ Mediator ▪ Memento ▪ **Observer** ▪ State

1. Creational Pattern

a. Creational Patterns deal with the creation of objects.
b. With Creational Patterns we can abstract away the process of object instantiation.
c. We can use composition rather than inheritance.

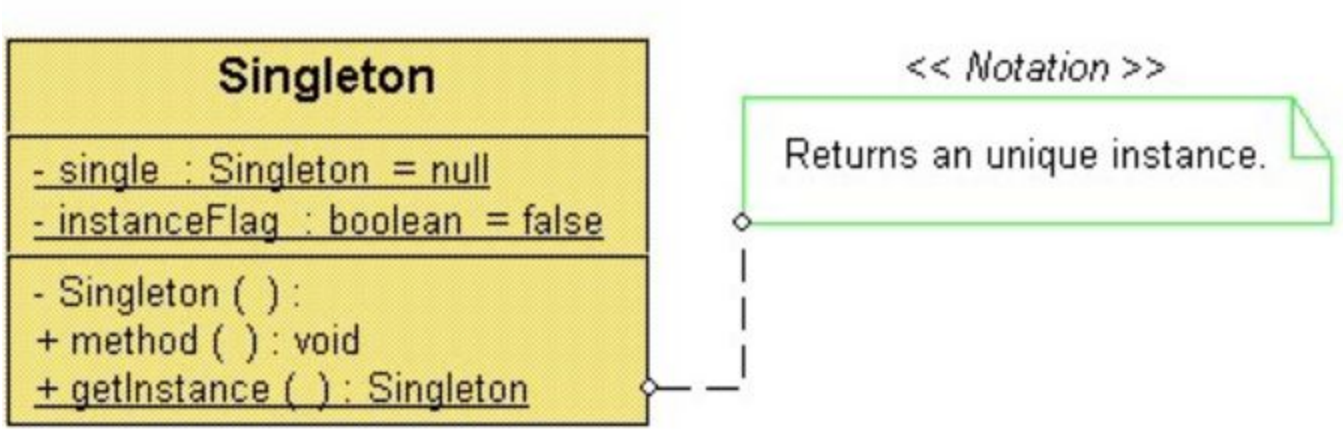
Singleton Pattern

1. You have a class that must only ever be instantiated once.
2. Design the class in a way that only single object is returned each time.
3. A singleton class is a special type of class that have only one object or instance at a time.
4. The new variable also points to the initial instance created if we attempt to instantiate the Singleton class after the first time.
5. This is implemented by using the core concepts of object-oriented programming namely access modifiers, constructors & static methods.

Steps to implement Singleton

1. Make all the constructors of the class private.
2. Delete the copy constructor of the class.
3. Make a private static pointer that can point to the same class object (singleton class).
4. Make a public static method that returns the pointer to the same class object (singleton class)

Class Diagram



Step 1: Class with single instance

```
class Singleton {
public:
   // This is how clients can access the single instance
   static Singleton* getInstance();

   void setValue(int val) {value_ = val;}
   int  getValue()           {return(value_);}

protected:
   int value_;

private:
   static Singleton* inst_;   // The one, single instance
   Singleton() : value_(0) {} // private constructor
   Singleton(const Singleton&);
   Singleton& operator=(const Singleton&);
};
```

Private Static Pointer

Private Constructor

Step2:Define Static Singleton Pointer

```
// Define the static Singleton pointer
Singleton* Singleton::inst_ = NULL;

Singleton* Singleton::getInstance() {
   if (inst_ == NULL) {
      inst_ = new Singleton();
   }
   return(inst_);
}

int main() {

   Singleton* p1 = Singleton::getInstance();

   p1->setValue(10);

   Singleton* p2 = Singleton::getInstance();

   cout << "Value = " << p2->getValue() << '\n';
}
```

Abstract Factory

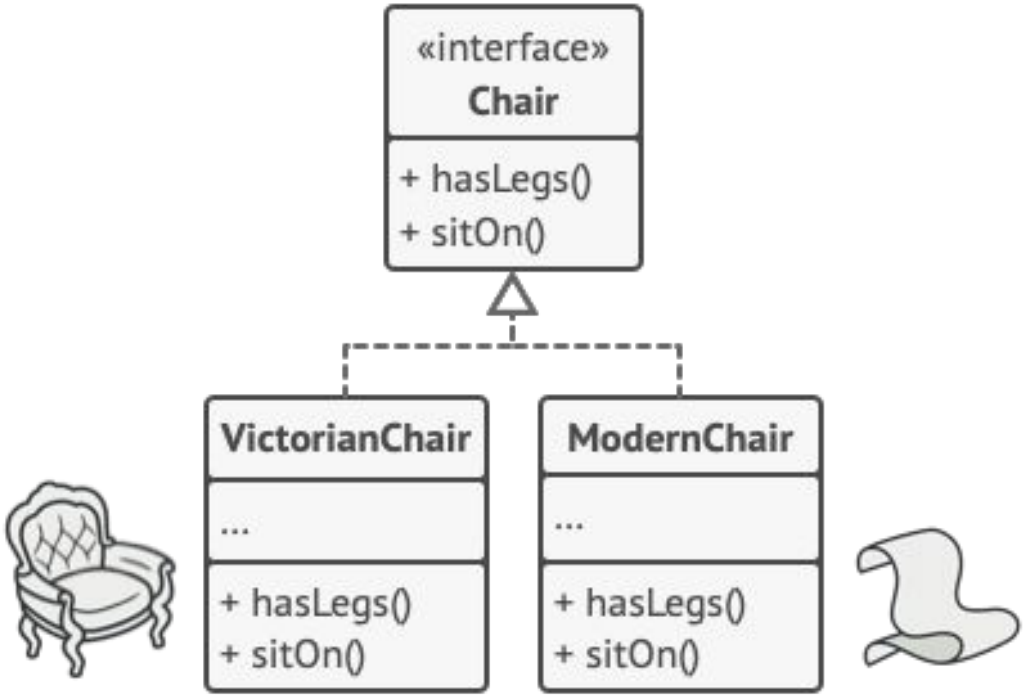
Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

1. A family of related products, say:

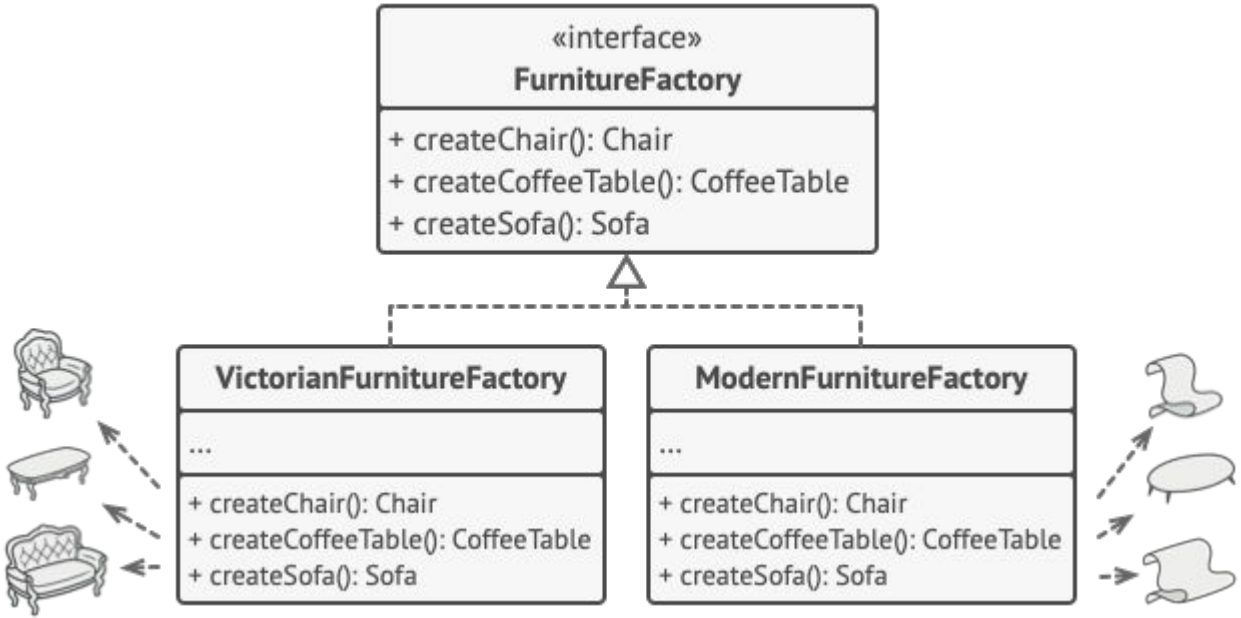
   `Chair` + `Sofa` + `CoffeeTable`.

2. Several variants of this family. For example, products
3. `Chair` + `Sofa` + `CoffeeTable` are available in these variants:
4. `Modern`, `Victorian`, `ArtDeco`.

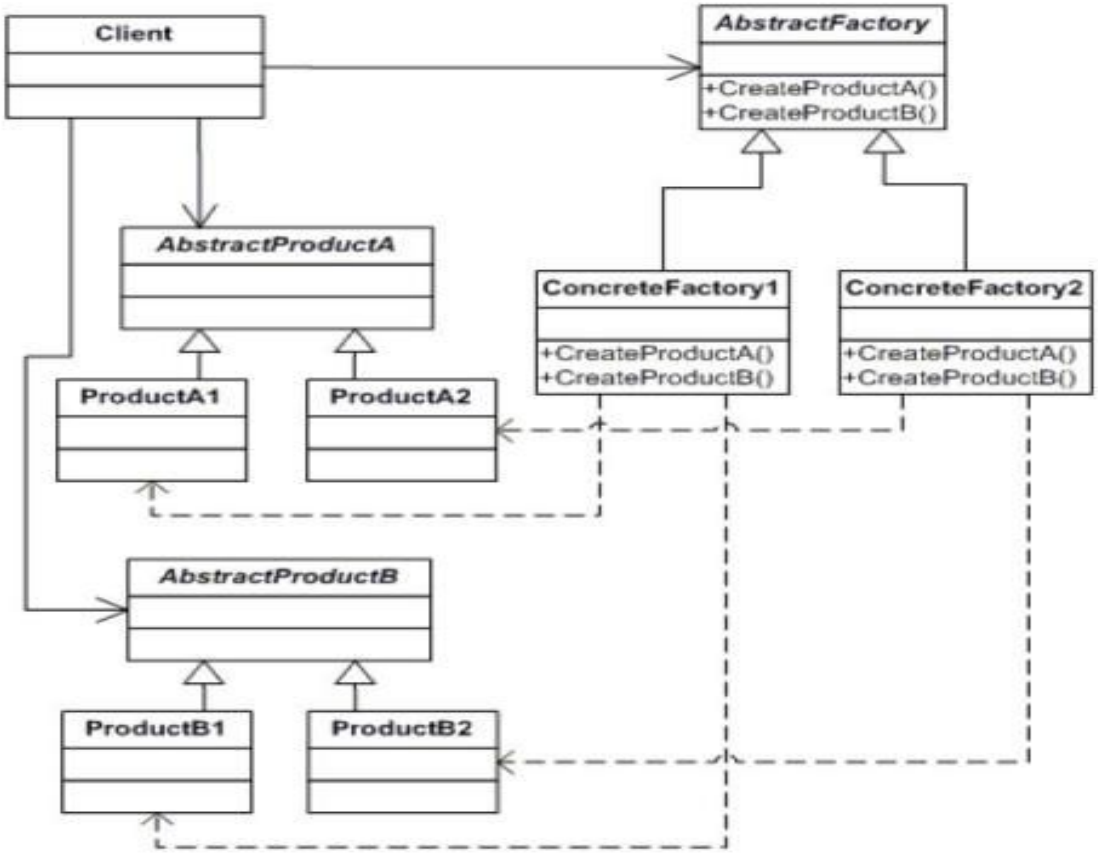
Abstract Factory



Class Diagram of Abstract Factory



Example 2


Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

Steps

- `AbstractFactory`: declares an interface for operations that create abstract products
- `ConcreteFactory`: implements the operations to create concrete product objects
- `AbstractProduct`: declares an interface for a type of product object
- `Product`: defines a product object to be created by the corresponding concrete factory implements the `AbstractProduct` interface
- `Client`: uses interfaces declared by `AbstractFactory` and `AbstractProduct` classes

Implementation

Step1: We need to create the appropriate object containing the information about cell phone based on the user request of:

1. Type of phone
2. Phone manufacturer

Step 2: To make a simple scenario, let's assume we have 3 manufacturers:

1. Nokia
2. Samsung
3. HTC

Steps:Creating Abstract Factory

```
//ISmart interface
class ISmart
{
public:
    virtual std::string Name() = 0;
};

//IDumb interface
class IDumb
{
public:
    virtual std::string Name() = 0;
};
```

Step 2: Defining Concrete Class1

C++

```
class Asha : public IDumb
{
public:
     std::string Name()
     {
          return "Asha";
     }
};

class Primo : public IDumb
{
public:
     std::string Name()
     {
          return "Guru";
     }
};

class Genie : public IDumb
{
public:
     std::string Name()
     {
          return "Genie";
     }
};
```

Step 2: Defining Concrete CLass 2

C++

```
class Asha : public IDumb
{
public:
    std::string Name()
    {
        return "Asha";
    }
};

class Primo : public IDumb
{
public:
    std::string Name()
    {
        return "Guru";
    }
};

class Genie : public IDumb
{
public:
    std::string Name()
    {
        return "Genie";
    }
};
```

Step 3: Creating an Abstract Factory

```
class APhoneFactory
{
public:
    enum PHONE_FACTORIES
    {
        SAMSUNG,
        HTC,
        NOKIA
    };

    virtual ISmart* GetSmart() = 0;
    virtual IDumb* GetDumb() = 0;

    static APhoneFactory* CreateFactory(PHONE_FACTORIES factory);
};

//CPP File
APhoneFactory* APhoneFactory::CreateFactory(PHONE_FACTORIES factory)
{
    if(factory == PHONE_FACTORIES::SAMSUNG)
    {
        return new SamsungFactory();
    }
    else if(factory == PHONE_FACTORIES::HTC)
    {
        return new HTCFactory();
    }
    else if(factory == PHONE_FACTORIES::NOKIA)
    {
        return new NokiaFactory();
    }
}
```

Step4: Creating a Concrete Factory

```
class SamsungFactory : public APhoneFactory
{
public:
    ISmart* GetSmart()
    {
        return new GalaxyS2();
    }

    IDumb* GetDumb()
    {
        return new Primo();
    }
};

class HTCFactory : public APhoneFactory
{
public:
    ISmart* GetSmart()
    {
        return new Titan();
    }

    IDumb* GetDumb()
    {
        return new Genie();
    }
};

class NokiaFactory : public APhoneFactory
{
public:
    ISmart* GetSmart()
    {
        return new Lumia();
    }

    IDumb* GetDumb()
    {
        return new Asha();
    }
};
```

Step5: Preparing a client

C++

```
int main(int argc, char* argv[])
{
    APhoneFactory *factory = APhoneFactory::CreateFactory
                (APhoneFactory::PHONE_FACTORIES::SAMSUNG);

    cout << "Dumb phone from Samsung: " << factory->GetDumb()->Name() << "\n";
    delete factory->GetDumb(); //USe of smart pointer will get rid of these delete
    cout << "Smart phone from Samsung: " << factory->GetSmart()->Name() << "\n";
    delete factory->GetSmart(); //USe of smart pointer will get rid of these delete
    delete factory;
    getchar();

    factory = APhoneFactory::CreateFactory(APhoneFactory::PHONE_FACTORIES::HTC);
    cout << "Dumb phone from HTC: " << factory->GetDumb()->Name() << "\n";
    delete factory->GetDumb(); //USe of smart pointer will get rid of these delete
    cout << "Smart phone from HTC: " << factory->GetSmart()->Name() << "\n";
    delete factory->GetSmart(); //USe of smart pointer will get rid of these delete
    delete factory;
    getchar();

    factory = APhoneFactory::CreateFactory(APhoneFactory::PHONE_FACTORIES::NOKIA);
    cout << "Dumb phone from Nokia: " << factory->GetDumb()->Name() << "\n";
    delete factory->GetDumb(); //USe of smart pointer will get rid of these delete
    cout << "Smart phone from Nokia: " << factory->GetSmart()->Name() << "\n";
    delete factory->GetSmart(); //USe of smart pointer will get rid of these delete
    getchar();

    return 0;
}
```

Final Output

The concrete products here are not telling anything but names of products but they can contain more information too.

Advantage of Factory Pattern

- **Consistency:** It ensures that objects created by a factory are compatible and consistent within a family, improving the overall system's integrity.
- **Flexibility:** The pattern allows for the easy addition of new product families or variations without modifying existing client code.
- **Encapsulation:** Concrete classes are encapsulated within their respective factories, reducing dependencies and making it easier to manage changes.

Disadvantages

- **Complexity:** Implementing the pattern can lead to a large number of classes and interfaces, potentially increasing code complexity.
- **Rigidity:** Modifying or extending a product family may require changes in multiple places, making the system less flexible.
- **Runtime Costs:** Creating objects through factories can introduce some runtime overhead, although it’s usually negligible.

Apply the Changes but Follow the Pattern

Singleton-Case 2

1. We do not want the singleton by copied so that there is only one instance. This can be achieved by declaring a private copy constructor and a private assignment operator.
2. The **getInstance()** method should return a reference rather than a pointer.
3. This blocks a client from deleting the object. Also, by making destructor private, we can achieve the same effect.

Factory Pattern-Case 2

Apply the Factory Architecture for your selected Projects

Prototype Design Pattern

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

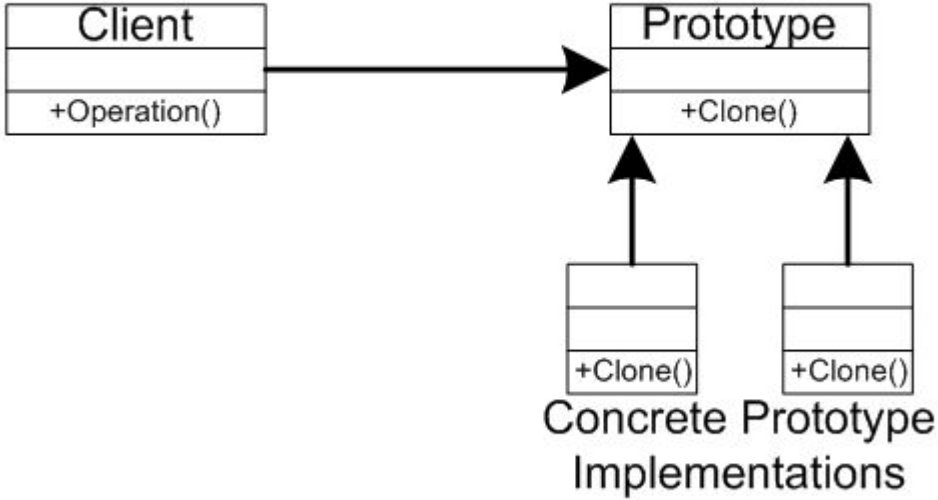
Why to use Prototype?

1. How you can create an exact copy of object?
2. First step is, you have to create a new object of the same class, go through all the fields of the original object and copy their values to the new object.
3. Sure? ,not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.
4. Using duplicate of the code, new code become dependent.

Why to use Prototype?

1. The pattern declares a common interface for all objects that support cloning.
2. This interface allow you to clone an object without coupling your code to the class of that object.
3. This is possible if an interface just have a single `clone` method.
4. The implementation of the `clone` method is very similar in all classes.
5. The method creates an object of the current class and carries over all of the field values of the old object into the new one.
6. Cloning allow you to copy private fields as allowed in most programming languages to access private fields of other objects that belong to the same class.
7. This will help to create Prototype architectural pattern.

Diagram


Client
+Operation()
Prototype
+Clone()
+Clone()
+Clone()
Concrete Prototype
Implementations

Method 1 for prototype

Use Copy Constructor techniques

Step 1:Defining an abstract Class

```
class Shape {
public:
    virtual Shape* clone() const = 0; // Clone method for creating copies.
    virtual void draw() const = 0; // Draw method for rendering the shape.
    virtual ~Shape() {}                // Virtual destructor for proper cleanup.
};
```

Step2

Step 2: Create Concrete Prototype Classes

1. Now, we define concrete classes i.e., classes that can be instantiated, meaning you can create objects (instances) of those classes) that inherit from our abstract base class ***Circle*** and ***Rectangle*.**
2. These classes implement the **clone()** and **draw()** methods for their respective shapes.

Step2

```
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    Shape* clone() const override {
        return new Circle(*this);
    }

    void draw() const override {
        std::cout << "Drawing a circle with radius " << radius << std::endl;
    }
};
```

Define Rectangle Class

```
class Rectangle : public Shape {
private:
    double width;
    double height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    Shape* clone() const override {
        return new Rectangle(*this);
    }

    void draw() const override {
        std::cout << "Drawing a rectangle with width " << width << "
and height " << height << std::endl;
    }
};
```

Step3

1. Creating Prototype instance of rectangle

```
Rectangle rectanglePrototype(4.0, 6.0);
```

2. Clone the prototype to define new shapes

```
Shape* shape2 = rectanglePrototype.clone();
```

3. Draw

```
shape2->draw(); // Output: Drawing a rectangle with width 4 and height 6
```

Home Assignment

Use Clone method to apply prototype Design pattern

Comparison of Creational Design patterns

Rational Design Patterns	Summary	Major Points
Abstract Factory	Create an instance of several families of classes.	Declare interface, Concrete and abstract class
Singleton	A class of which only a single instance to be exist	Abstract Modifier, Constructor, single copy of object
Prototype	A fully initialized instance to be copied and cloned	Use Copy Constructor or Conining

Structural Design Patterns

- Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.
- **SSDP** are concerned about the communication among classes and objects that may form larger structures.
- The structural design patterns **simplifies the structure by identifying the relationships i.e.,**
  - Cohesion
  - Coupling
  - Inheritance
  - Polymorphism
-

Types of Structural Design Pattern

- **Adapter**
  Match interfaces of different classes
- **Bridge**
  Separates an object's interface from its implementation
- **Composite**
  A tree structure of simple and composite objects
- **Decorator**
  Add responsibilities to objects dynamically
- **Facade**
  A single class that represents an entire subsystem
- **Flyweight**
  A fine-grained instance used for efficient sharing
- **Private Class Data**
  Restricts accessor/mutator access
- **Proxy**
  An object representing another object

Adapter Design Pattern

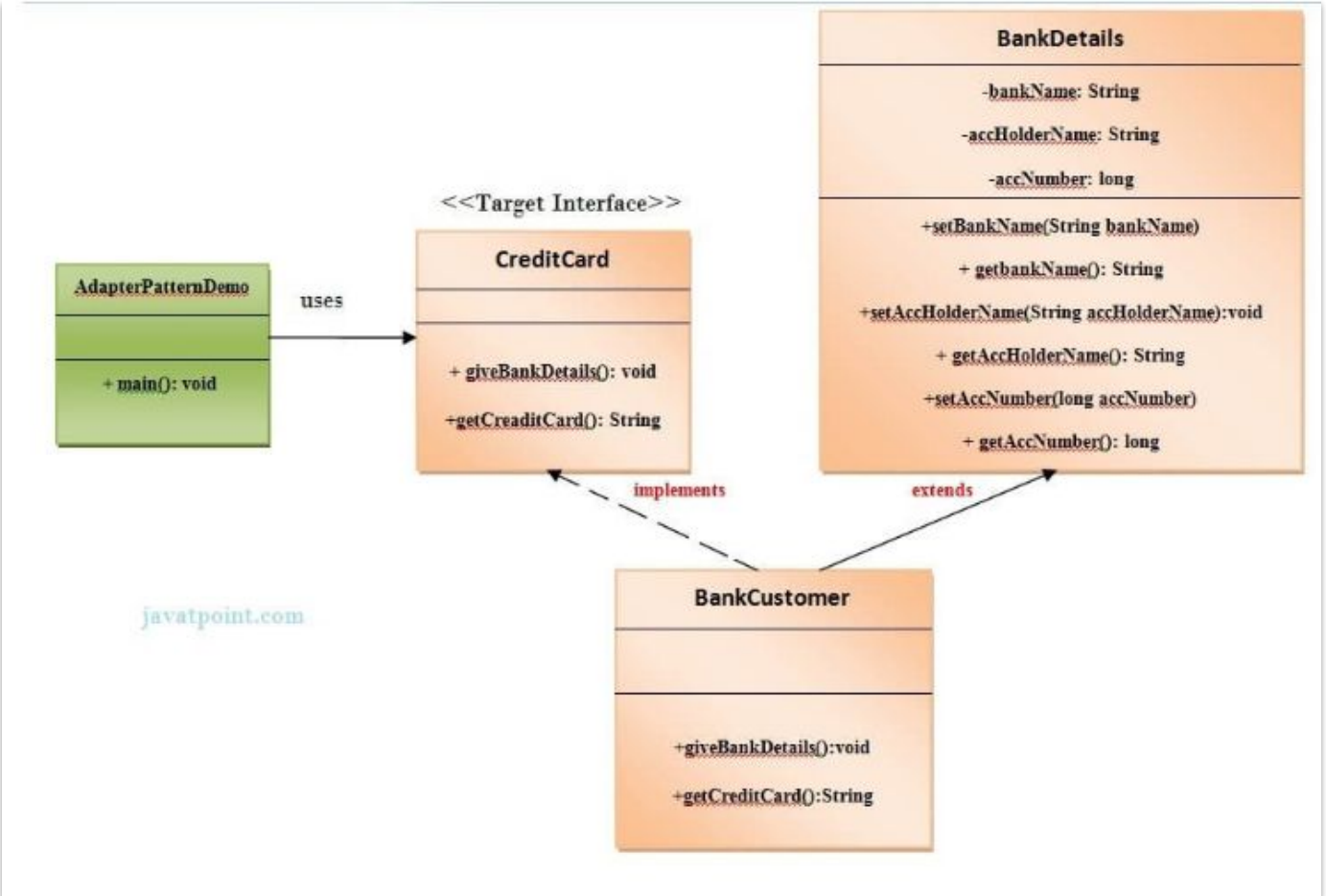
- Adapter Pattern "**converts the interface of a class into another interface that a client wants.**
- **AP provide provide** interface according to client requirement while using the services of a class with a different interface.
- Also known as Wrapper.

Specification of Adapter Pattern

There are the following specifications for the adapter pattern:

- **Target Interface:** This is the desired interface class which will be used by the clients.
- **Adapter class:** This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.
- **Adaptee class:** This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.
- **Client:** This class will interact with the Adapter class.

UML Diagram


BankDetails
-bankName: String
-accHolderName: String
-accNumber: long
+setBankName(String bankName)
+ getbankName(): String
+setAccHolderName(String accHolderName):void
+ getAccHolderName(): String
+setAccNumber(long accNumber)
+ getAccNumber(): long
<<Target Interface>>
CreditCard
+ giveBankDetails(): void
+getCreaditCard(): String
AdapterPatternDemo
+ main(): void
uses
implements
extends
BankCustomer
+giveBankDetails():void
+getCreditCard():String
javatpoint.com

Step 1: Design the interface

```
1. public interface CreditCard {
2.     public void giveBankDetails();
3.     public String getCreditCard();
4. }// End of the CreditCard interface.
```

Step 2: Create an Adaptee Class

```
1.  public class BankDetails{
2.      private String bankName;
3.      private String accHolderName;
4.      private long accNumber;
5.
6.      public String getBankName() {
7.          return bankName;     }
8.      public void setBankName(String bankName) {
9.          this.bankName = bankName;     }
10.     public String getAccHolderName() {
11.         return accHolderName;     }
12.     public void setAccHolderName(String accHolderName) {
13.         this.accHolderName = accHolderName;     }
14.     public long getAccNumber() {
15.         return accNumber;     }
16.     public void setAccNumber(long accNumber) {
17.         this.accNumber = accNumber;     } }// End of the BankDetails class.
```

- Define Private Data member
- Define their getter and setter function
- Getter function return value
- Setter properties are void that target data member

Step 3: Create an Adapter Class

```
1.  import java.io.BufferedReader; import java.io.InputStreamReader;
2.  public class BankCustomer extends BankDetails implements CreditCard {
3.   public void giveBankDetails(){
4.    try{    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
5.     System.out.print("Enter the account holder name :");
6.     String customername=br.readLine();
7.     System.out.print("\n");
8.       System.out.print("Enter the account number:");
9.     long accno=Long.parseLong(br.readLine());
10.    System.out.print("\n");
11.    System.out.print("Enter the bank name :");
12.    String bankname=br.readLine();
13.      setAccHolderName(customername);
14.    setAccNumber(accno);
15.    setBankName(bankname);
16.    }catch(Exception e){    e.printStackTrace();    } }
17.   @Override
18.   public String getCreditCard() {
19.    long accno=getAccNumber();
20.    String accholdername=getAccHolderName();
21.    String bname=getBankName();
22.     return ("The Account number "+accno+" of "+accholdername+" in "+bname+ "
23.       bank is valid and authenticated for issuing the credit card. ");   }
```

- Get all information from user
- Pass the information variable to public setter member of Adpatee class
  - setAccHolderName(customername);
- Override the getter member of adaptee class

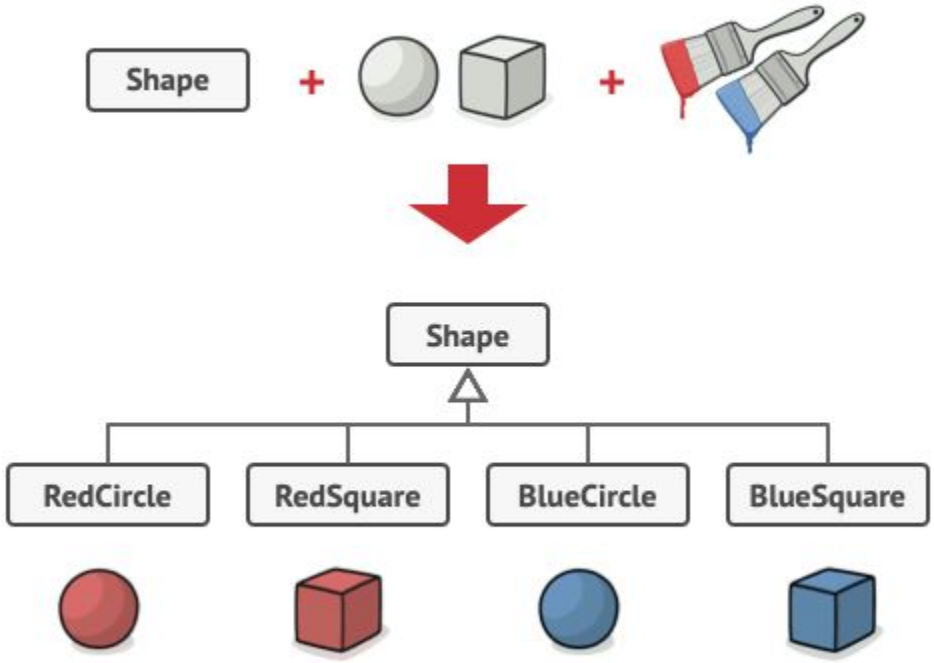
Step 4: Create an Adapter Pattern Demo Class

```
1.  public class AdapterPatternDemo {
2.   public static void main(String args[]){
3.    CreditCard targetInterface=new BankCustomer();
4.    targetInterface.giveBankDetails();
5.    System.out.print(targetInterface.getCreditCard());
6.   }
7.  }
```

Bridge Design Pattern

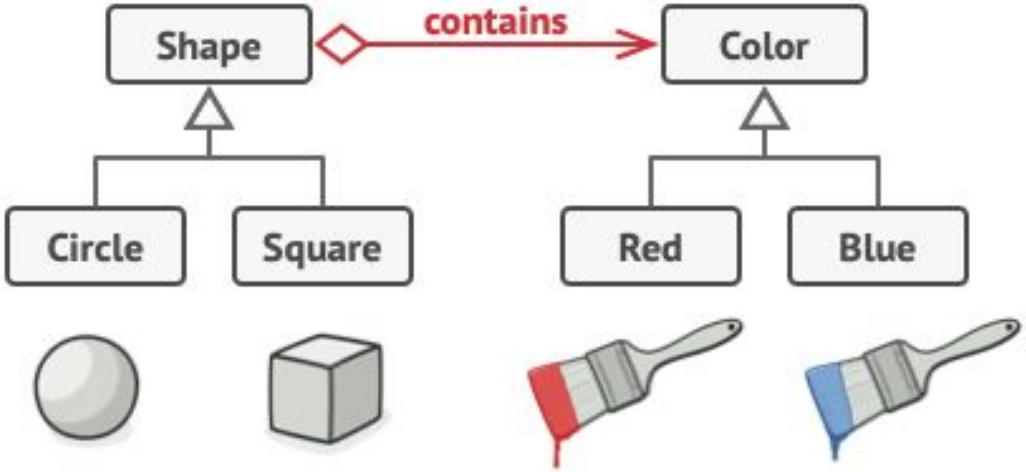
- A Bridge Pattern targets **decoupling the functional abstraction from the implementation so that the two can vary independently.**
- Bridge lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.
- it enables the separation of implementation from the interface.
- It improves the extensibility.
- It allows the hiding of implementation details from the client.

Problem?



Number of class combinations grows in geometric progression.

Solution



You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.

When to use Bridge Pattern

1. you want run-time binding of the implementation,
2. When you don't want a permanent binding between the functional abstraction and its implementation.
3. When both the functional abstraction and its implementation need to extended using sub-classes.
4. You want to share an implementation among multiple objects,
5. It is mostly used in those places where changes are made in the implementation does not affect the clients.

Step 1: Define Interface

```
1.  public interface Question {
2.      public void nextQuestion();
3.      public void previousQuestion();
4.      public void newQuestion(String q);
5.      public void deleteQuestion(String q);
6.      public void displayQuestion();
7.      public void displayAllQuestions();
8.  }
```

Class that use Interface

```
1.  public class JavaQuestions implements Question {
2.   private List <String> questions = new ArrayList<String>();
3.   private int current = 0;
4.   public JavaQuestions(){
5.     questions.add("What is class? ");     questions.add("What is interface? ");   questions.add("What is abstraction? "); }
6.   public void nextQuestion() {
7.     if( current <= questions.size()-1 )
8.     current++;
9.     System.out.print(current);   }
10.    public void previousQuestion() {
11.    if( current > 0 )
12.    current--;   }
13.    public void newQuestion(String quest) {
14.    questions.add(quest);   }
15.    public void deleteQuestion(String quest) {
16.    questions.remove(quest);   } }
17.  }// End of the JavaQuestions class.
```

Step 3: Create a class that use as a bridge between class and interface

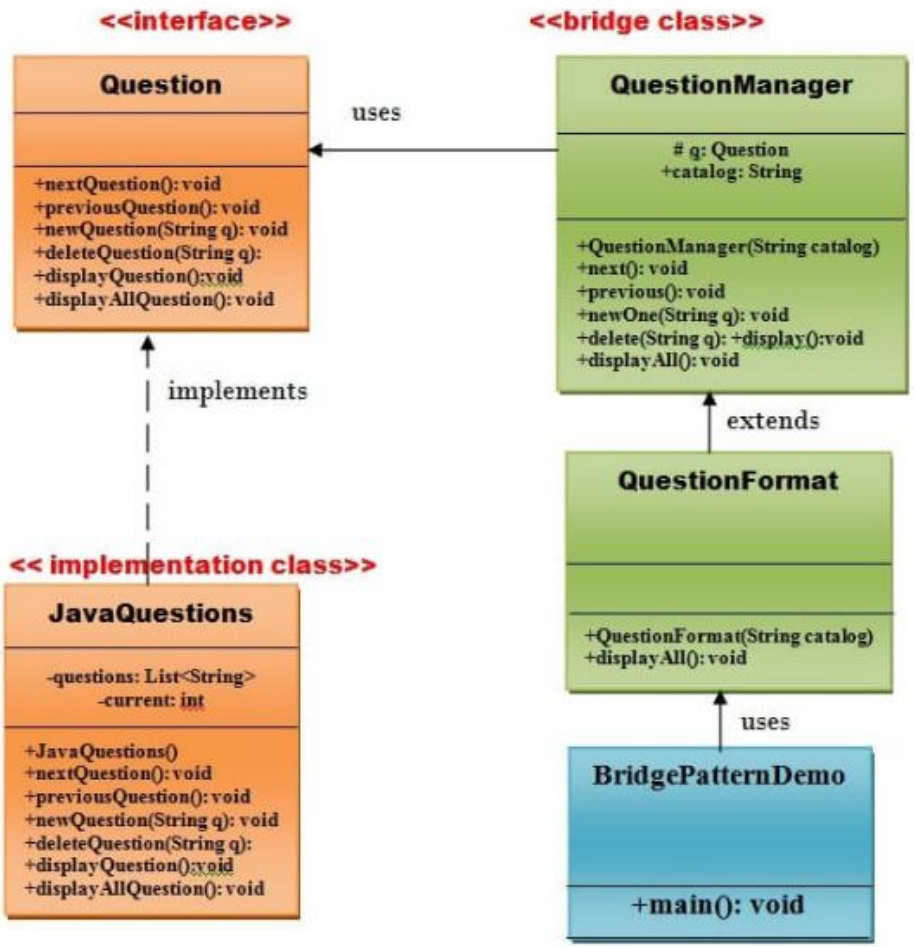
```
1.  public class QuestionManager  {
2.      protected Question q;
3.      public String catalog;
4.      public QuestionManager(String catalog) {
5.      this.catalog=catalog;     }
6.      public void next() {
7.      q.nextQuestion();     }
8.      public void previous() {
9.      q.previousQuestion();     }
10.     public void newOne(String quest) {
11.     q.newQuestion(quest);     }
12.     public void delete(String quest) {
13.     q.deleteQuestion(quest);     }
14.     public void display() {
15.     q.displayQuestion();     }
16.     public void displayAll() {
17.         System.out.println("Question Paper: " + catalog);
18.     q.displayAllQuestions();     } }// End of the QuestionManager class.
```

Step 4: Class that extends one main bridge class

```
1.  public class QuestionFormat extends QuestionManager {
2.      public QuestionFormat(String catalog){
3.          super(catalog);
4.      }
5.      public void displayAll() {
6.          System.out.println("\n-----------------------------------------------------");
7.          super.displayAll();
8.          System.out.println("-----------------------------------------------------");
9.      }
10. }// End of the QuestionFormat class.
```

Step 5: Define the class that play as main role of demonstration

```
1.  public class BridgePatternDemo {
2.     public static void main(String[] args) {
3.     QuestionFormat questions = new QuestionFormat("Java Programming Language");
4.     questions.q = new JavaQuestions();
5.        questions.delete("what is class?");
6.        questions.display();
7.     questions.newOne("What is inheritance? ");
8.
9.     questions.newOne("How many types of inheritance are there in java?");
10.    questions.displayAll();
11.    }
12. }//
```


<<interface>>
Question
+nextQuestion(): void
+previousQuestion(): void
+newQuestion(String q): void
+deleteQuestion(String q):
+displayQuestion():void
+displayAllQuestion(): void
uses
<<bridge class>>
QuestionManager
q: Question
+catalog: String
+QuestionManager(String catalog)
+next(): void
+previous(): void
+newOne(String q): void
+delete(String q): +display():void
+displayAll(): void
implements
extends
<< implementation class>>
JavaQuestions
-questions: List<String>
-current: int
+JavaQuestions()
+nextQuestion(): void
+previousQuestion(): void
+newQuestion(String q): void
+deleteQuestion(String q):
+displayQuestion():void
+displayAllQuestion(): void
QuestionFormat
+QuestionFormat(String catalog)
+displayAll(): void
uses
BridgePatternDemo
+main(): void

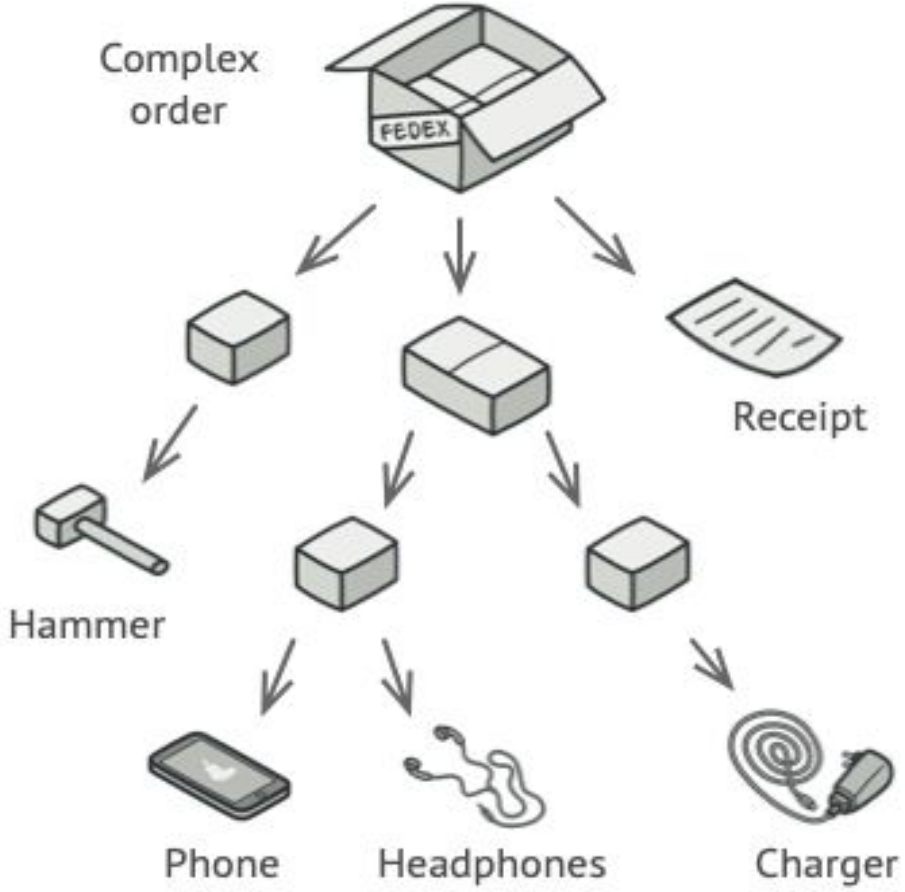
Implements vs extends

- The difference between an interface and a regular class is that in an interface you can not implement any of the declared methods. Only the class that "implements" the interface can implement the methods.
- Generally **implements** used for implementing an *interface* and **extends** used for *extension* of base class behaviour or *abstract* class.
- **extends**: A derived class can extend a base class. You may redefine the behaviour of an established relation. Derived class "***is a***" base class type
- **implements**: You are implementing a contract. The class implementing the interface "***has a***" capability.

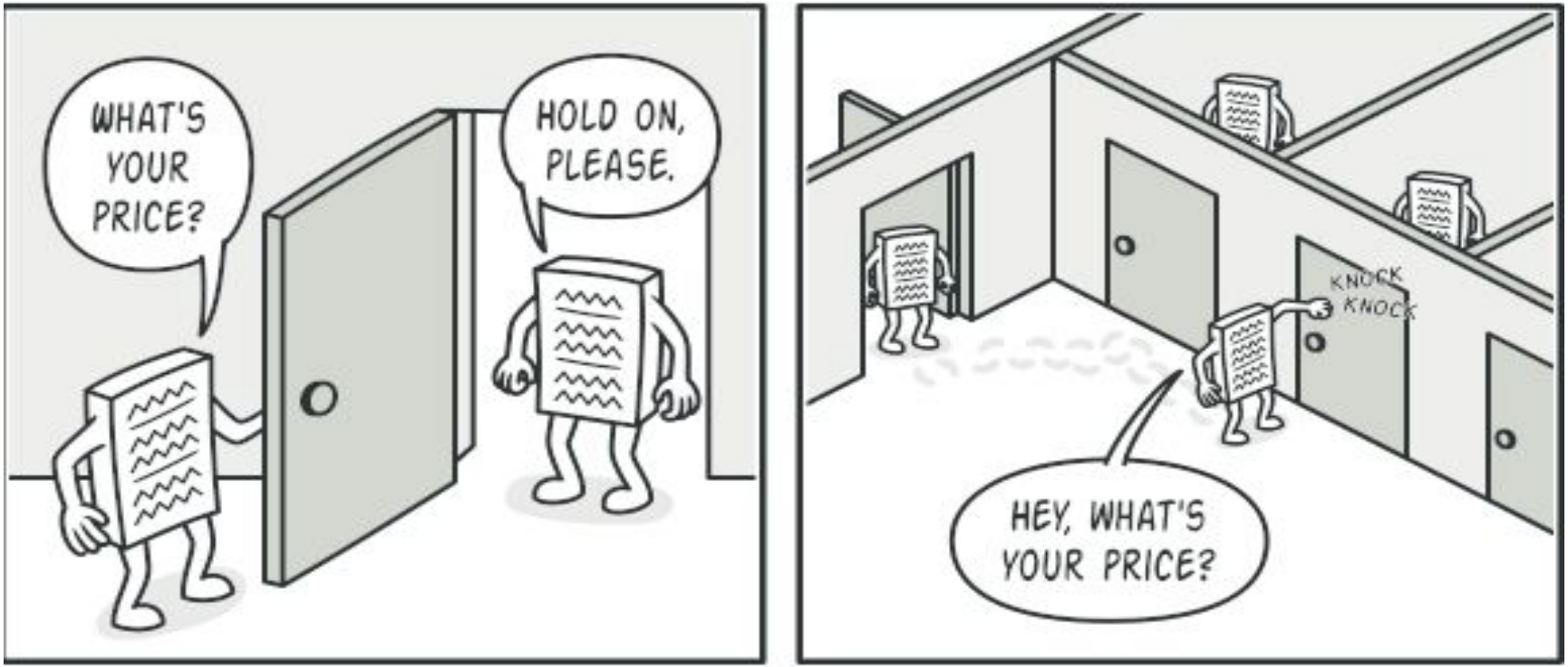
Composite Design Pattern

- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many "has a" up the "is a" hierarchy

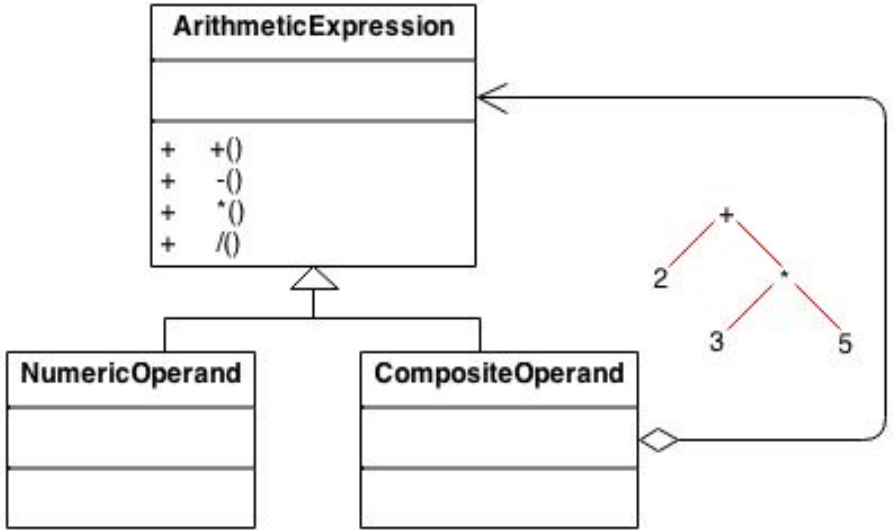
Problem


Complex order
FEDEX
Receipt
Hammer
Phone
Headphones
Charger

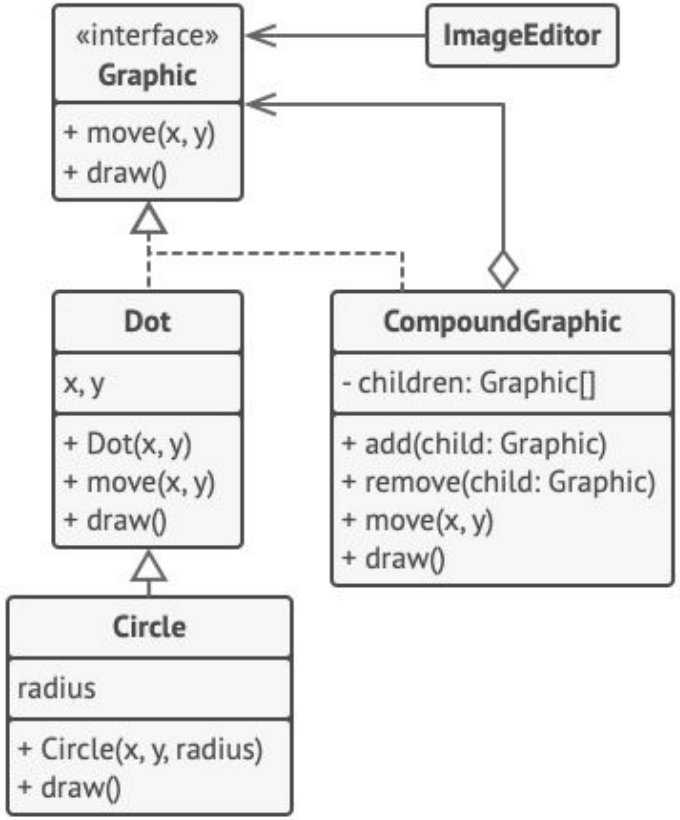
Ask or Wrap?


WHAT'S YOUR PRICE?
HOLD ON, PLEASE.
KNOCK KNOCK
HEY, WHAT'S YOUR PRICE?

Composite


ArithmeticExpression
+ +()
+ -()
+ *()
+ /()
NumericOperand
CompositeOperand
+
2
*
3
5

Composite



The geometric shapes editor example.

How to build

1. Make sure that the core model of your app can be represented as a tree structure.
2. Try to break it down into simple elements and containers.
3. Declare the component interface with a list of methods that make sense for both simple and complex components.
4. Create a leaf class to represent simple elements.
5. A program may have multiple different leaf classes.
6. Create a container class to represent complex elements , provide an array field for storing references to sub-elements.
7. The array must be able to store both leaves and containers, so make sure it’s declared with the component interface type.
8. While implementing the methods of the component interface, remember that a container is supposed to be delegating most of the work to sub-elements.
9. Finally, define the methods for adding and removal of child elements in the container.
   Keep in mind that these operations can be declared in the component interface.
10. This would violate the *Interface Segregation Principle* because the methods will be empty in the leaf class.
11. However, the client will be able to treat all the elements equally, even when composing the tree.

Implementation Example

https://drive.google.com/drive/folders/1nyY2NQmdC0dkkX3aOtVf_9ObRnyPkrhz

Decorator Design Pattern

1. This pattern attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.
2. It involves creating a set of decorator classes that are used to wrap concrete components.
3. This pattern is useful when you need to add functionality to objects in a flexible and reusable way

Characteristics

- This pattern promotes flexibility and extensibility in software systems by allowing developers to compose objects with different combinations of functionalities at runtime.
- It follows the open/closed principle.
- The Decorator Pattern is commonly used in scenarios where a variety of optional features or behaviors need to be added, such as in text formatting, graphical user interfaces, or customization of products like coffee or ice cream.

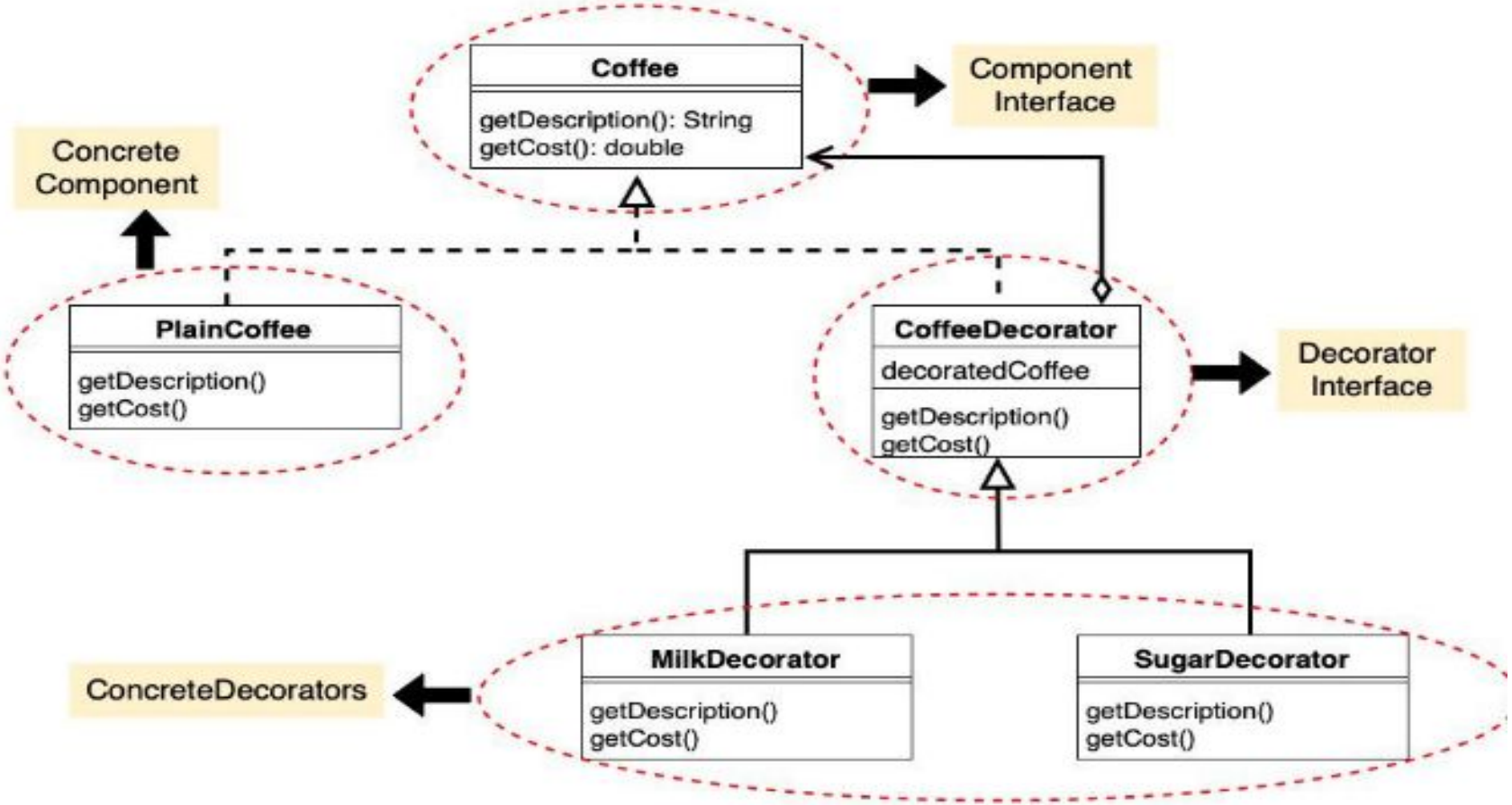
Some particular cases when we can use Decorator Design Pattern

Below are some of the use cases of Decorator Design Pattern:

- **Extending Functionality**: When you have a base component with basic functionality, but you need to add additional features or behaviors to it dynamically without altering its structure.
- **Multiple Combinations of Features**: When you want to provide multiple combinations of features or options to an object. Decorators can be stacked and combined in different ways to create customized variations of objects, providing flexibility to users.
- **Legacy Code Integration**: When working with legacy code or third-party libraries where modifying the existing codebase is not feasible or desirable, decorators can be used to extend the functionality of existing objects without altering their implementation.
- **GUI Components**: In graphical user interface (GUI) development, decorators can be used to add additional visual effects, such as borders, shadows, or animations, to GUI components like buttons, panels, or windows.
- **Input/Output Streams**: Decorators are commonly used in input/output stream classes in languages like Java. They allow you to wrap streams with additional functionality such as buffering, compression, encryption, or logging without modifying the original stream classes.

1.

Class Diagram of Decorator Design Pattern



Assignment

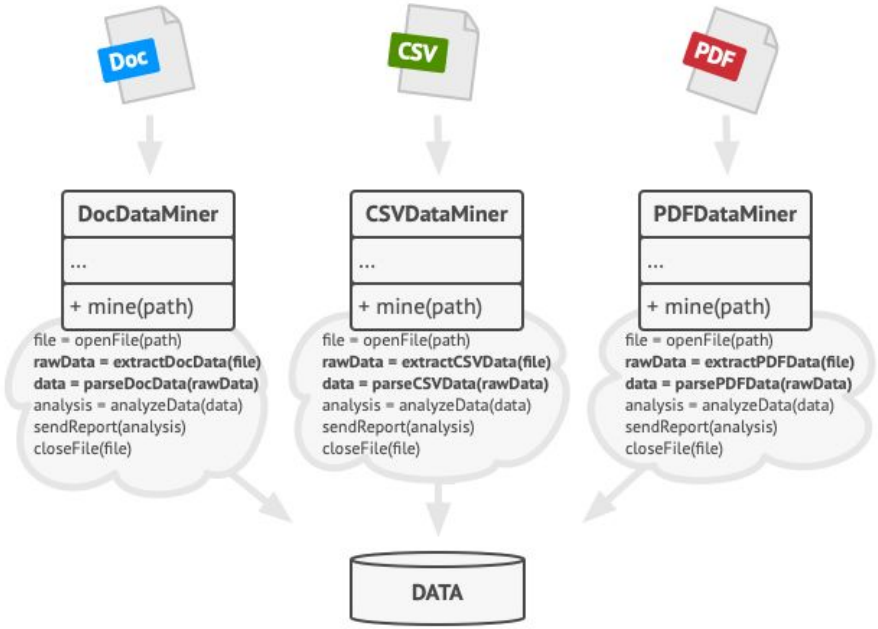
1. A student of PUCIT has a choice to select the FYP domain amongst the following
    a. Machine Learning
    b. Natural Language Processing
    c. Artificial Intelligence
2. Student must have CGPA of 2.75, Must need to provide their detail of core subject passing like OOP, DSA and DB.
3. Apply any structural Design pattern of your choice

Scope	Creational Patterns	Structural Patterns	Behavioral Patterns
Class Patterns	▪ **Factory method**	▪ **Adapter**	▪ Interpreter ▪ **Template method**
Object Patterns	▪ Abstract factory ▪ Builders ▪ Prototype ▪ **Singleton**	▪ **Adapter** ▪ **Bridge** ▪ **Composite** ▪ **Decorator** ▪ **Facade** ▪ **Proxy** ▪ Flyweight	▪ Chain of responsibility ▪ Command ▪ **Iterator** ▪ Mediator ▪ Memento ▪ **Observer** ▪ State

Behavioral Pattern

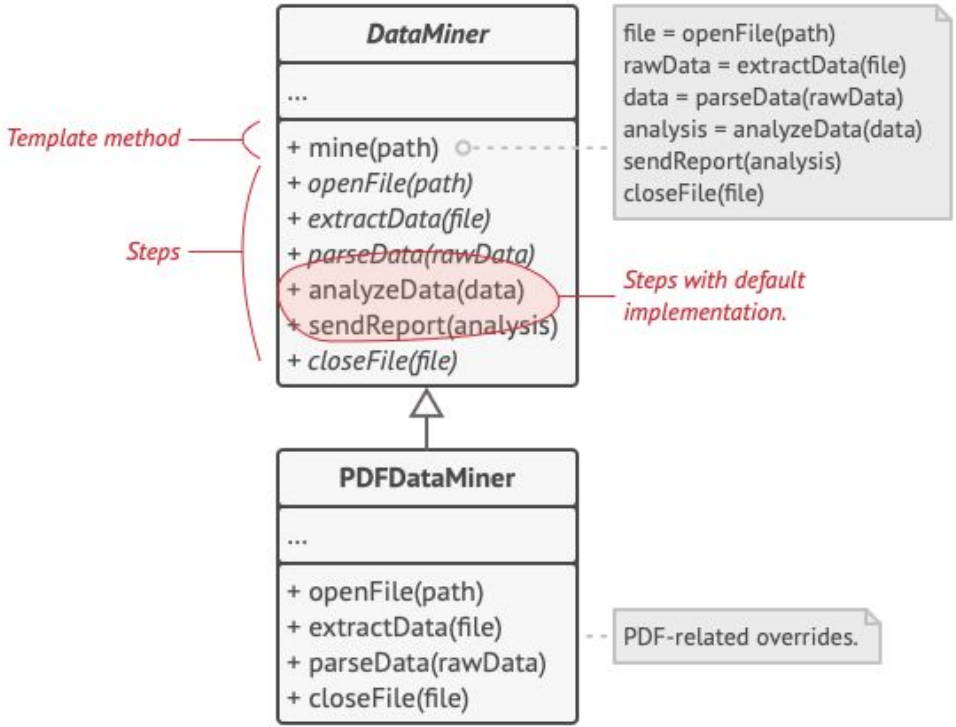
Template Design pattern

- Template pattern defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
- it can provide default implementation that might be common for all or some the subclasses.
- To make sure that subclasses don't override the template method, we should make it final.



Data mining classes contained a lot of duplicate code.

Template Design Pattern



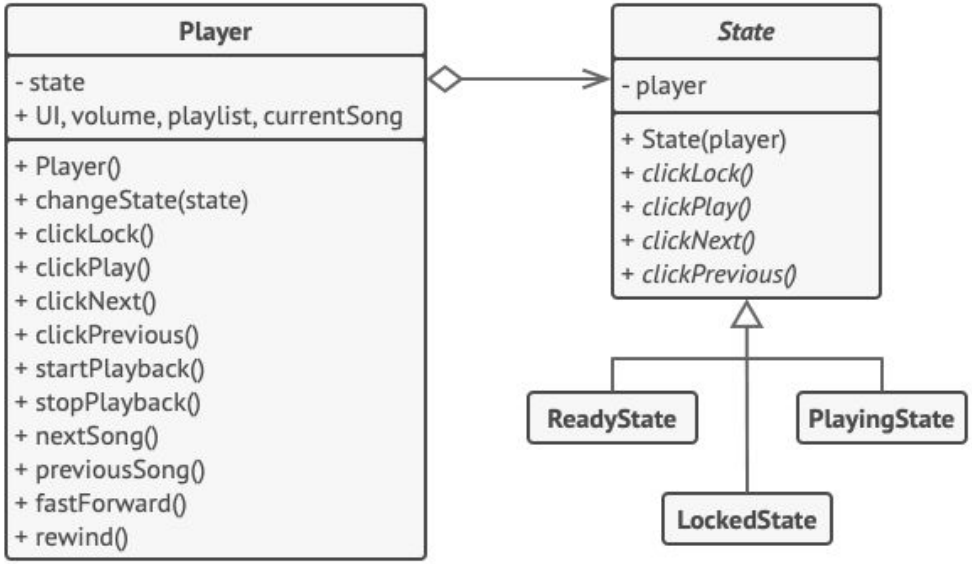
State Design Pattern

- State is a behavioral design pattern that lets an object alter its behavior when its internal state changes.
- It appears as if the object changed its class.
- State machines are usually implemented with lots of conditional statements (`if` or `switch`).
- The State pattern suggests that you create new classes for all possible states of an object .
- Extract all state-specific behaviors into these classes.

When to use

- When the phone is unlocked, pressing buttons leads to executing various functions.
- When the phone is locked, pressing any button leads to the unlock screen.
- When the phone's charge is low, pressing any button shows the charging screen.

State Design Pattern

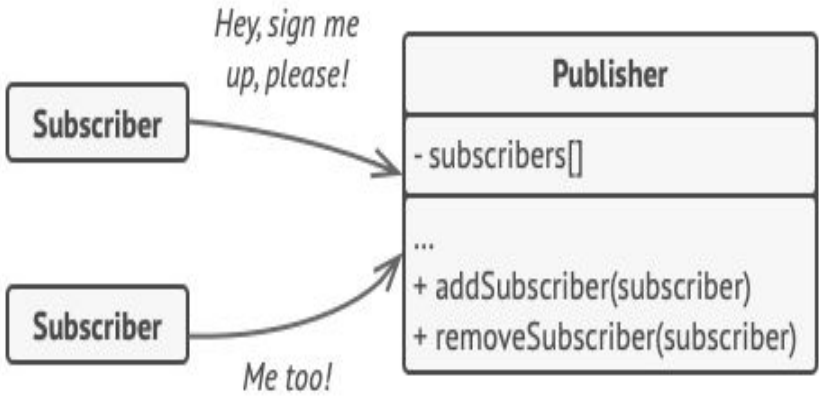


Example of changing object behavior with state objects.

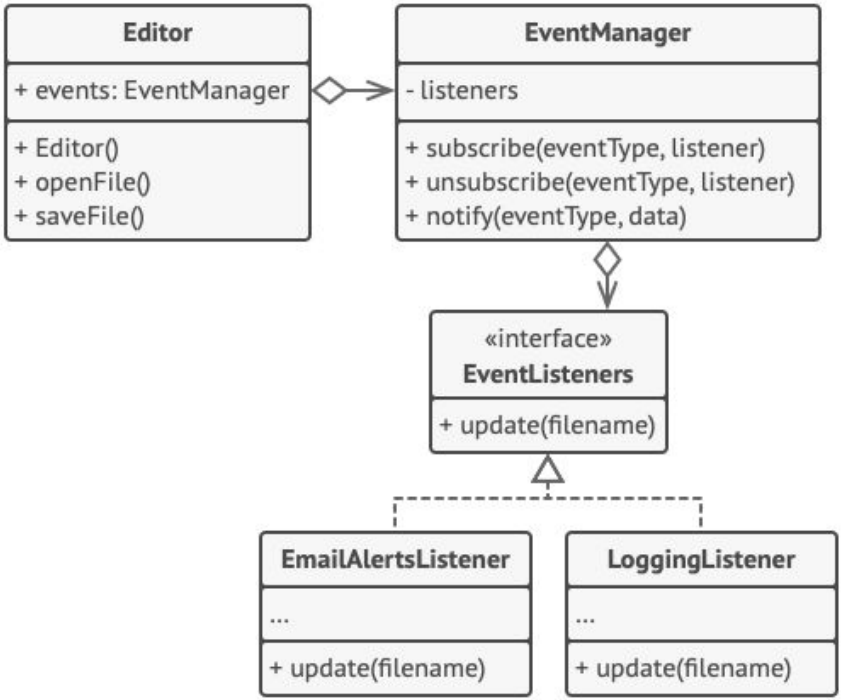
Observer Design Pattern

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

- Define a one-to-many dependency between objects to know change of state with its dependents.
- One object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The "View" part of Model-View-Controller.



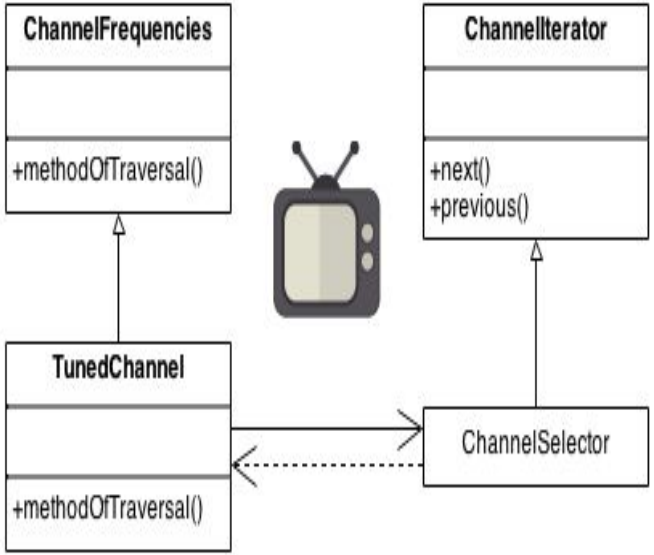
Observer



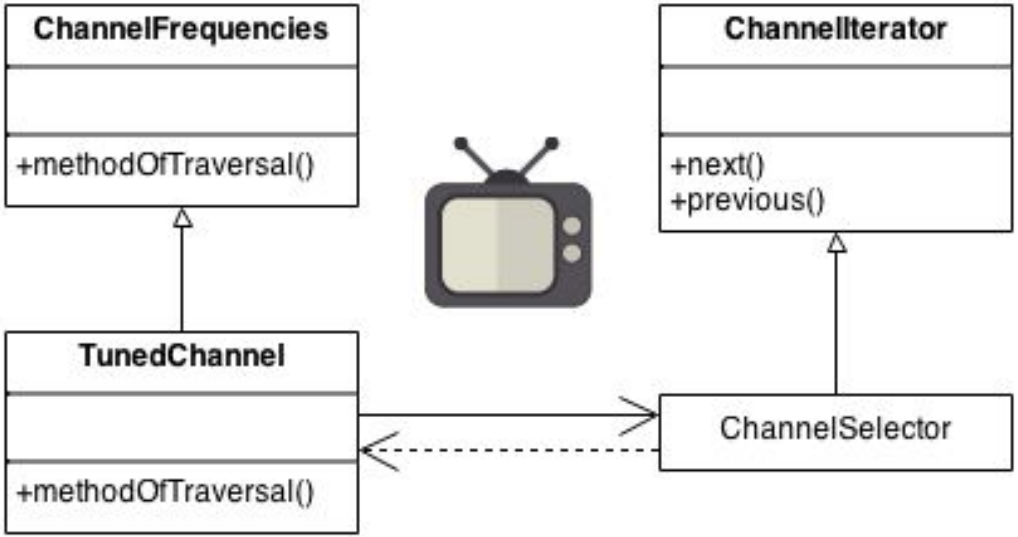
Notifying objects about events that happen to other objects

Iterator

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.
- Promote to "full object status" the traversal of a collection.
- Polymorphic traversal



Iterator

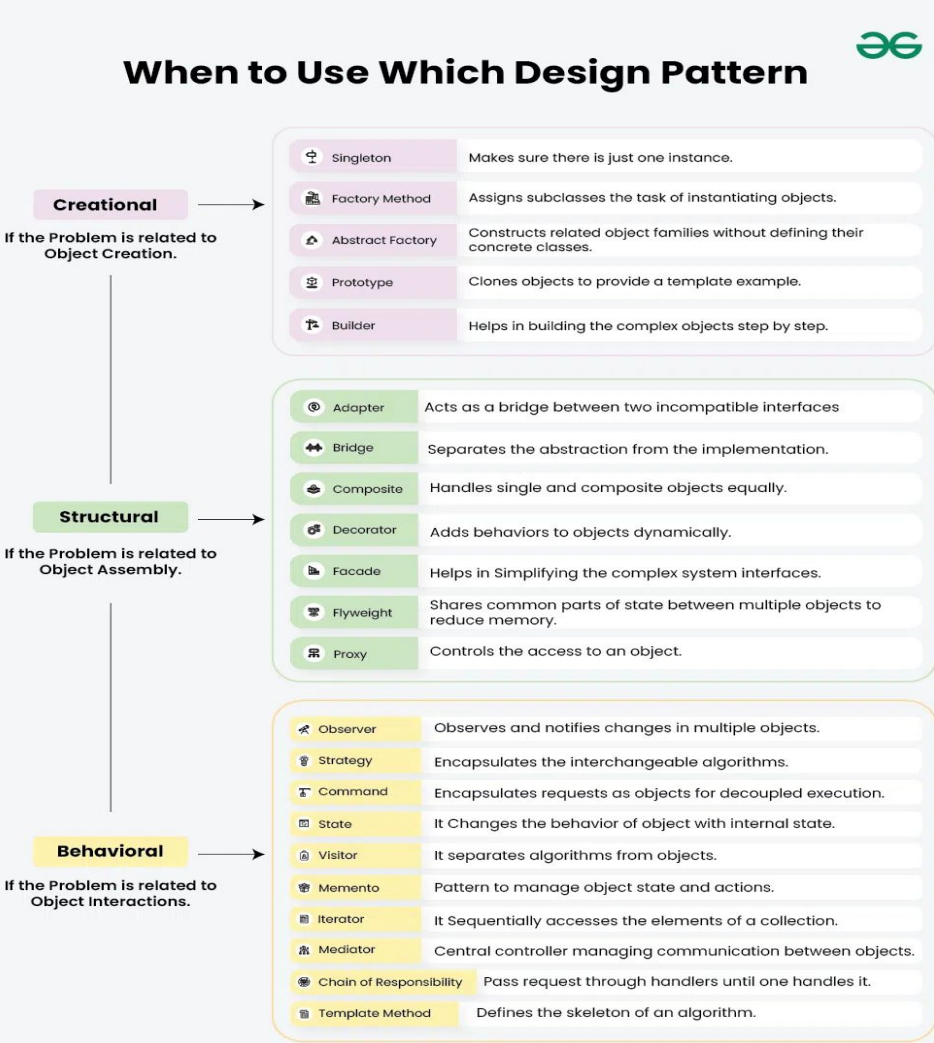

ChannelFrequencies
+methodOfTraversal()
ChannelIterator
+next()
+previous()
TunedChannel
+methodOfTraversal()
ChannelSelector

Iterator Steps

1. Add a `create_iterator()` method to the "collection" class, and grant the "iterator" class privileged access.
2. Design an "iterator" class that can **encapsulate traversal of the** "collection" class.
3. Clients ask the collection object to create an iterator object.
4. Clients use the `first()`, `is_done()`, `next()`, and `current_item()` protocol to access the elements of the collection class.

Comparison


When to Use Which Design Pattern
Creational
If the Problem is related to Object Creation.
Singleton
Makes sure there is just one instance.
Factory Method
Assigns subclasses the task of instantiating objects.
Abstract Factory
Constructs related object families without defining their concrete classes.
Prototype
Clones objects to provide a template example.
Builder
Helps in building the complex objects step by step.
Structural
If the Problem is related to Object Assembly.
Adapter
Acts as a bridge between two incompatible interfaces
Bridge
Separates the abstraction from the implementation.
Composite
Handles single and composite objects equally.
Decorator
Adds behaviors to objects dynamically.
Facade
Helps in Simplifying the complex system interfaces.
Flyweight
Shares common parts of state between multiple objects to reduce memory.
Proxy
Controls the access to an object.
Behavioral
If the Problem is related to Object Interactions.
Observer
Observes and notifies changes in multiple objects.
Strategy
Encapsulates the interchangeable algorithms.
Command
Encapsulates requests as objects for decoupled execution.
State
It Changes the behavior of object with internal state.
Visitor
It separates algorithms from objects.
Memento
Pattern to manage object state and actions.
Iterator
It Sequentially accesses the elements of a collection.
Mediator
Central controller managing communication between objects.
Chain of Responsibility
Pass request through handlers until one handles it.
Template Method
Defines the skeleton of an algorithm.


When to Use Which Design Pattern
Creational
If the Problem is related to Object Creation.
Singleton
Makes sure there is just one instance.
Factory Method
Assigns subclasses the task of instantiating objects.
Abstract Factory
Constructs related object families without defining their concrete classes.
Prototype
Clones objects to provide a template example.
Builder
Helps in building the complex objects step by step.
Structural
If the Problem is related to Object Assembly.
Adapter
Acts as a bridge between two incompatible interfaces
Bridge
Separates the abstraction from the implementation.
Composite
Handles single and composite objects equally.
Decorator
Adds behaviors to objects dynamically.
Facade
Helps in Simplifying the complex system interfaces.
Flyweight
Shares common parts of state between multiple objects to reduce memory.
Proxy
Controls the access to an object.
Behavioral
If the Problem is related to Object Interactions.
Observer
Observes and notifies changes in multiple objects.
Strategy
Encapsulates the interchangeable algorithms.
Command
Encapsulates requests as objects for decoupled execution.
State
It Changes the behavior of object with internal state.
Visitor
It separates algorithms from objects.
Memento
Pattern to manage object state and actions.
Iterator
It Sequentially accesses the elements of a collection.
Mediator
Central controller managing communication between objects.
Chain of Responsibility
Pass request through handlers until one handles it.
Template Method
Defines the skeleton of an algorithm.

How to identify design pattern from Code ?

- You need to identify Cohesion
- Coupling
- Inheritance
- Object
- Parent Class
- Child Class
- Interface

Where to Start

- Using AST (automated)
- Using SQL Query (automated)https://docs.google.com/document/d/1GwDUsBv3FyjadzZpdBZhARUWfxLpKqLf/edit?usp=sharing&ouid=113679652048052408192&rtpof=true&sd=true
- NLP Parsers(automated)
- Class diagram (manually)