LAB MANUAL

Course: CSE354 Design Patterns

Department of Computer Science

COMSATS University Islamabad, Abbottabad Campus

Table of Contents

LAB 01: OOPs Concepts and Objects Relationships

Purpose

The purpose of this lab is to:

- Review the OOP concepts (OOP) for improving code readability and reusability by defining how to structure an object oriented programs efficiently.
- Implement the six common relationships among objects in object oriented software.

Outcomes

The expected outcome of this lab activity is students' ability to:

- Utilize the basic OOPs concepts in designing object oriented software's
- Identify the scalability and maintenance provided by these concepts
- Understand object relations in code

Introduction

Our journey in computer programming usually starts with function oriented programming with a structural language like C. To design more efficient algorithm, we learned and utilized the data structures, but to design more scalable and maintainable software we need to be closed to the real world as much as possible. Therefore, students learn OOPs concepts, but mostly lacks in their utilization. This lab will help you to start explore it to a bit deep and start utilizing these concepts in software design. In real world objects relates to each other, but in code how these relationships are realized in code is also the focus of this lab.

Activity 1

Carefully read the following articles and then implement the coding examples.

- 6 OOP Concepts in Java with examples
  - from https://raygun.com/blog/oop-concepts-java/

OOP concepts allow us to create specific interactions between Java objects. They make it possible to reuse code without creating security risks or making a Java program less readable. While implementing the code examples, you should be able to understand that:

Abstraction in Java:

- Hides the underlying complexity of data
- Helps avoid repetitive code

- Presents only the signature of internal functionality
- Gives flexibility to programmers to change the implementation of the abstract behavior
- Partial abstraction (0-100%) can be achieved with abstract classes
- Total abstraction (100%) can be achieved with interfaces

Encapsulation in Java:

- Restricts direct access to data members (fields) of a class.
- Fields are set to private
- Each field has a getter and setter method
- Getter methods return the field
- Setter methods let us change the value of the field

Polymorphism in Java:

- The same method name is used several times.
- Different methods of the same name can be called from the object.
- All Java objects can be considered polymorphic (at the minimum, they are of their own type and instances of the Object class).
- Example of static polymorphism in Java is method overloading.
- Example of dynamic polymorphism in Java is method overriding.

Inheritance in Java:

- A class (child class) can extend another class (parent class) by inheriting its features.
- Implements the DRY (Don't Repeat Yourself) programming principle.
- Improves code reusability.
- Multilevel inheritance is allowed in Java (a child class can have its own child class as well).
- Multiple inheritances are not allowed in Java (a class can't extend more than one class).

Activity 2

In real world different objects share their stats and collaborate, use, serve or delegate responsibilities with each other. The same also happens in object oriented software since they almost replicate their solution closed to the real word. Let's omit the beginner level details of object oriented programming and start with first understanding the most common relationships among objects which are:

1. Dependency
2. Inheritance (partial abstraction)
3. Interface implementation (full abstraction)
4. Association
5. Aggregation
6. Composition

While doing the implementation in Activity 1, You need to understand about the relations that:

Association in Java:

- Two separate classes are associated through their objects.
- The two classes are unrelated and each can exist without the other one.
- Can be a one-to-one, one-to-many, many-to-one, or many-to-many relationship.

Aggregation in Java:

- One-directional association.
- Represents a HAS-A relationship between two classes.
- Only one class is dependent on the other.

Composition in Java:

- A restricted form of aggregation
- Represents a PART-OF relationship between two classes
- Both classes are dependent on each other
- If one class ceases to exist, the other can't survive alone

Home Activities

HW 1: To understand more about dependencies read the following article.

- http://tutorials.jenkov.com/ood/understanding-dependencies.html

HW2: Select at least two items from the following list to find their answers and them implement them. Share your finding with the class through your suggested mechanism by the class teacher:

1. Overloading of the main method is possible or not?
2. What is returned by the constructor, and how you can identify it from its declaration?
3. Can we create a program without main method? How many main methods are allowed in Java Programs?
4. What are the six ways to use this keyword?
5. Prove that multiple inheritance is not supported in Java?
6. When to use aggregation and not composition and vice versa?
7. How to override the static method?
8. Is overloading of the main method possible?
9. Give any real world example of using the covariant return type?
10. Discuss different usages of Java super keyword?
11. What is instance initializer block and why we use it?
12. What are the different usages of final variable?
13. What is a marker or tagged interface?
14. What is runtime polymorphism or dynamic method dispatch?
15. What is the difference between static and dynamic binding?
16. How down-casting is possible in Java?
17. What is the purpose of a private constructor?
18. What is object cloning?
19. Differentiate shallow copy from deep copy and implement each with an example.
20. In your class context find the implementation of OOPs concepts.

Assignment Deliverables

You will have to create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 02: Singleton and Observer

Purpose

The purpose of this lab is to understand and apply one creational pattern "Singleton" and one behavioral pattern "Observer" in their respective problem context.

Outcomes

The expected outcome of this lab activity is the student's ability to:

- Know Thread Safe Code
- Know Static Vs Dynamic Context of Object
- Find applicability and implementation of Singleton pattern
- Find applicability and implementation of Observer pattern
- Know Thread Safe Code

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Singleton and Observer patters.

- For Singleton Patthern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Singleton
- For Observer Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Observer

Note: In case of physical labs this activity is optional but recommended.

Activity 2

Run all the three Singleton design pattern examples in the Java IDE (recommended NetBeans) from the following URL

https://en.wikipedia.org/wiki/Singleton_pattern

Activity 3

Run two more a bit complicated examples from the GitHub on following URLs:

- https://github.com/sshpk/dp_fall20/tree/master/SingeltonFA20 and

- https://github.com/sshpk/dp_fall20/tree/master/SingeltonFA20Homework

Activity 4

Figure 1.7 shows another example of shopping cart contents and checkout option that we discussed in the previous lab session. Redraw this interface by addressing the existing issues.

1) No “Continue shopping" button
2)

Activity 5

Understand and Run the following examples of Observer design pattern:

- Observer pattern - Wikipedia
- dp_fall20/ObserverFA20/src/pk/cuiatd/dp/observer1 at master · sshpk/dp_fall20 · GitHub
- dp_fall20/ObserverFA20/src/pk/cuiatd/dp/observer2 at master · sshpk/dp_fall20 · GitHub

Home Activities

Home Work 1:

Convert the second example in Activity 3 of SingeltonFA20Homework to Thread Safe implementation.

Home Work 2:

Modify Wikipedia example to covert the anonymous concrete observer to a proper concrete observer class.

Home Work 3:

Modify the third example (observer2), Such that it notifies the availability via update method to the subscriber and then the message is displayed according to the value in the update method parameter.

Instead of passing the availability in the update method, Convert now to pass the instance of the WhiteShirt(Publisher) to its subscriber and then they act accordingly.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 03: Decorator and Factory Method

Purpose

The purpose of this lab is to understand how you can attach additional responsibilities to an object dynamically via restructuring its representation using the Decorator design pattern. In second session of the lab we learn about Factory Method that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Outcomes

The expected outcome of this lab activity is the student's ability to:

- Restructure the representation of an object with the help of Decorator to attach additional responsibility dynamically to an object.
- Provides an interface for creating objects in a superclass and allow subclasses to alter the type of the object.

Decorator Implementation Guidelines

1. Make sure your business domain can be represented as a primary component with multiple optional layers over it.
2. Figure out what methods are common to both the primary component and the optional layers. Create a component interface and declare those methods there.
3. Create a concrete component class and define the base behavior in it.
4. Create a base decorator class. It should have a field for storing a reference to a wrapped object. The field should be declared with the component interface type to allow linking to concrete components as well as decorators. The base decorator must delegate all work to the wrapped object.
5. Make sure all classes implement the component interface.
6. Create concrete decorators by extending them from the base decorator. A concrete decorator must execute its behavior before or after the call to the parent method (which always delegates to the wrapped object).
7. The client code must be responsible for creating decorators and composing them in the way the client needs.

Factory Method Implementation Guidelines

1. Make all products follow the same interface. This interface should declare methods that make sense in every product.
2. Add an empty factory method inside the creator class. The return type of the method should match the common product interface.

3. In the creator's code find all references to product constructors. One by one, replace them with calls to the factory method, while extracting the product creation code into the factory method.
4. You might need to add a temporary parameter to the factory method to control the type of returned product.
5. At this point, the code of the factory method may look pretty ugly. It may have a large switch operator that picks which product class to instantiate. But don't worry, we'll fix it soon enough.
6. Now, create a set of creator subclasses for each type of product listed in the factory method. Override the factory method in the subclasses and extract the appropriate bits of construction code from the base method.
7. If there are too many product types and it doesn't make sense to create subclasses for all of them, you can reuse the control parameter from the base class in subclasses.
8. For instance, imagine that you have the following hierarchy of classes: the base Mail class with a couple of subclasses: AirMail and GroundMail; the Transport classes are Plane, Truck and Train. While the AirMail class only uses Plane objects, GroundMail may work with both Truck and Train objects. You can create a new subclass (say TrainMail) to handle both cases, but there's another option. The client code can pass an argument to the factory method of the GroundMail class to control which product it wants to receive.
9. If, after all of the extractions, the base factory method has become empty, you can make it abstract. If there's something left, you can make it a default behavior of the method.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Decorator and Factory Method.

- For Decorator Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Decoratoy
- For Factory Method Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Factory-Method

Note: In case of physical labs this activity is optional but recommended.

Activity 2

The text book motivational code example for window/scrolling scenario is available on the following link. You need to run the version in Java which is quite simple.

- https://en.wikipedia.org/wiki/ Decorator_pattern

Activity 3

Run one another very simple example about the Decorator pattern related to decorating geometrical shapes from the following link:

- https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

Activity 4

Imagine that you're working on a notification library which lets other programs notify their users about important events. You need to first focus on the problem in this context and then understand its solution using the Decorator pattern from the following link:

- https://refactoring.guru/design-patterns/decorator

For Factory Method

Activity 5

Run the example on

- https://github.com/sshpk/dp_fall20/tree/master/FactoryMethodFA20

Activity 6

Also run the following examples of factory method:

- Design Patterns: Factory Method in Java (refactoring.guru)
- Factory method pattern - Wikipedia Modify the C# example into Java

Home Activities

Update the example with following additions:

Home Work 1:

Modify the example in Activity 3 to convert the output to "Shape : Circle with Red color" when it is decorated and "Shape: Circle" when not decorated.

Home Work 2:

For example, in Activity 3, Add one more decorator ThickBorderDecorator, which will decorate the shape with thick border.

Home Work 3:

Print shapes without decoration, with red color, with thick border only and then with both red color and thick border decoration in the Demo class.

Home Work 4:

For Activity 5, Add one another TriangleGeometry class to provide the Triangle factory method.

Home Work 5:

Instead of using four classifiers i.e. (Geometry, SquareGeometry, CircleGeometry etc.), We can implement factory method with a single method with any polymorphic behavior by passing it the type of object we need and then the method returns the desired object. Provide this single factory method implementation for the same example.

. **Home Work 6**:

Think on how can you utilize the Factory Method in your final year project and provide its implementation. (Optional)

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 04: Strategy and Chain of Responsibility

Purpose

The purpose of this lab is to understand two more design patterns “Strategy” and “Chain of Responsibility”. By Strategy pattern you can define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. While “Chain of Responsibility” will enable you to you pass requests along a chain of handlers.

Outcomes

The expected outcome of this lab activity is the student’s ability to:

- Understand and implement the Strategy design pattern.
- Understand and implement the Chain of Responsibility design pattern.

Strategy Implementation Guidelines

1. The client code must be responsible for creating decorators and composing them in the way the client needs. In the context class, identify an algorithm that’s prone to frequent changes. It may also be a massive conditional that selects and executes a variant of the same algorithm at runtime.
2. Declare the strategy interface common to all variants of the algorithm.
3. One by one, extract all algorithms into their own classes. They should all implement the strategy interface.
4. In the context class, add a field for storing a reference to a strategy object. Provide a setter for replacing values of that field. The context should work with the strategy object only via the strategy interface. The context may define an interface which lets the strategy access its data.
5. Clients of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job.

Chain of Responsibility Implementation Guidelines

1. Declare the handler interface and describe the signature of a method for handling requests.

   Decide how the client will pass the request data into the method. The most flexible way is to convert the request into an object and pass it to the handling method as an argument.

2. To eliminate duplicate boilerplate code in concrete handlers, it might be worth creating an abstract base handler class, derived from the handler interface.

This class should have a field for storing a reference to the next handler in the chain. Consider making the class immutable. However, if you plan to modify chains at runtime, you need to define a setter for altering the value of the reference field.

You can also implement the convenient default behavior for the handling method, which is to forward the request to the next object unless there's none left. Concrete handlers will be able to use this behavior by calling the parent method.

3. One by one create concrete handler subclasses and implement their handling methods. Each handler should make two decisions when receiving a request:
    a. Whether it'll process the request.
    b. Whether it'll pass the request along the chain.

4. The client may either assemble chains on its own or receive pre-built chains from other objects. In the latter case, you must implement some factory classes to build chains according to the configuration or environment settings.
5. The client may trigger any handler in the chain, not just the first one. The request will be passed along the chain until some handler refuses to pass it further or until it reaches the end of the chain.
6. Due to the dynamic nature of the chain, the client should be ready to handle the following scenarios:

    - The chain may consist of a single link.
    - Some requests may not reach the end of the chain.
    - Others may reach the end of the chain unhandled.

1.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Strategy and Chain of Responsibility design patterns.

- For Strategy Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Strategy
- For Chain of Responsibility Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Chain-of-Responsibility

Note: In case of physical labs this activity is optional but recommended.

Activity 2

You need to run the following example of Strategy in Java:

- https://github.com/sshpk/dp_fall20/tree/master/StrategyFA20/src/pk/cuiatd/dp/strategy

Steps

1. Create a package with name "Strategy"
2. Add all the classes from the Github example to it.
3. Debug the Demo.java to understand it.
4. Complete the Lab Tasks

Activity 3

For the Chain of Responsibility design pattern, Run the following examples:

- Loan Approval example (Explained in Video)
- Files Handler
- Dispenser Notes Example

Home Activities

Update the example of Activity 3 with following additions:

Home Work 1:

Modify the context (ShoppingCart) to accept payment strategy in its constructor.

Home Work 2:

Add another payment strategy via Easypaisa that requires name and phone number of the account holder.

Home Work 3:

Modify all payment strategy such that if the cart amount is more than 2000 then give a 10% discount to the user.

Home Work 4:

For the first example in Activity 3,

1. Add interest rate to the loan class and modify the lab implementation such that limit

increased by the interest rate. for example, if limit is 10000 and interest rate is 5% then interest based limit will be 10500 and decision will be taken for loan approval on the interest based limit.

2. Instead of using one Chain Handler as in example of 1, You are required to create multiple handlers for each operation of the calculator i.e. Adder, Subtractor, Multiplier and Divider which has a compute polymorphic method that accepts two parameters a and b of type integer and perform the operation or pass it to the next handler in the chain. The result is return to the Demo class.

Home Work 5:

1. Find at least one situation where Chain of Responsibility pattern can be applied.
2. Find at least one situation where Strategy pattern can be applied.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher

LAB 05: Abstract Factory and Iterator

Purpose

This lab will enable the students to produce families of related objects without specifying their concrete classes using the Abstract Factory and traverse the elements of a collection without exposing its underlying representation using Iterator.

Outcomes

After completing this lab, students will be able to apply:

- **Abstract Factory** to produce families of related objects without specifying their concrete classes.
- **Iterator** to traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Implementation Guidelines for Abstract Factory

1. Map out a matrix of distinct product types versus variants of these products.
2. Declare abstract product interfaces for all product types. Then make all concrete product classes implement these interfaces.
3. Declare the abstract factory interface with a set of creation methods for all abstract products.
4. Implement a set of concrete factory classes, one for each product variant.
5. Create factory initialization code somewhere in the app. It should instantiate one of the concrete factory classes, depending on the application configuration or the current environment. Pass this factory object to all classes that construct products.
6. Scan through the code and find all direct calls to product constructors. Replace them with calls to the appropriate creation method on the factory object.

Implementation Guidelines for Iterator

1. Declare the iterator interface. At the very least, it must have a method for fetching the next element from a collection. But for the sake of convenience you can add a couple of other methods, such as fetching the previous element, tracking the current position, and checking the end of the iteration.
2. Declare the collection interface and describe a method for fetching iterators. The return type should be equal to that of the iterator interface. You may declare similar

methods if you plan to have several distinct groups of iterators.

3. Implement concrete iterator classes for the collections that you want to be traversable with iterators. An iterator object must be linked with a single collection instance. Usually, this link is established via the iterator's constructor.

4. Implement the collection interface in your collection classes. The main idea is to provide the client with a shortcut for creating iterators, tailored for a particular collection class. The collection object must pass itself to the iterator's constructor to establish a link between them.

5. Go over the client code to replace all of the collection traversal code with the use of iterators. The client fetches a new iterator object each time it needs to iterate over the collection elements.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Abstract Factory and Iterator design patterns.

- For Abstract Factory Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Abstract-Factory
- For Iterator Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Iterator

Note: In case of physical labs this activity is optional but recommended.

Activity 2

Run the following examples of Abstract Factory and Iterator Patterns:

- Dialogue Controls Abstract Factory Example
    - https://github.com/sshpk/dp_fall20/tree/master/AbstractFactoryFA20/src/pk/cuiatd/dp/afactory
- Animals Factory Example
    - https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
- TV Channels Iteration
    - https://github.com/sshpk/dp_fall20/tree/master/IteratorFA20/src/pk/cuiatd/dp/it
- Names Array Iteration
    - https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm

- Restaurant Menu Iteration
    - https://www.javatpoint.com/iterator-pattern
- https://www.javacodegeeks.com/2015/09/iterator-design-pattern.html
- https://www.avajava.com/tutorials/lessons/iterator-pattern.html
- Profiles Iteration Example
    - https://refactoring.guru/design-patterns/iterator/java/example

Home Activities

For the first example in Activity 2, with following additions:

Home Work 1:

Add one more control support for the TextField in the given example.

Home Work 2:

Add another type of Factory that support one additional platform i.e. AndroidFactory.

Note that you will have to add the support for all its relevant UI Controls as well.

Home Work 3:

For the second example in Activity 2 related to Animals Factory, add another class named 'Cat' and implement it in the factory class and return in demo.

Home Work 4:

1. Find at least one another situation not covered in above examples, where Abstract pattern can be applied.
2. Find at least one another situation not covered in above examples, where Iterator pattern can be applied.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 06: Sessional 1 Exam

Purpose

The purpose of this lab is to conduct the first sessional exam based on the activities conducted so far.

Tasks

The tasks will be decided by the respective course instructor/lab tutor.

LAB 07: State and Mediator

Purpose

This lab will help the students to understand how an object alter its behavior when its internal state changes using the State pattern and how to reduce chaotic dependencies between objects using the Mediator pattern.

Outcomes

After completing this lab, students will be able to know that:

- **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.
- **Mediator** is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

Implementation Guidelines for State Pattern

1. Decide what class will act as the context. It could be an existing class which already has the state-dependent code; or a new class, if the state-specific code is distributed across multiple classes.
2. Declare the state interface. Although it may mirror all the methods declared in the context, aim only for those that may contain state-specific behavior.
3. For every actual state, create a class that derives from the state interface. Then go over the methods of the context and extract all code related to that state into your newly created class.

   While moving the code to the state class, you might discover that it depends on private members of the context. There are several workarounds:

   - Make these fields or methods public.
   - Turn the behavior you're extracting into a public method in the context and call it from the state class. This way is ugly but quick, and you can always fix it later.
   - Nest the state classes into the context class, but only if your programming language supports nesting classes.
4. In the context class, add a reference field of the state interface type and a public setter that allows overriding the value of that field.
5. Go over the method of the context again and replace empty state conditionals with

calls to corresponding methods of the state object.

6. To switch the state of the context, create an instance of one of the state classes and pass it to the context. You can do this within the context itself, or in various states, or in the client. Wherever this is done, the class becomes dependent on the concrete state class that it instantiates.

Implementation Guidelines for Mediator

1. Identify a group of tightly coupled classes which would benefit from being more independent (e.g., for easier maintenance or simpler reuse of these classes).

2. Declare the mediator interface and describe the desired communication protocol between mediators and various components. In most cases, a single method for receiving notifications from components is sufficient.

   This interface is crucial when you want to reuse component classes in different contexts. As long as the component works with its mediator via the generic interface, you can link the component with a different implementation of the mediator.

3. Implement the concrete mediator class. This class would benefit from storing references to all of the components it manages.

4. You can go even further and make the mediator responsible for the creation and destruction of component objects. After this, the mediator may resemble a factory or a facade.

5. Components should store a reference to the mediator object. The connection is usually established in the component's constructor, where a mediator object is passed as an argument.

6. Change the components' code so that they call the mediator's notification method instead of methods on other components. Extract the code that involves calling other components into the mediator class. Execute this code whenever the mediator receives notifications from that component.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Abstract Factory and Iterator design patterns.

- For State Pattern
  - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/State
- For Mediator Pattern
  - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Mediator

Note: In case of physical labs this activity is optional but recommended.

Activity 2 - Mediator Simple Implementation

Steps:

1. Create a Java application with a package name "Mediator"
2. Create the following files with respective code in each file.

Mediator.java

```
package Mediator;
public interface Mediator {
    public void operationD();
    public void operationA();
}
```

BaseComponent.java

```
package Mediator;
public class BaseComponent {
     protected Mediator mediator;
     public void setMediator(Mediator objMediator){
          this.mediator = objMediator;
     }
}
```

ConcreteColleaque2.java

```
package Mediator;
public class ConcreteColleaque2 extends BaseComponent{
    public void doOperationC(){
        System.out.println("Colleaque 2 performed operation C.");
    }
    public void doOperationD(){
        System.out.println("Colleaque 2 performed operation D.");
    }
}
```

ConcreteMediator.java

```
package Mediator;
public class ConcreteMediator  implements Mediator{
    private ConcreteColleaque1 objColleaque1;
    private ConcreteColleaque2 objColleaque2;

    // Mediator aggregates its concrete colleaques
    public ConcreteMediator(ConcreteColleaque1 component1,
ConcreteColleaque2 component2) {
        this.objColleaque1 = component1;
        component1.setMediator(this);
        this.objColleaque2 = component2;
        component2.setMediator(this);
    }

    @Override
```

```
    public void operationA() {
        System.out.println("Mediator knows that operationA can be
done by Colleaque1.");
        objColleaque1.doOperationA();
    }
    @Override
    public void operationD() {
        System.out.println("Mediator knows operationD can be done
by Colleaque2.");
        objColleaque2.doOperationD();
    }
}
```

Client.java

```
package Mediator;
public class Client {
   public static void main(String[] args) {
        // 1. Initiate concrete colleques
        ConcreteColleaque1 objColleaque1 = new
ConcreteColleaque1();
        ConcreteColleaque2 objColleaque2 = new
ConcreteColleaque2();
        // 2. Initiate mediator with controlling components
        Mediator objM =  new ConcreteMediator(objColleaque1,
objColleaque2);
        // 3. Ask Mediator to perform operations
        objM.operationA();
        objM.operationD();
   }
}
```

Activity 3

Run the following example of the state design pattern:

- https://github.com/sshpk/dp_fall20/tree/master/StateFA20/src/pk/cuiatd/dp/state

Home Activities

For the first example in Activity 2, Modify it with the following additions:

Home Work 1:

Add another ConcreteColleaque3 with Operation doOperationAll that internally ask mediator to perform OperationA and OperationD. Also mediator should be setup via its constructor.

Home Work 2:

Modify the concrete mediator such that concrete colleques will subscribe and unsubscribe to it than passing it colleques in constructor.

Home Work 3:

1. After watching the video demonstration in Activity 1, follow the guidelines and add one more state to the TV which indicate that it is OK or not working and also provide the respective behavior against the state.
2. Add one more attribute for volume with possible values Low, Medium and High. The respective behavior should work like on transition we can set volume from low to medium and not directly to High.

Home Work 4:

1. Find at least one another situation not covered in above examples, where State pattern can be applied.
2. Find at least one another situation not covered in above examples, where Mediator pattern can be applied.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

LAB 08: Bridge and Builder

Purpose

This lab will help the students to understand how split a large class or a set of closely related classes into two separate hierarchies (abstraction and implementation) using the Bridge pattern and how to construct complex objects step by step using the Builder pattern.

Outcomes

After completing this lab, students will be able to know that:

- **Bridge** is a structural design pattern that 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.
- **Builder** is a creational design pattern that 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.

Implementation Guidelines for Bridge Pattern

1. Identify the orthogonal dimensions in your classes. These independent concepts could be: abstraction/platform, domain/infrastructure, front-end/back-end, or interface/implementation.
2. See what operations the client needs and define them in the base abstraction class.
3. Determine the operations available on all platforms. Declare the ones that the abstraction needs in the general implementation interface.
4. For all platforms in your domain create concrete implementation classes, but make sure they all follow the implementation interface.
5. Inside the abstraction class, add a reference field for the implementation type. The abstraction delegates most of the work to the implementation object that's referenced in that field.
6. If you have several variants of high-level logic, create refined abstractions for each variant by extending the base abstraction class.
7. The client code should pass an implementation object to the abstraction's constructor to associate one with the other. After that, the client can forget about the implementation and work only with the abstraction object.

Implementation Guidelines for Builder

1. Make sure that you can clearly define the common construction steps for building all available product representations. Otherwise, you won't be able to proceed with implementing the pattern.
2. Declare these steps in the base builder interface.
3. Create a concrete builder class for each of the product representations and implement their construction steps.

   Don't forget about implementing a method for fetching the result of the construction. The reason why this method can't be declared inside the builder interface is that various builders may construct products that don't have a common interface. Therefore, you don't know what would be the return type for such a method. However, if you're dealing with products from a single hierarchy, the fetching method can be safely added to the base interface.
4. Think about creating a director class. It may encapsulate various ways to construct a product using the same builder object.
5. The client code creates both the builder and the director objects. Before construction starts, the client must pass a builder object to the director. Usually, the client does this only once, via parameters of the director's constructor. The director uses the builder object in all further construction. There's an alternative approach, where the builder is passed directly to the construction method of the director.
6. 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.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the _____ and _____ design patterns.

- For Bridge Pattern
  - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Bridge
- For Builder Pattern
  - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Builder

Note: In case of physical labs this activity is optional but recommended.

Activity 2 - Bridge Simple Implementation

Run the following examples of the Bridge design pattern:

- Accounts Java Example
    - https://en.wikipedia.org/wiki/Bridge_pattern
- Devices Example
    - https://refactoring.guru/design-patterns/bridge/java/example

Activity 3

Run the following examples of the Builder design pattern:

- Car Builder
    - https://refactoring.guru/design-patterns/builder/java/example
    - dp_fall21/BuilderFA21/src/pk/cuiatd/builder at master · sshpk/dp_fall21 · GitHub
- Bike Builder (C# to Java Conversion required)
    - https://en.wikipedia.org/wiki/Builder_pattern

Home Activities

Home Work 1:

For the first example in Activity 2, Implement the upper withdrawal limit of 50,000 on the account. Log an error message if the user tries to withdraw amount greater than the limit.

Home Work 2:

For the second example in Activity 2, Add a child protected mode support, so that if it is turn on then channels with odd number are not shown. Also modify the behavior accordingly for the Remote device of the TV.

Home Work 3:

For the Car Builder example in Activity 3, Add another car support for type VAN with SEMI_AUTOMATIC transmission.

Home Work 4:

1. Find at least one another situation not covered in above examples, where Bridge pattern can be applied.
2. Find at least one another situation not covered in above examples, where Builder pattern can be applied.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teacher.

Purpose

This lab will help the students to understand how to allow objects with incompatible interfaces to collaborate using the Adapter pattern, fit more objects into the available amount of RAM using the Flyweight pattern and provide a simplified interface to a library using Façade pattern.

Outcomes

After completing this lab, students will be able to know that:

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.
- **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.
- **Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Implementation Guidelines for Adapter Pattern

1. Make sure that you have at least two classes with incompatible interfaces:
    a. A useful service class, which you can't change (often 3rd-party, legacy or with lots of existing dependencies).
    b. One or several client classes that would benefit from using the service class.
2. Declare the client interface and describe how clients communicate with the service.
3. Create the adapter class and make it follow the client interface. Leave all the methods empty for now.
4. Add a field to the adapter class to store a reference to the service object. The common practice is to initialize this field via the constructor, but sometimes it's more convenient to pass it to the adapter when calling its methods.
5. One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.

6. Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.

Implementation Guidelines for Flyweight

1. Divide fields of a class that will become a flyweight into two parts:
    - the intrinsic state: the fields that contain unchanging data duplicated across many objects
    - the extrinsic state: the fields that contain contextual data unique to each object
2. Leave the fields that represent the intrinsic state in the class, but make sure they're immutable. They should take their initial values only inside the constructor.
3. Go over methods that use fields of the extrinsic state. For each field used in the method, introduce a new parameter and use it instead of the field.
4. Optionally, create a factory class to manage the pool of flyweights. It should check for an existing flyweight before creating a new one. Once the factory is in place, clients must only request flyweights through it. They should describe the desired flyweight by passing its intrinsic state to the factory.
5. The client must store or calculate values of the extrinsic state (context) to be able to call methods of flyweight objects. For the sake of convenience, the extrinsic state along with the flyweight-referencing field may be moved to a separate context class.

Implementation Guidelines for Facade

1. Check whether it's possible to provide a simpler interface than what an existing subsystem already provides. You're on the right track if this interface makes the client code independent from many of the subsystem's classes.
2. Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.
3. To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade. Now the client code is protected from any changes in the subsystem code. For example, when a subsystem gets upgraded to a new version, you will only need to modify the code in the facade.
4. If the facade becomes too big, consider extracting part of its behavior to a new, refined facade class.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Adapter, Flyweight and Facade design patterns.

- For Adapter Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Adapter
- For Flyweight Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Flyweight
- For Façade Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Facade

Note: In case of physical labs this activity is optional but recommended.

Activity 2 – Audio Player

We are going to create an audio player that should be able to play audios in mp3, mp4 and vlc formats. Run the following example of the Adapter design pattern:

- https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

Activity 3

Run the following examples of the Flyweight design pattern:

- https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm
- Forest Example
    - https://refactoring.guru/design-patterns/flyweight/java/example

Activity 4

Run the following examples of the Facade design pattern:

- Shapes API via Facade
    - https://www.tutorialspoint.com/design_pattern/facade_pattern.htm
- Media Player Example
    - https://refactoring.guru/design-patterns/facade/java/example

Home Activities

Home Work 1:

For the first example in Activity 2:

1. Add a support of one another format in the default AudioPlayer
2. Add two more classes each one supports another format of the audio that implements AdvancedMediaPlayer and then utilize them from the main method.
3. Simplify the AdvancedMediaPlayer interface such that dummy methods are not required to be implemented in the concrete player class. For example, PlayVlc method in Mp4Player class is with a dummy definition and not in use.

Home Work 2:

For the first example in Activity 3:

1. Add RectangleShape and PolygonShape classes like the CircleShape and then use them from your main method.
2. Add LineThinkness and Position Intrinsic stats to new classes.
3. Randomize the radius extrinsic state for all concrete shape classes

Home Work 3:

1. Carefully watch the demonstration of the Façade pattern and then combine it with Decorator pattern as per the guidelines in the video on the following URL.
    - https://youtu.be/-sAFcIahKSk

Home Work 4:

1. Find at least one another situation not covered in above examples, where Adapter pattern can be applied.
2. Find at least one another situation not covered in above examples, where Flyweight pattern can be applied.
3. Find at least one another situation not covered in above examples, where Facade pattern can be applied.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teach

LAB 10: Template Method, Proxy and Composite

Purpose

This lab will help the students:

- To defines the skeleton of an algorithm in the superclass but let's subclasses override specific steps of the algorithm using Template Method pattern.
- To provide a substitute or placeholder for another object using Proxy pattern and
- Compose objects into tree structures and then work with these structures as if they were individual objects using Composite pattern.

Outcomes

After completing this lab, students will be able to know that:

- **Template Method** is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but let's subclasses override specific steps of the algorithm without changing its structure.
- **Proxy** is a structural design pattern that lets you provide a substitute or placeholder for another object. 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.
- **Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Implementation Guidelines for Template Method Pattern

1. Analyze the target algorithm to see whether you can break it into steps. Consider which steps are common to all subclasses and which ones will always be unique.
2. Create the abstract base class and declare the template method and a set of abstract methods representing the algorithm's steps. Outline the algorithm's structure in the template method by executing corresponding steps. Consider making the template method final to prevent subclasses from overriding it.
3. It's okay if all the steps end up being abstract. However, some steps might benefit from having a default implementation. Subclasses don't have to implement those methods.
4. Think of adding hooks between the crucial steps of the algorithm.
5. For each variation of the algorithm, create a new concrete subclass. It must implement all of the abstract steps, but may also override some of the optional ones.

Implementation Guidelines for Proxy Pattern

1. If there's no pre-existing service interface, create one to make proxy and service objects interchangeable. Extracting the interface from the service class isn't always possible, because you'd need to change all of the service's clients to use that interface. Plan B is to make the proxy a subclass of the service class, and this way it'll inherit the interface of the service.

2. Create the proxy class. It should have a field for storing a reference to the service. Usually, proxies create and manage the whole life cycle of their services. On rare occasions, a service is passed to the proxy via a constructor by the client.

3. Implement the proxy methods according to their purposes. In most cases, after doing some work, the proxy should delegate the work to the service object.

4. Consider introducing a creation method that decides whether the client gets a proxy or a real service. This can be a simple static method in the proxy class or a full-blown factory method.

5. Consider implementing lazy initialization for the service object.

Implementation Guidelines for Composite Pattern

1. Make sure that the core model of your app can be represented as a tree structure. Try to break it down into simple elements and containers. Remember that containers must be able to contain both simple elements and other containers.

2. Declare the component interface with a list of methods that make sense for both simple and complex components.

3. Create a leaf class to represent simple elements. A program may have multiple different leaf classes.

4. Create a container class to represent complex elements. In this class, provide an array field for storing references to sub-elements. The array must be able to store both leaves and containers, so make sure it's declared with the component interface type.

   While implementing the methods of the component interface, remember that a container is supposed to be delegating most of the work to sub-elements.

5. 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. This would violate the Interface Segregation Principle because the methods will be empty in the leaf class. However, the client will be able to treat all the elements equally, even when composing the tree.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Template Method, Proxy and Composite design patterns.

- For Template Method Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Template-Method
- For Proxy Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Proxy
- For Composite Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Composite

Note: In case of physical labs this activity is optional but recommended.

Activity 2

After watching the video in Activity 1 for Template Method, Run the following examples of the Template Method design pattern:

- https://www.tutorialspoint.com/design_pattern/template_pattern.htm
- https://refactoring.guru/design-patterns/template-method/java/example

Activity 3

Run the following example of the Proxy design pattern:

- https://www.tutorialspoint.com/design_pattern/proxy_pattern.htm
- https://refactoring.guru/design-patterns/proxy/java/example

Activity 4

Run the following example of the Composite design pattern:

- https://www.tutorialspoint.com/design_pattern/composite_pattern.htm
- https://refactoring.guru/design-patterns/composite/java/example

Home Activities

Home Work 1:

For the first example in Activity 2, Modify it with following additions:

1. Add one more concrete step to the template method "breakForRefreshment" after startPlay method and update the template method skeleton accordingly.

2. Add one variable step to the template method "performShiftNow" which is activated after the startPlay method and then recall startPlay after performShiftNow. For cricket game "The batting team started fielding now" is printed while for football game it prints "The pole are shifted between the teams".
3. Update changes in the Demo class if necessary.

Home Work 2:

For the example in Activity 3, Modify it with following additions:

1. Add a code attribute to proxy and real image classes and set its default via its constructor. Display the real image only if display code is matching to the real image code.
2. Add some additional details (e.g. file size) while displaying the image in the Proxy
3. Parameterize the code in the display method to display fake on bad code or real image on code matching.

Home Work 3:

1. Add one more primitive component Triangle to this example.
2. Combine the Triangle with the Circle and draw that composite component on the top-right side of the canvas.
3. (Optional) Look for the use of Composite pattern in your final year project and implement that part in Java.

Home Work 4:

1. Find at least one another situation not covered in above examples, where Template Method pattern can be applied.
2. Find an example of using the Proxy pattern in your semester project, or any other example of your own.
3. Find at least one another situation not covered in above examples, where Composite pattern can be applied.

Assignment Deliverables

- Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teach

LAB 11: Prototype, Command and Interpreter

Purpose

This lab will help the students to understand how to copy existing objects without making your code dependent on their classes using Prototype, turns a request into a stand-alone object that contains all information about the request using Command pattern and define a representation for the grammar of a language along with an interpreter using Interpreter design pattern.

Outcomes

After completing this lab, students will be able to know that:

- **Prototype** is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
- **Command** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue a request's execution, and support undoable operations.
- **Interpreter** define a representation for the grammar of a language along with an interpreter that uses the representation to interpret sentences in the language.

Implementation Guidelines for Prototype Pattern

1. Create the prototype interface and declare the clone method in it. Or just add the method to all classes of an existing class hierarchy, if you have one.
2. A prototype class must define the alternative constructor that accepts an object of that class as an argument. The constructor must copy the values of all fields defined in the class from the passed object into the newly created instance. If you're changing a subclass, you must call the parent constructor to let the superclass handle the cloning of its private fields.

   If your programming language doesn't support method overloading, you may define a special method for copying the object data. The constructor is a more convenient place to do this because it delivers the resulting object right after you call the new operator.
3. The cloning method usually consists of just one line: running a new operator with the prototypical version of the constructor. Note, that every class must explicitly override the cloning method and use its own class name along with the new operator. Otherwise, the cloning method may produce an object of a parent class.
4. Optionally, create a centralized prototype registry to store a catalog of frequently used

prototypes.

You can implement the registry as a new factory class or put it in the base prototype class with a static method for fetching the prototype. This method should search for a prototype based on search criteria that the client code passes to the method. The criteria might either be a simple string tag or a complex set of search parameters. After the appropriate prototype is found, the registry should clone it and return the copy to the client.

Finally, replace the direct calls to the subclasses' constructors with calls to the factory method of the prototype registry.

Implementation Guidelines for Command

1. Declare the command interface with a single execution method.
2. Start extracting requests into concrete command classes that implement the command interface. Each class must have a set of fields for storing the request arguments along with a reference to the actual receiver object. All these values must be initialized via the command's constructor.
3. Identify classes that will act as senders. Add the fields for storing commands into these classes. Senders should communicate with their commands only via the command interface. Senders usually don't create command objects on their own, but rather get them from the client code.
4. Change the senders so they execute the command instead of sending a request to the receiver directly.
5. The client should initialize objects in the following order:
    - Create receivers.
    - Create commands, and associate them with receivers if needed.
    - Create senders, and associate them with specific commands.

Implementation Guidelines for Interpreter

1. Creating the abstract syntax tree. The abstract syntax tree can be created by a table-driven parser, by a hand-crafted (usually recursive descent) parser, or directly by the client.
2. Defining the Interpret operation. You don't have to define the Interpret operation in the expression classes.
3. Sharing terminal symbols with the Flyweight pattern. Grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Prototype, Command and Interpreter design patterns.

- For Prototype Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Prototype
- For Command Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Command
- For Interpreter Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Interpreter

Note: In case of physical labs this activity is optional but recommended.

Activity 2

Run the following example of the Prototype design pattern:

- https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm
- https://refactoring.guru/design-patterns/prototype/java/example

Activity 3

Run the following example of the Command design pattern:

- https://www.tutorialspoint.com/design_pattern/command_pattern.htm
- https://refactoring.guru/design-patterns/command/java/example

Activity 4

Run the following example of the Interpreter design pattern:

- https://www.tutorialspoint.com/design_pattern/interpreter_pattern.htm

Home Activities

Home Work 1:

For the first example in Activity 2, Modify it with the following:

1. Eliminate the parameterized constructor and implement Cloneable interface to create object clone and then copy values in the clone method.
2. Modify Example 3 to add caching support for cloneable objects.

Home Work 2:

For example, 1

1. Call and debug the WriteFileCommand in main method.
2. Add a logic via a bool variable such that if the execute is called the Boolean variable is set to true and the undo action set it to false. On exiting the application if the value is true then exit after confirmation.

For example, 2

1. Implement the replace command such that a selected text gets replaced with a value already in the clipboard with label as Ctr+H.

Home Work 3:

1. Find at least one another situation not covered in above examples, where Prototype pattern can be applied.
2. Find at least one another situation not covered in above examples, where Command pattern can be applied.
3. Find at least one another situation not covered in above examples, where Interpreter pattern can be applied.

Assignment Deliverables

- Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teach

LAB 12: Sessional 2 Exam

Purpose

The purpose of this lab is to conduct the second sessional exam based on the activities conducted so far.

Tasks

The tasks will be decided by the respective course instructor/lab tutor.

LAB 13: Visitor, Memento and Null Patterns

Purpose

This lab will help the students to understand how to separate algorithms from the objects on which they operate using Visitor pattern, how to save and restore the previous state of an object without revealing the details of its implementation using Memento pattern and how to gracefully handle the mighty Null pointer exceptions.

Outcomes

After completing this lab, students will be able to know that:

- **Visitor** is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.
- **Memento** is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.
- **Null object** design pattern describes the uses of null objects and their behavior.

Implementation Guidelines for Visitor Pattern

1. Declare the visitor interface with a set of "visiting" methods, one per each concrete element class that exists in the program.
2. Declare the element interface. If you're working with an existing element class hierarchy, add the abstract "acceptance" method to the base class of the hierarchy. This method should accept a visitor object as an argument.
3. Implement the acceptance methods in all concrete element classes. These methods must simply redirect the call to a visiting method on the incoming visitor object which matches the class of the current element.
4. The element classes should only work with visitors via the visitor interface. Visitors, however, must be aware of all concrete element classes, referenced as parameter types of the visiting methods.
5. For each behavior that can't be implemented inside the element hierarchy, create a new concrete visitor class and implement all of the visiting methods.

   You might encounter a situation where the visitor will need access to some private members of the element class. In this case, you can either make these fields or methods public, violating the element's encapsulation, or nest the visitor class in the element class. The latter is only possible if you're lucky to work with a programming language that supports nested classes.

6. The client must create visitor objects and pass them into elements via “acceptance” methods.

Implementation Guidelines for Memento Pattern

1. Determine what class will play the role of the originator. It’s important to know whether the program uses one central object of this type or multiple smaller ones.

2. Create the memento class. One by one, declare a set of fields that mirror the fields declared inside the originator class.

3. Make the memento class immutable. A memento should accept the data just once, via the constructor. The class should have no setters.

4. If your programming language supports nested classes, nest the memento inside the originator. If not, extract a blank interface from the memento class and make all other objects use it to refer to the memento. You may add some metadata operations to the interface, but nothing that exposes the originator’s state.

5. Add a method for producing mementos to the originator class. The originator should pass its state to the memento via one or multiple arguments of the memento’s constructor.

   The return type of the method should be of the interface you extracted in the previous step (assuming that you extracted it at all). Under the hood, the memento-producing method should work directly with the memento class.

6. Add a method for restoring the originator’s state to its class. It should accept a memento object as an argument. If you extracted an interface in the previous step, make it the type of the parameter. In this case, you need to typecast the incoming object to the memento class, since the originator needs full access to that object.

7. The caretaker, whether it represents a command object, a history, or something entirely different, should know when to request new mementos from the originator, how to store them and when to restore the originator with a particular memento.

8. The link between caretakers and originators may be moved into the memento class. In this case, each memento must be connected to the originator that had created it. The restoration method would also move to the memento class. However, this would all make sense only if the memento class is nested into originator or the originator class provides sufficient setters for overriding its state.

Implementation Guidelines for Null Object Pattern

1. This pattern should be used carefully as it can make errors/bugs appear as normal program execution.

2. Care should be taken not to implement this pattern just to avoid null checks and make code more readable, since the harder-to-read code may just move to another place and be less standard

3. The common pattern in most languages with reference types is to compare a reference to a single value referred to as null or nil.

4. There is additional need for testing that no code anywhere ever assigns null instead of the null object, because in most cases and languages with static typing, this is not a compiler error if the null object is of a reference type, although it would certainly lead to errors at run time in parts of the code where the pattern was used to avoid null checks
5. Checking for the null object instead of for the null or nil value introduces overhead, as does the singleton pattern likely itself upon obtaining the singleton reference.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Visitor, Memento and Null Object design patterns.

- For Visitor Pattern
  - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Visitor
- For Memento Pattern
  - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Memento
- For Null Object Pattern
  - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Null-Object-Pattern

Note: In case of physical labs this activity is optional but recommended.

Activity 2

Run the following examples of the Visitor design pattern:

- Computer Parts Example
  - https://www.tutorialspoint.com/design_pattern/visitor_pattern.htm
- Shapes Exporter
  - https://refactoring.guru/design-patterns/visitor/java/example

Activity 3

Run the following example of the Memento design pattern:

- https://www.tutorialspoint.com/design_pattern/memento_pattern.htm
- https://refactoring.guru/design-patterns/memento/java/example

Activity 4

Run the following example of the Null Object design pattern:

- https://www.tutorialspoint.com/design_pattern/null_object_pattern.htm

Home Activities

Home Work 1:

1. For the first example in Activity 2, Add support for one more device "External Hard Disk", When you display the parts the new parts should also be displayed in the results.
2. For the second example in Activity 2, Export the Shapes in JSON format as well.

Home Work 2:

1. For the second example in Activity 3, Add one another primitive shape (Line or Oval) to the Shapes and then utilize/draw as primitive and compound shape in the canvas of the editor.
2. Implement the respective command history on this new shape as well like others.

Home Work 3:

1. Justify your outcome when you want to get the customer name against the null value of the name.
2. Is there any possibility to generate NullPointerException with the code example from the Demo class while reading customers by name?

Home Work 4:

1. Find at least one another situation not covered in above examples, where Visitor pattern can be applied.
2. Find at least one another situation not covered in above examples, where Memento pattern can be applied.
3. Find at least one scenario where Null Object design pattern is mandatory.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teach

Purpose

This lab will help the students to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations, separate the concerns of the application in Model, View and Controller aspects.

Outcomes

After completing this lab, students will be able to know that:

- **Filter pattern** or Criteria pattern is a design pattern that enables developers to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations.
- In **Model View Controller (MVC)** pattern is used to separate application's concerns.
    - **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
    - **View** - View represents the visualization of the data that model contains.
    - **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Implementation Guidelines for Filter Pattern

This type of design pattern comes under structural pattern as this pattern combines multiple criteria to obtain single criteria.

Implementation Guidelines for MVC Pattern

In addition to dividing the application into these components, the model–view–controller design defines the interactions between them.

1. The model is responsible for managing the data of the application. It receives user input from the controller.
2. The view renders presentation of the model in a particular format.
6. The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

Activity 1 (Remote teaching)

Login in to subexpert.com with your student account and then watch the following interactive videos that demonstrates the Filter and Model View Controller (MVC) design patterns.

- For Filter Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Filter-Design-Pattern
- For MVC Pattern
    - https://www.subexpert.com/CourseLectures/OnTopic/Design-Patterns/Model-View-Controller-Pattern

Note: In case of physical labs this activity is optional but recommended.

Activity 2

Run the following example of the Filter design pattern:

- https://www.tutorialspoint.com/design_pattern/filter_pattern.htm

Activity 3

Run the following example of the MVC design pattern:

- https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm

Home Activities

Home Work 1:

For the first example in Activity 2, Modify it with the following additions:

1. Add DOB attribute for the Person and then create a CriteriaDOB such that Persons within a range of dates are filtered.
2. Apply two criteria collectively on the person's collection.

Home Work 2:

For the first example in Activity 3, Modify it with the following additions:

1. Convert the Model part of the example into N-Layer architecture, such that data is retrieved from another DAL Layer.

Home Work 3:

4. Find at least one another situation not covered in above examples, where Filter pattern can be applied.
5. Find at least one another situation not covered in above examples, where MVC pattern can be applied.

Assignment Deliverables

Create a one-minute video demonstration of your home activities, upload it to a cloud storage and then share the link with your class teach

LAB 15 Presentation on Latest Design Patterns

Purpose

The purpose of this lab is to let the student find at least one latest design pattern and then create a video demonstration for theoretical and implementation perspective.

Outcomes

After completing this lab, students will be able to know that:

- **Presentation** on one latest design pattern selected from the list in the next section.
- Three coding examples that spans in a simple, medium and advance usage of the pattern
- A screen recorded video demonstration of the pattern and its coding examples in less than five minutes.

List of Patterns

Choose a pattern from the following list and ensure that no other student in your class has selected that pattern.

Creational patterns

1. **Dependency Injection** A class accepts the objects it requires from an injector instead of creating the objects directly
2. **Lazy initialization** Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.
3. **Multiton** Ensure a class has only named instances, and provide a global point of access to them.
4. **Object pool** Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.
5. **Resource acquisition is initialization (RAII)** Ensure that resources are properly released by tying them to the lifespan of suitable objects.

Structural Patterns

6. **Extension object**: Adding functionality to a hierarchy without changing the hierarchy.
7. **Front controller**: The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.

8. **Marker**: Empty interface to associate metadata with a class.
9. **Module**: Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity
10. **Twin**: Twin allows modeling of multiple inheritance in programming languages that do not support this feature.

Behavioral patterns

11. **Blackboard**: Artificial intelligence pattern for combining disparate sources of data (see blackboard system)
12. **Null object**: Avoid null references by providing a default object.
13. **Servant**: Define common functionality for a group of classes. The servant pattern is also frequently called helper class or utility class implementation for a given set of classes. The helper classes generally have no objects hence they have all static methods that act upon different kinds of class objects.
14. **Specification**: Re-combinable business logic in a Boolean fashion.

Concurrency patterns

15. **Active Object** Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
16. **Balking** Only execute an action on an object when the object is in a particular state.
17. **Binding properties** Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way
18. **Compute kernel** The same calculation many times in parallel, differing by integer parameters used with non-branching pointer math into shared arrays, such as GPU-optimized Matrix multiplication or Convolutional neural network.
19. **Double-checked locking** Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed.
    Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.
20. **Event-based asynchronous** Addresses problems with the asynchronous pattern that occur in multithreaded programs.
21. **Guarded suspension** Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
22. **Join** Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing. Compared to the use of threads and locks, this is a high-level programming model.
23. **Lock** One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.[
24. **Messaging design pattern (MDP)** Allows the interchange of information (i.e. messages) between components and applications.
25. **Monitor object** An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
26. **Reactor** A reactor object provides an asynchronous interface to resources that must be handled synchronously.

27. **Read-write lock** Allows concurrent read access to an object, but requires exclusive access for write operations. An underlying semaphore might be used for writing, and a Copy-on-write mechanism may or may not be used.
28. **Scheduler** Explicitly control when threads may execute single-threaded code.
29. **Thread pool** A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.
30. **Thread-specific storage** Static or "global" memory local to a thread.
31. **Safe Concurrency with Exclusive Ownership** Avoiding the need for runtime concurrent mechanisms, because exclusive ownership can be proven. This is a notable capability of the Rust language, but compile-time checking isn't the only means, a programmer will often manually design such patterns into code - omitting the use of locking mechanism because the programmer assesses that a given variable is never going to be concurrently accessed.
32. **CPU atomic operation**: x86 and other CPU architectures support a range of atomic instructions that guarantee memory safety for modifying and accessing primitive values (integers). For example, two threads may both increment a counter safely. These capabilities can also be used to implement the mechanisms for other concurrency patterns as above. The C# language uses the Interlocked class for these capabilities.

Activity 1 (Remote teaching)

Select and prepare a presentation on one latest design pattern. Explain its theoretical aspects along with the problem contexts where this pattern can be applied.

Activity 2

For each problem context, in the first activity, provide a coding example.

Activity 3

Prepare a screen recording with your own voice and demonstrate the pattern in as minimum time as possible.

Home Activities

Home Work 1:

Provide a review against the two design pattern videos and suggest improvements where necessary in the video demonstration.

Home Work 2:

Make sure to incorporate the suggestions against your video and then share the latest version with the class.

Assignment Deliverables

One demonstration video against your design pattern and then its updated version along with addressed improvements suggested by your class fellows.

LAB 16: Final Exam

Purpose

The purpose of this lab is to conduct the final exam based on the activities conducted throughout the semester.

Tasks

The tasks will be decided by the respective course instructor/lab tutor.