EMERGING PROGRAMMING PARADIGM
(CEC12)
(LAB FILE)

Name: Shubham

Submitted To: **Roll No.:** 2017UCO1649

Kanika Ma'am **Branch:** COE-3

TABLE OF CONTENTS

1. Implement Facade design pattern and draw the corresponding class diagram.

Facade Design Pattern

Github Link :
https://github.com/porcelainruler/EPP_Lab/tree/master/Facade_Design_Pattern

layout	title	folder	permalink	categories
pattern	Facade_Design_Pattern	EPP_Lab	EPP_Lab/Facade_Design_Pattern/	Structural Design Pattern

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Explanation

Real world example

How does a goldmine work? "Well, the miners go down there and dig gold!" you say. That is what you believe because you are using a simple interface that goldmine provides on the outside, internally it has to do a lot of stuff to make it happen. This simple interface to the complex subsystem is a facade.

In plain words

Facade pattern provides a simplified interface to a complex subsystem.

Wikipedia says

A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

Programmatic Example

Taking our goldmine example from above. Here we have the dwarven mine worker hierarchy

Code :

```
public abstract class DwarvenMineWorker {

  private static final Logger LOGGER =
LoggerFactory.getLogger(DwarvenMineWorker.class);

  public void goToSleep() {
    LOGGER.info("{} goes to sleep.", name());
  }

  public void wakeUp() {
    LOGGER.info("{} wakes up.", name());
  }

  public void goHome() {
    LOGGER.info("{} goes home.", name());
  }

  public void goToMine() {
    LOGGER.info("{} goes to the mine.", name());
  }

  private void action(Action action) {
    switch (action) {
      case GO_TO_SLEEP:
        goToSleep();
        break;
      case WAKE_UP:
        wakeUp();
        break;
      case GO_HOME:
        goHome();
        break;
      case GO_TO_MINE:
        goToMine();
        break;
      case WORK:
        work();
        break;
      default:
        LOGGER.info("Undefined action");
        break;
    }
  }

  public void action(Action... actions) {
    Arrays.stream(actions).forEach(this::action);
  }

  public abstract void work();

  public abstract String name();

  static enum Action {
    GO_TO_SLEEP, WAKE_UP, GO_HOME, GO_TO_MINE, WORK
  }
}

public class DwarvenTunnelDigger extends DwarvenMineWorker {

  private static final Logger LOGGER =
LoggerFactory.getLogger(DwarvenTunnelDigger.class);

  @Override
  public void work() {
    LOGGER.info("{} creates another promising tunnel.", name());
  }

  @Override
  public String name() {
    return "Dwarven tunnel digger";
  }
}

public class DwarvenGoldDigger extends DwarvenMineWorker {

  private static final Logger LOGGER =
LoggerFactory.getLogger(DwarvenGoldDigger.class);
```

```
  @Override
  public void work() {
    LOGGER.info("{} digs for gold.", name());
  }

  @Override
  public String name() {
    return "Dwarf gold digger";
  }
}

public class DwarvenCartOperator extends DwarvenMineWorker {

  private static final Logger LOGGER =
LoggerFactory.getLogger(DwarvenCartOperator.class);

  @Override
  public void work() {
    LOGGER.info("{} moves gold chunks out of the mine.", name());
  }

  @Override
  public String name() {
    return "Dwarf cart operator";
  }
}
```

<u>To operate all these goldmine workers we have the facade:</u>

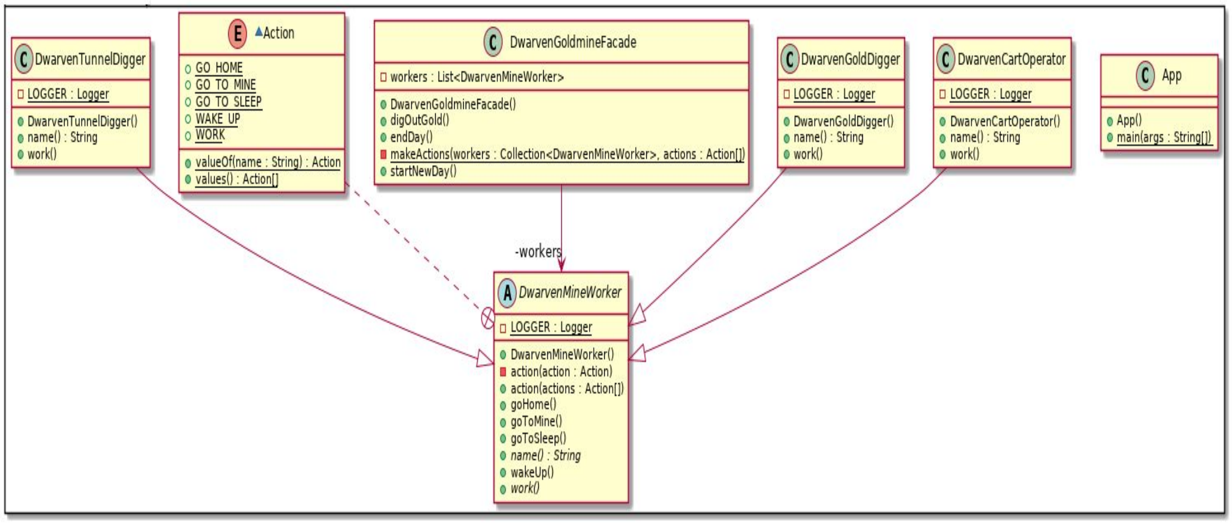
Code :

```
public class DwarvenGoldmineFacade {

  private final List<DwarvenMineWorker> workers;

  public DwarvenGoldmineFacade() {
      workers = List.of(
            new DwarvenGoldDigger(),
            new DwarvenCartOperator(),
            new DwarvenTunnelDigger());
  }

  public void startNewDay() {
    makeActions(workers, DwarvenMineWorker.Action.WAKE_UP,
DwarvenMineWorker.Action.GO_TO_MINE);
  }
  public void digOutGold() {
    makeActions(workers, DwarvenMineWorker.Action.WORK);
  }

  public void endDay() {
    makeActions(workers, DwarvenMineWorker.Action.GO_HOME,
DwarvenMineWorker.Action.GO_TO_SLEEP);
  }

  private static void makeActions(Collection<DwarvenMineWorker> workers,
      DwarvenMineWorker.Action... actions) {
    workers.forEach(worker -> worker.action(actions));
  }
}
```

Now to use the facade:

```
DwarvenGoldmineFacade facade = new DwarvenGoldmineFacade();
facade.startNewDay();
// Dwarf gold digger wakes up.
// Dwarf gold digger goes to the mine.
// Dwarf cart operator wakes up.
// Dwarf cart operator goes to the mine.
// Dwarven tunnel digger wakes up.
// Dwarven tunnel digger goes to the mine.
facade.digOutGold();
// Dwarf gold digger digs for gold.
// Dwarf cart operator moves gold chunks out of the mine.
// Dwarven tunnel digger creates another promising tunnel.
facade.endDay();
// Dwarf gold digger goes home.
// Dwarf gold digger goes to sleep.
// Dwarf cart operator goes home.
// Dwarf cart operator goes to sleep.
// Dwarven tunnel digger goes home.
// Dwarven tunnel digger goes to sleep.
```

Class diagram



Applicability

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

2. Implement Adaptor design pattern and draw the corresponding class diagram.

Adaptor Design Pattern

Github Link : https://github.com/porcelainruler/EPP_Lab/tree/master/Adapter_Design_Pattern

layout	title	folder	permalink	categories
pattern	Adaptor_Design_Pattern	EPP_Lab	EPP_Lab/Adaptor_Design_Pattern/	Structural Design Pattern

Also known as

Wrapper

Intent

Convert the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Implementation

Real world example

Consider that you have some pictures in your memory card and you need to transfer them to your computer. In order to transfer them you need some kind of adapter that is compatible with your computer ports so that you can attach a memory card to your computer. In this case the card reader is an adapter. Another example would be the famous power adapter; a three legged plug can't be connected to a two pronged outlet, it needs to use a power adapter that makes it compatible with the two pronged outlet. Yet another example would be a translator translating words spoken by one person to another

Programmatic Example

Consider a captain that can only use rowing boats and cannot sail at all.

First we have interfaces `RowingBoat` and `FishingBoat`

```
public interface RowingBoat {
  void row();
}

public class FishingBoat {
  private static final Logger LOGGER =
LoggerFactory.getLogger(FishingBoat.class);
  public void sail() {
    LOGGER.info("The fishing boat is sailing");
  }
}
```

And captain expects an implementation of `RowingBoat` interface to be able to move

```
public class Captain {

  private RowingBoat rowingBoat;
  // default constructor and setter for rowingBoat
  public Captain(RowingBoat rowingBoat) {
    this.rowingBoat = rowingBoat;
  }

  public void row() {
    rowingBoat.row();
  }
}
```

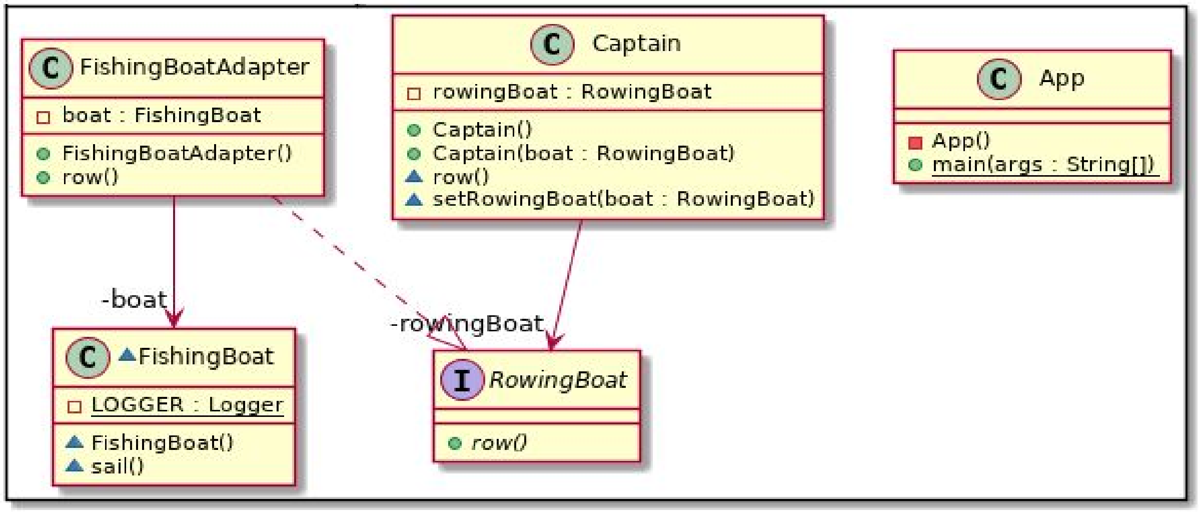
Now let's say the pirates are coming and our captain needs to escape but there is only a fishing boat available. We need to create an adapter that allows the captain to operate the fishing boat with his rowing boat skills.

```
public class FishingBoatAdapter implements RowingBoat {
  private static final Logger LOGGER =
LoggerFactory.getLogger(FishingBoatAdapter.class);

  private FishingBoat boat;

  public FishingBoatAdapter() {
    boat = new FishingBoat();
  }

  @Override
  public void row() {
    boat.sail();
  }
}
```

And now the `Captain` can use the `FishingBoat` to escape the pirates.

```
var captain = new Captain(new FishingBoatAdapter());
captain.row();
```

Class diagram



3. Implement Bridge design pattern and draw the corresponding class diagram.

Bridge Design Pattern

Github Link :
https://github.com/porcelainruler/EPP_Lab/tree/master/Bridge_Design_Pattern

layout	title	folder	permalink	categories
pattern	Bridge_Design_Pattern	EPP_Lab	EPP_Lab/Bridgee_Design_Pattern/	Structural Design Pattern

Also known as

Handle/Body

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Implementation

Real world example

Consider you have a weapon with different enchantments and you are supposed to allow mixing different weapons with different enchantments. What would you do? Create multiple copies of each of the weapons for each of the enchantments or would you just create separate enchantment and set it for the weapon as needed? Bridge pattern allows you to do the second.

Programmatic Example

Translating our weapon example from above. Here we have the `Weapon` hierarchy

```
public interface Weapon {
  void wield();
  void swing();
  void unwield();
  Enchantment getEnchantment();
}

public class Sword implements Weapon {

  private final Enchantment enchantment;

  public Sword(Enchantment enchantment) {
    this.enchantment = enchantment;
  }

  @Override
  public void wield() {
    LOGGER.info("The sword is wielded.");
    enchantment.onActivate();
  }

  @Override
  public void swing() {
    LOGGER.info("The sword is swinged.");
    enchantment.apply();
  }

  @Override
  public void unwield() {
    LOGGER.info("The sword is unwielded.");
    enchantment.onDeactivate();
  }

  @Override
  public Enchantment getEnchantment() {
    return enchantment;
  }
}

public class Hammer implements Weapon {

  private final Enchantment enchantment;

  public Hammer(Enchantment enchantment) {
    this.enchantment = enchantment;
  }

  @Override
  public void wield() {
    LOGGER.info("The hammer is wielded.");
    enchantment.onActivate();
  }

  @Override
  public void swing() {
    LOGGER.info("The hammer is swinged.");
    enchantment.apply();
  }

  @Override
  public void unwield() {
    LOGGER.info("The hammer is unwielded.");
    enchantment.onDeactivate();
  }

  @Override
  public Enchantment getEnchantment() {
    return enchantment;
  }
}
```

And the separate enchantment hierarchy

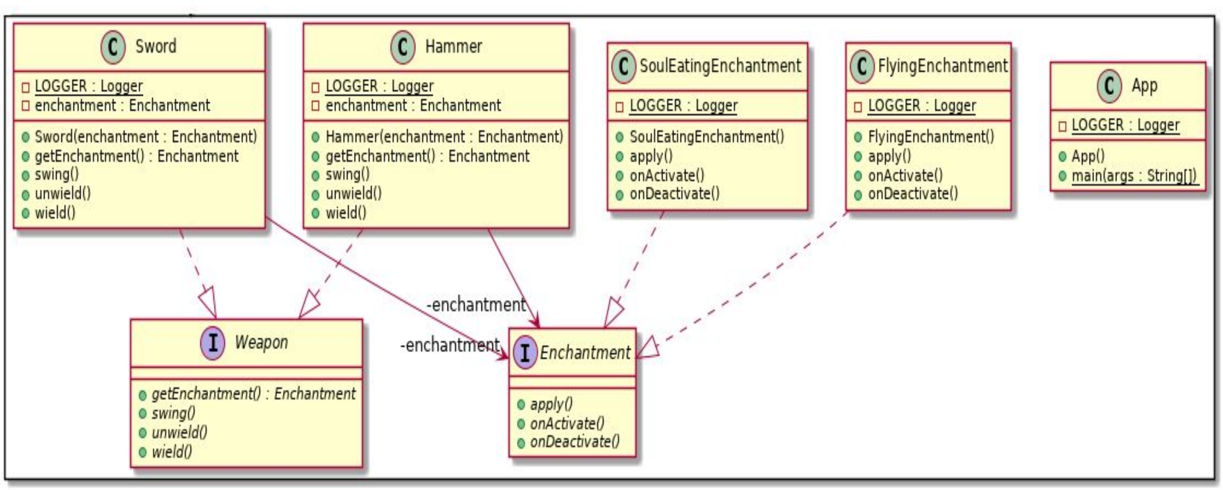
```
public interface Enchantment {
  void onActivate();
  void apply();
  void onDeactivate();
}

public class FlyingEnchantment implements Enchantment {
  @Override
  public void onActivate() {
    LOGGER.info("The item begins to glow faintly.");
  }

  @Override
  public void apply() {
    LOGGER.info("The item flies and strikes the enemies finally returning to
owner's hand.");
  }

  @Override
  public void onDeactivate() {
    LOGGER.info("The item's glow fades.");
  }
}

public class SoulEatingEnchantment implements Enchantment {

  @Override
  public void onActivate() {
    LOGGER.info("The item spreads bloodlust.");
  }

  @Override
  public void apply() {
    LOGGER.info("The item eats the soul of enemies.");
  }

  @Override
  public void onDeactivate() {
    LOGGER.info("Bloodlust slowly disappears.");
  }
}
```

And both the hierarchies in action

```
var enchantedSword = new Sword(new SoulEatingEnchantment());
enchantedSword.wield();
enchantedSword.swing();
enchantedSword.unwield();
// The sword is wielded.
// The item spreads bloodlust.
// The sword is swinged.
// The item eats the soul of enemies.
// The sword is unwielded.
// Bloodlust slowly disappears.
```

```
var hammer = new Hammer(new FlyingEnchantment());
hammer.wield();
hammer.swing();
hammer.unwield();
// The hammer is wielded.
// The item begins to glow faintly.
// The hammer is swinged.
// The item flies and strikes the enemies finally returning to owner's hand.
// The hammer is unwielded.
// The item's glow fades.
```

Class diagram



4. Implement the Decorator design pattern and draw the corresponding class diagram.

Decorator Design Pattern

Github Link : https://github.com/porcelainruler/EPP_Lab/tree/master/Decorator_Design_Pattern

layout	title	folder	permalink	categories
pattern	Decorator_Design_Pattern	EPP_Lab	EPP_Lab/Decoratorr_Design_Pattern/	Structural Design Pattern

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Implementation

Real world example

There is an angry troll living in the nearby hills. Usually it goes bare handed but sometimes it has a weapon. To arm the troll it's not necessary to create a new troll but to decorate it dynamically with a suitable weapon.

Programmatic Example

Let's take the troll example. First of all we have a simple troll implementing the troll interface

```
public interface Troll {
  void attack();
  int getAttackPower();
  void fleeBattle();
}

public class SimpleTroll implements Troll {

  private static final Logger LOGGER = LoggerFactory.getLogger(SimpleTroll.class);

  @Override
  public void attack() {
    LOGGER.info("The troll tries to grab you!");
  }

  @Override
  public int getAttackPower() {
    return 10;
  }

  @Override
  public void fleeBattle() {
    LOGGER.info("The troll shrieks in horror and runs away!");
  }
}
```

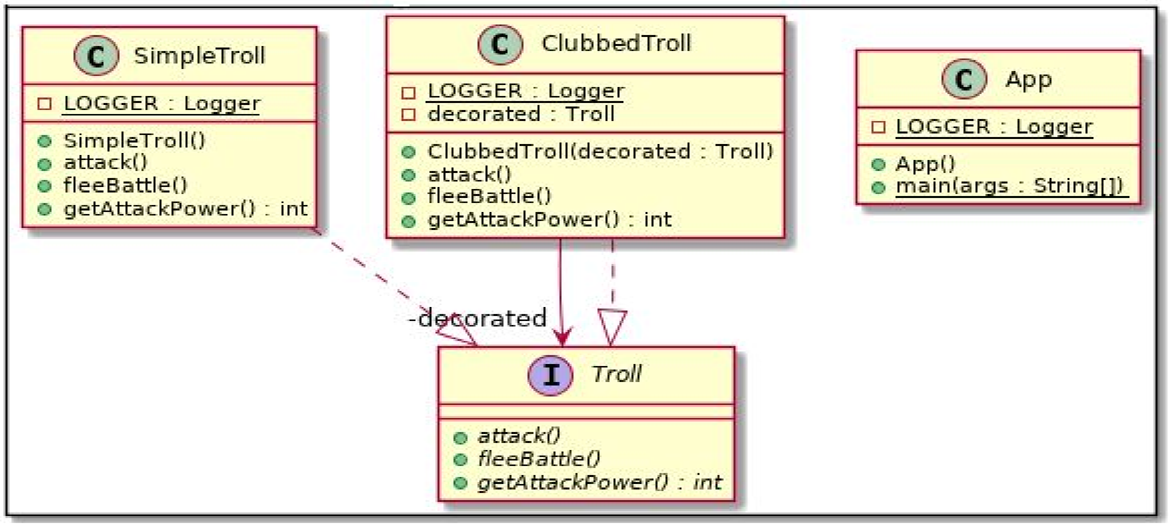
Next we want to add club for the troll. We can do it dynamically by using a decorator

```
public class ClubbedTroll implements Troll {

  private static final Logger LOGGER = LoggerFactory.getLogger(ClubbedTroll.class);

  private Troll decorated;

  public ClubbedTroll(Troll decorated) {
    this.decorated = decorated;
  }

  @Override
  public void attack() {
    decorated.attack();
    LOGGER.info("The troll swings at you with a club!");
  }

  @Override
  public int getAttackPower() {
    return decorated.getAttackPower() + 10;
  }

  @Override
  public void fleeBattle() {

    decorated.fleeBattle();

  }

}
```

Here's the troll in action

```
// simple troll

var troll = new SimpleTroll();

troll.attack(); // The troll tries to grab you!

troll.fleeBattle(); // The troll shrieks in horror and runs away!

// change the behavior of the simple troll by adding a decorator

var clubbedTroll = new ClubbedTroll(troll);

clubbedTroll.attack(); // The troll tries to grab you! The troll swings at you
with a club!

clubbedTroll.fleeBattle(); // The troll shrieks in horror and runs away!
```

Class diagram



5. Implement Proxy design pattern and draw the corresponding class diagram.

Proxy Design Pattern

Github Link :
https://github.com/porcelainruler/EPP_Lab/tree/master/Proxy_Design_Pattern

layout	title	folder	permalink	categories
pattern	Proxy_ Design_ Pattern	EPP_Lab	EPP_Lab/Proxy_Design_Pattern/	Structural Design Pattern

Also known as

Surrogate

Intent

Provide a surrogate or placeholder for another object to control access to it.

Implementation

Real world example

Imagine a tower where the local wizards go to study their spells. The ivory tower can only be accessed through a proxy which ensures that only the first three wizards can enter. Here the proxy represents the functionality of the tower and adds access control to it.

Programmatic Example

Taking our wizard tower example from above. Firstly we have the wizard tower interface and the ivory tower class

```
public interface WizardTower {


  void enter(Wizard wizard);

}


public class IvoryTower implements WizardTower {


  private static final Logger LOGGER =
LoggerFactory.getLogger(IvoryTower.class);


  public void enter(Wizard wizard) {

    LOGGER.info("{} enters the tower.", wizard);

  }


}
```

Then a simple wizard class

```
public class Wizard {
  private final String name;


  public Wizard(String name) {

    this.name = name;

  }


  @Override

  public String toString() {

    return name;

  }

}
```

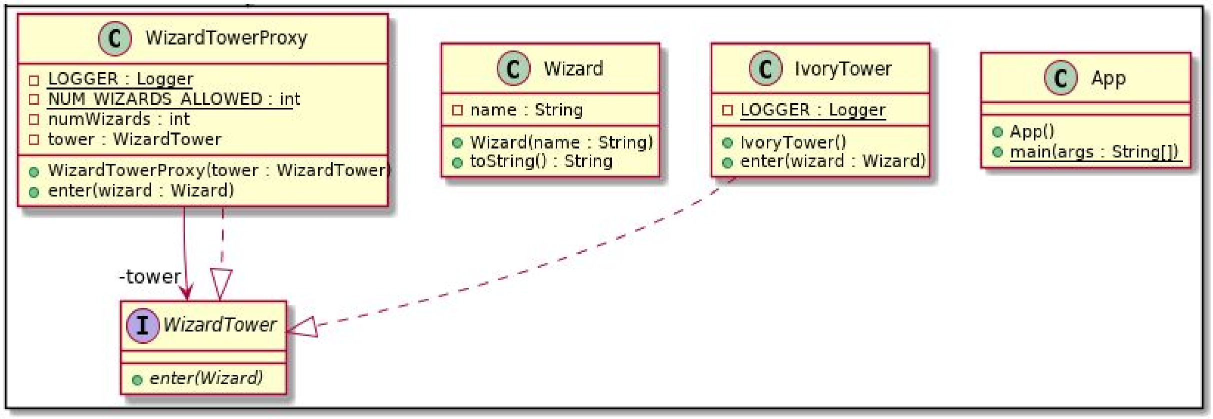
Then we have the proxy to add access control to wizard tower

```
public class WizardTowerProxy implements WizardTower {


  private static final Logger LOGGER =
LoggerFactory.getLogger(WizardTowerProxy.class);


  private static final int NUM_WIZARDS_ALLOWED = 3;


  private int numWizards;
  private final WizardTower tower;


  public WizardTowerProxy(WizardTower tower) {
    this.tower = tower;
  }


  @Override
  public void enter(Wizard wizard) {
    if (numWizards < NUM_WIZARDS_ALLOWED) {
      tower.enter(wizard);
      numWizards++;
    } else {
      LOGGER.info("{} is not allowed to enter!", wizard);
    }
  }
}
```

And here is tower entering scenario

```
var proxy = new WizardTowerProxy(new IvoryTower());
proxy.enter(new Wizard("Red wizard")); // Red wizard enters the tower.
proxy.enter(new Wizard("White wizard")); // White wizard enters the tower.
proxy.enter(new Wizard("Black wizard")); // Black wizard enters the tower.
proxy.enter(new Wizard("Green wizard")); // Green wizard is not allowed to enter!

proxy.enter(new Wizard("Brown wizard")); // Brown wizard is not allowed to enter!
```

Class diagram



6. Implement Template design pattern and draw the corresponding class diagram.

Template Design Pattern

Github Link : https://github.com/porcelainruler/EPP_Lab/tree/master/Template_Design_Pattern

layout	title	folder	permalink	categories
pattern	Template_Design_Pattern	EPP_Lab	EPP_Lab/Template_Design_Pattern/	Behavioral Design Pattern

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

To make sure that subclasses don't override the template method, the template method should be declared `final`.

Implementation

Programmatic Example

Let's take the Thief example. First of all we have a simple thief and we implement its various stealing methods, steal function.

`Libraries :`

```
package com.porcelainruler.templatemethod;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;
```

HalflingThief :

```
/**

 * Halfling thief uses {@link StealingMethod} to steal.

 */

public class HalflingThief {

  private StealingMethod method;

  public HalflingThief(StealingMethod method) {

    this.method = method;

  }



  public void steal() {

    method.steal();

  }

  public void changeMethod(StealingMethod method) {

    this.method = method;

  }

}
```

HitansRunMethod :

```
/**
 * HitAndRunMethod implementation of {@link StealingMethod}.
 */
public class HitAndRunMethod extends StealingMethod {
  private static final Logger LOGGER =
LoggerFactory.getLogger(HitAndRunMethod.class);
  @Override
  protected String pickTarget() {
    return "old goblin woman";
  }
  @Override
  protected void confuseTarget(String target) {
    LOGGER.info("Approach the {} from behind.", target);
  }
  @Override
  protected void stealTheItem(String target) {
    LOGGER.info("Grab the handbag and run away fast!");
  }
}
```

StealingMethod :

```
/**
 * StealingMethod defines skeleton for the algorithm.
 */
public abstract class StealingMethod {
  private static final Logger LOGGER =
LoggerFactory.getLogger(StealingMethod.class);
  protected abstract String pickTarget();
  protected abstract void confuseTarget(String target);
  protected abstract void stealTheItem(String target);
  /**
   * Steal.
   */
  public void steal() {
    var target = pickTarget();
    LOGGER.info("The target has been chosen as {}.", target);
    confuseTarget(target);
    stealTheItem(target);
  }
}
```

SubtleMethod :

```
/**
```

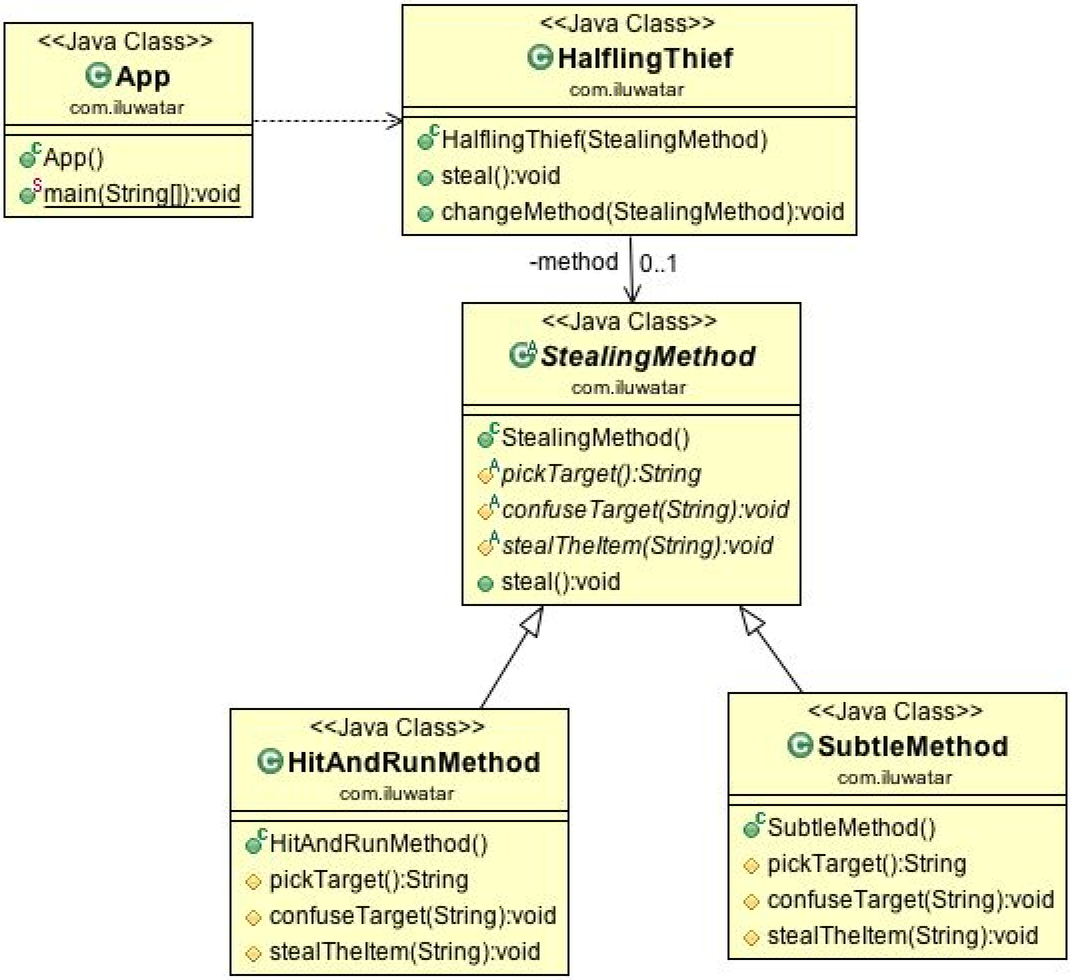
```
 * SubtleMethod implementation of {@link StealingMethod}.
 */
public class SubtleMethod extends StealingMethod {
  private static final Logger LOGGER = LoggerFactory.getLogger(SubtleMethod.class);
  @Override
  protected String pickTarget() {
    return "shopkeeper";
  }
  @Override
  protected void confuseTarget(String target) {
    LOGGER.info("Approach the {} with tears running and hug him!", target);
  }
  @Override
  protected void stealTheItem(String target) {
    LOGGER.info("While in close contact grab the {}'s wallet.", target);
  }
}
```

Here’s How Thief works :

```
/**
```

```
 * Template Method defines a skeleton for an algorithm. The algorithm
subclasses provide
 * implementation for the blank parts.
 *
 * <p>In this example {@link HalflingThief} contains {@link
StealingMethod} that can be changed.
 * First the thief hits with {@link HitAndRunMethod} and then with {@link
SubtleMethod}.
 */
public class App {
  /**
   * Program entry point.
   * @param args command line args
   */
  public static void main(String[] args) {
    var thief = new HalflingThief(new HitAndRunMethod());
    thief.steal();
    thief.changeMethod(new SubtleMethod());
    thief.steal();
  }
}
```

Class diagram



7. Implement the Observer design pattern and draw the corresponding class diagram.

Observer Design Pattern

Github Link :
https://github.com/porcelainruler/EPP_Lab/tree/master/Observer_Design_Pattern

layout	title	folder	permalink	categories
pattern	Observer_Design_Pattern	EPP_Lab	EPP_Lab/Observer_Design_Pattern/	Behavioral Design Pattern

Also known as

Dependents, Publish-Subscribe

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Implementation

Programmatic Example

Taking our wizard tower example from above. Firstly we have the wizard tower interface and the ivory tower class

Libraries :

```
package com.porcelainruler.observer;


import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.util.ArrayList;

import java.util.List;
```

Weather

```
/**

 * Weather can be observed by implementing {@link WeatherObserver}
interface and registering as

 * listener.

 */

public class Weather {
  private static final Logger LOGGER =
LoggerFactory.getLogger(Weather.class);

  private WeatherType currentWeather;
  private List<WeatherObserver> observers;

  public Weather() {
    observers = new ArrayList<>();
    currentWeather = WeatherType.SUNNY;
  }

  public void addObserver(WeatherObserver obs) {
    observers.add(obs);
  }

  public void removeObserver(WeatherObserver obs) {
    observers.remove(obs);
  }

  /**
   * Makes time pass for weather.
   */
  public void timePasses() {
    WeatherType[] enumValues = WeatherType.values();
    currentWeather = enumValues[(currentWeather.ordinal() + 1) %
enumValues.length];
    LOGGER.info("The weather changed to {}.", currentWeather);
    notifyObservers();
  }


  private void notifyObservers() {
    for (WeatherObserver obs : observers) {
      obs.update(currentWeather);
    }
  }
}
```

WeatherType

```
/**
 * WeatherType enumeration.
 */
public enum WeatherType {
  SUNNY, RAINY, WINDY, COLD;
  @Override
  public String toString() {
    return this.name().toLowerCase();
  }
}
```

WeatherObserver

```
/**
 * Observer interface.
 */
public interface WeatherObserver {
  void update(WeatherType currentWeather);
}
```

Orcs

```
/**
 * Orcs.
 */
public class Orcs implements WeatherObserver {
  private static final Logger LOGGER =
LoggerFactory.getLogger(Orcs.class);
  @Override
  public void update(WeatherType currentWeather) {
    switch (currentWeather) {
      case COLD:
        LOGGER.info("The orcs are freezing cold.");
        break;
      case RAINY:
        LOGGER.info("The orcs are dripping wet.");
        break;
      case SUNNY:
        LOGGER.info("The sun hurts the orcs' eyes.");
        break;
      case WINDY:
        LOGGER.info("The orc smell almost vanishes in the wind.");
        break;
      default:
        break;
    }
  }
}
```

Hobbits

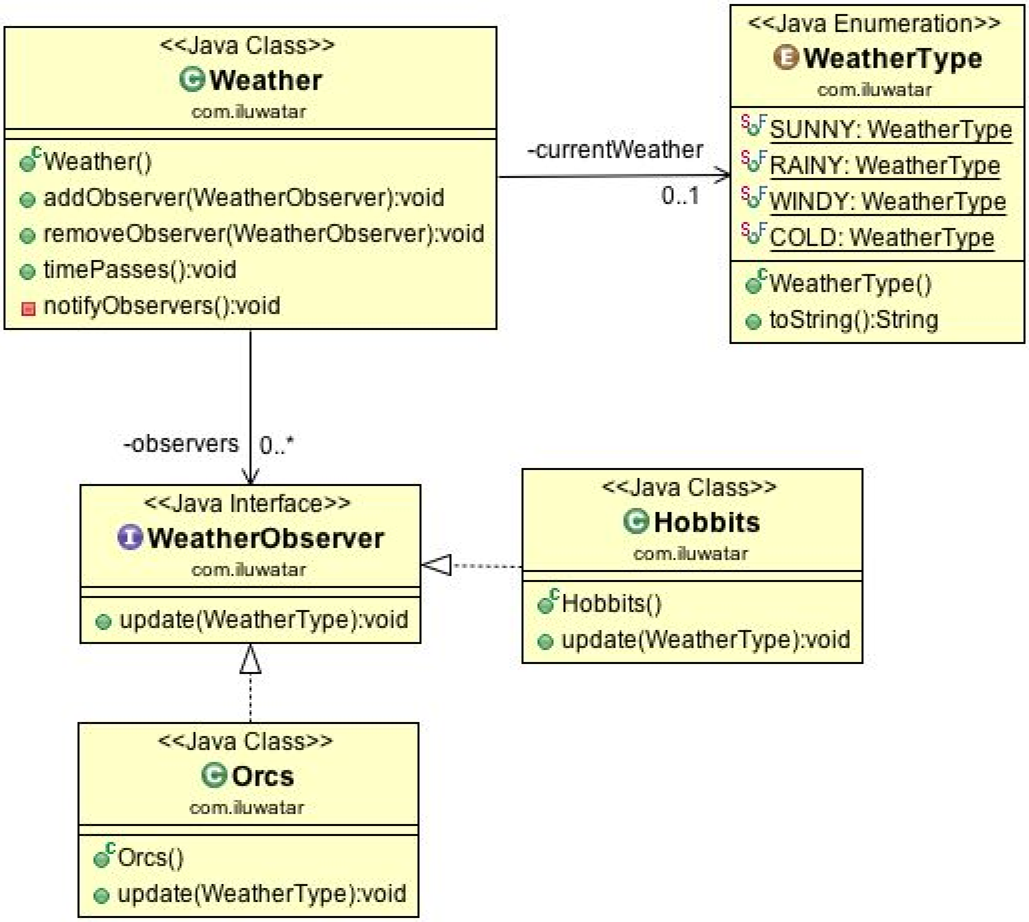
```
/**
```

```
 * Hobbits.
 */
public class Hobbits implements WeatherObserver {
  private static final Logger LOGGER = LoggerFactory.getLogger(Hobbits.class);
  @Override
  public void update(WeatherType currentWeather) {
    switch (currentWeather) {
      case COLD:
        LOGGER.info("The hobbits are shivering in the cold weather.");
        break;
      case RAINY:
        LOGGER.info("The hobbits look for cover from the rain.");
        break;
      case SUNNY:
        LOGGER.info("The happy hobbits bade in the warm sun.");
        break;
      case WINDY:
        LOGGER.info("The hobbits hold their hats tightly in the windy weather.");
        break;
      default:
        break;

    }

  }

}
```

And here is how WeatherObserver observes and updates the weather scenario :

```
package com.porcelainruler.observer;



import com.porcelainruler.observer.generic.GHobbits;

import com.porcelainruler.observer.generic.GOrcs;

import com.porcelainruler.observer.generic.GWeather;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;



/**

 * The Observer pattern is a software design pattern in which an object, called the subject,

 * maintains a list of its dependents, called observers, and notifies them automatically of any

 * state changes, usually by calling one of their methods. It is mainly used to implement

 * distributed event handling systems. The Observer pattern is also a key part in the familiar
 * model-view-controller (MVC) architectural pattern. The Observer pattern is implemented in

 * numerous programming libraries and systems, including almost all GUI toolkits.

 *

 * <p>In this example {@link Weather} has a state that can be observed. The {@link Orcs} and {@link

 * Hobbits} register as observers and receive notifications when the {@link Weather} changes.

 */

public class App {



  private static final Logger LOGGER = LoggerFactory.getLogger(App.class);



  /**

   * Program entry point.

   *

   * @param args command line args

   */

  public static void main(String[] args) {



    Weather weather = new Weather();

    weather.addObserver(new Orcs());
    weather.addObserver(new Hobbits());

    weather.timePasses();
    weather.timePasses();
    weather.timePasses();
    weather.timePasses();

    // Generic observer inspired by Java Generics and Collection by Naftalin & Wadler
    LOGGER.info("--Running generic version--");
    GWeather genericWeather = new GWeather();
    genericWeather.addObserver(new GOrcs());
    genericWeather.addObserver(new GHobbits());

    genericWeather.timePasses();
    genericWeather.timePasses();
    genericWeather.timePasses();
    genericWeather.timePasses();
  }
}
```

Class diagram



8. Implement Iterator design pattern and draw the corresponding class diagram.

Iterator Design Pattern

Github Link : https://github.com/porcelainruler/EPP_Lab/tree/master/Iterator_Design_Pattern

layout	title	folder	permalink	categories
pattern	Iterator_Design_Pattern	EPP_Lab	EPP_Lab/Iterator_Design_Pattern/	Behavioral Design Pattern

Also known as

Cursor

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Implementation

Programmatic Example

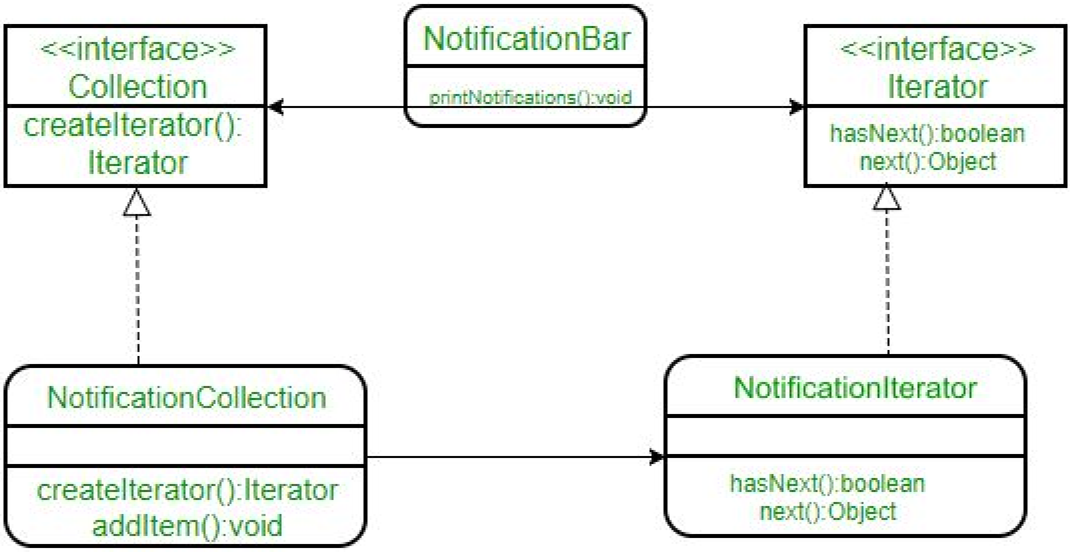
Suppose we are creating a notification bar in our application that displays all the notifications which are held in a notification collection. NotificationCollection provides an iterator to iterate over its elements without exposing how it has implemented the collection (array in this case) to the Client (NotificationBar).

```
// A simple Notification class
class Notification
{
    // To store notification message
    String notification;
    public Notification(String notification)
    {
        this.notification = notification;
    }
    public String getNotification()
    {
        return notification;
    }
}
// Collection interface
interface Collection
{
    public Iterator createIterator();
```

```
}


// Collection of notifications
class NotificationCollection implements Collection
{
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    Notification[] notificationList;

    public NotificationCollection()
    {
        notificationList = new Notification[MAX_ITEMS];
        // Let us add some dummy notifications
        addItem("Notification 1");
        addItem("Notification 2");
        addItem("Notification 3");
    }
    public void addItem(String str)
    {
        Notification notification = new Notification(str);
        if (numberOfItems >= MAX_ITEMS)
            System.err.println("Full");
        else
        {
            notificationList[numberOfItems] = notification;
            numberOfItems = numberOfItems + 1;
        }
    }
    public Iterator createIterator()
    {
        return new NotificationIterator(notificationList);
    }
}
// We could also use Java.Util.Iterator
interface Iterator
{
    // indicates whether there are more elements to
    // iterate over
    boolean hasNext();
    // returns the next element
    Object next();
}
// Notification iterator
class NotificationIterator implements Iterator
```

```
{

    Notification[] notificationList;

    // maintains curr pos of iterator over the array

    int pos = 0;

    // Constructor takes the array of notifiactionList are

    // going to iterate over.

    public  NotificationIterator (Notification[] notificationList)

    {

        this.notificationList = notificationList;

    }

    public Object next()

    {

        // return next element in the array and increment pos

        Notification notification =  notificationList[pos];

        pos += 1;

        return notification;

    }

    public boolean hasNext()

    {

        if (pos >= notificationList.length ||

            notificationList[pos] == null)

            return false;
        else
            return true;
    }
}
// Contains collection of notifications as an object of
// NotificationCollection
class NotificationBar
{
    NotificationCollection notifications;
    public NotificationBar(NotificationCollection notifications)
    {
        this.notifications = notifications;
    }
    public void printNotifications()
    {
        Iterator iterator = notifications.createIterator();
        System.out.println("-------NOTIFICATION BAR------------");
        while (iterator.hasNext())
        {
            Notification n = (Notification)iterator.next();
            System.out.println(n.getNotification());
        }
    }

}

// Driver class

class Main

{

    public static void main(String args[])

    {

        NotificationCollection nc = new NotificationCollection();

        NotificationBar nb = new NotificationBar(nc);

        nb.printNotifications();

    }

}
```

Class diagram


<<interface>>
Collection
createIterator():
Iterator
NotificationBar
printNotifications():void
<<interface>>
Iterator
hasNext():boolean
next():Object
NotificationCollection
createIterator():Iterator
addItem():void
NotificationIterator
hasNext():boolean
next():Object