Екзаменаційна робота

Пупова Н.А

Група К-29

Білет 19

Принципи YAGNI та OCP

**“Yagni”** – акронім від “You Aren't Gonna Need It”. Цей каже, що розробникам треба фокусуватися на розробці тільки основного функціоналу програми. Не витрачати час на надлишковий функціонал(в якому немає безпосередньої потреби).

Використання принципу “Yagni” має такі плюси:

* Час витрачається тільки на необхідний функціонал, таким чином більша ймовірність вкластися в дедлайни
* Не треба документувати надлишкові функції
* Надлишковий функціонал не обмежує та не ускладнює додавання нових необхідних можливостей до програми
* Програма залишається відносно простою

Як приклад порушення використання цього принципу може служити такий код:

public class Ship {  
 private String name;  
 private double maxSpeed;  
   
 public void swim() {  
 //not done yet  
 }  
   
 public void transport\_by\_bus() {  
 //focusing on this redundant feature  
 }  
}

Тут команда розробляє корабель і витрачає весь час на те, щоб корабель міг перевозитися автобусом(надлишковий функціонал). Однак, основний функціонал(пливти) поки не реалізовано.

**“OCP” –** “Open Closed Principle”, один з принципів S**O**LID. Означає, що програмні сутності мають бути відкритими для розширення та закритими для змін.

Як я це розумію: треба розробляти програму так, щоб при додаванні нового компоненту(класу), не треба було змінювати вже існуючий код. “Open Closed Principle” надає перевагу використанню успадкування та реалізації інтерфейсів.

Код, який порушує OCP:

public class SortUtil {  
 public static void sort(String algorithmName) {  
 if(algorithmName.equals("QuickSort")) {  
 new QuickSort.sort();  
 } else if(algorithmName.equals("HeapSort")) {  
 new HeapSort.sort();  
 } else {  
 throw NoSuchAlgorithmException("No such algorithm: " + algorithmName);  
 }  
 }  
}

Цей клас використовується для сортування. Якщо ми захочемо додати новий вид сортування, нам треба буде модифікувати вже існуючий код(додавати новий if). Згодом, цей метод може стади досить складним та налічувати багато if.

Замість цього варто використовувати успадкування.

Створити інтерфейс Sorting

public interface Sorting {  
 public void sort();  
}

Створити алгоритми сортування, що реалізують цей інтерфейс:

public class HeapSort implements Sorting {  
 @Override  
 public void sort() {  
 //do sort  
 }  
}

public class QuickSort implements Sorting {  
 @Override  
 public void sort() {  
 //do Sort  
 }  
}

Та використовувати Sorting як аргумент методу sort:

public class SortUtil {  
 public static void sort(Sorting algo) {  
 algo.sort();  
 }  
}

Таким чином, метод став набагато меншим.

А якщо ми забажаємо додати новий метод сортування, то ми просто створимо новий клас, що буде успадковувати Sorting. При цьому вже існуючий код ми не будемо чіпати

Порівняти патерни Observer, Singleton, Abstract Factory

**Singleton** використовується якщо нам потрібен тільки один обьект даного класу протягом роботи програми. Як часто використовуваний приклад можна взяти кеш:

public class Cache {  
 private static Cache *instance*;  
   
 public static Cache getInstance() {  
 if(*instance* == null) {  
 *instance* = new Cache();  
 }  
 return *instance*;  
 }  
   
 private Map<String, String> map;  
   
 //private constructor to avoid new instantiations  
 private Cache() {  
   
 }  
   
 public void put(String key, String value) {  
 map.put(key, value);  
 }  
   
 public String get(String key) {  
 return map.get(key);  
 }  
}

Використання:

public class UserService {  
 private UserDatabase userDatabase;  
   
 public String getUserInfo(String username) {  
 String infoInCache = Cache.*getInstance*().get(username);  
 //if no user info in cache  
 if(infoInCache == null) {  
 //then load info from DB  
 infoInCache = userDatabase.getInfo(username);  
 //put info in cache  
 Cache.*getInstance*().put(username, infoInCache);  
 }  
 return infoInCache;  
 }  
}

**Abstract Factory** – фабрика, що створює різні сімейства обьектів. Тобто ми створюємо сімейства пов’язаних об’єктів, не прив’язуючись до конкретних класів створюваних об’єктів.

Як приклад, у нас є декілька сімейств, що знаходяться в різниї ієрархіях классів:

AudiCar, BmwCar, AudiSportCar, BmwSportCar.

Приклад:

Є інтерфейси Car та SportCar:

public interface Car {  
 public void go();  
}

public interface SportCar {  
 public void fastGo();  
}

Є їх імплементації

public class AudiCar implements Car {  
 @Override  
 public void go() {  
 //go  
 }  
}

public class BmwCar implements Car{  
 @Override  
 public void go() {  
   
 }  
}

public class AudiSportCar implements SportCar {  
 @Override  
 public void fastGo() {  
  
 }  
}

public class BmwSportCar implements SportCar {  
 @Override  
 public void fastGo() {  
   
 }  
}

Абстрактна фабрика:

public interface AbstractFactory {  
 public Car car();  
 public SportCar sportCar();  
}

Та її імплементації

public class AudiFactory implements AbstractFactory {  
 @Override  
 public Car car() {  
 return new AudiCar();  
 }  
  
 @Override  
 public SportCar sportCar() {  
 return new AudiSportCar();  
 }  
}

public class BmwFactory implements AbstractFactory{  
 @Override  
 public Car car() {  
 return new BmwCar();  
 }  
  
 @Override  
 public SportCar sportCar() {  
 return new BmwSportCar();  
 }  
}

**Subscriber** створює механізм підписки, що дає можливість одним обьектам реагувати на подію іншого обьекта

Як приклад можна привести youtube блогера та його глядачів. Коли актор виконує метод uploadVideo() він викликає метод showNotification у його підписників:

Youtube bloger = publisher

Audience = subscriber

public abstract class YoutubeSubscriber {  
 public abstract void showNotification();  
 public void subscribe(YoutubeBloger bloger) {  
 bloger.subscribe(this);  
 }  
}

public class YoutubeBloger {  
 private List<YoutubeSubscriber> subscribers;  
  
 public void uploadVideo() {  
 //upload video  
 for(YoutubeSubscriber subscriber: subscribers) {  
 subscriber.showNotification();  
 }  
 }  
   
 public void subscribe(YoutubeSubscriber subscriber) {  
 subscribers.add(subscriber);  
 }  
}

І його глядачі:

public class Vasya extends YoutubeSubscriber {  
 @Override  
 public void showNotification() {  
 //show notification on Iphone and Macbook  
 }  
}

public class Kolya extends YoutubeSubscriber {  
 @Override  
 public void showNotification() {  
 //show notification on Nokia  
 }  
}

Вони колись підписалися на YoutubeBlogger

public class Internet {  
 public void subscribes(YoutubeSubscriber subscriber, YoutubeBloger bloger) {  
 subscriber.subscribe(bloger);  
 }  
}

І тепер коли блогер загрузить нове відео, його підписники отримають повідомлення.

Спроектувати (наприклад, використовуючи UML) засіб для генерації документів на основі шаблону та заповнення даних (наприклад, білети до іспиту).

