TSE3151

SOFTWARE DESIGN

Behavioral Design Pattern Assignment

AHMED HASSAN MOHAMMED SALEM

(1191102340)

FACULTY OF COMPUTING AND INFORMATICS

MULTIMEDIA UNIVERSITY

29th December 2022

Acknowledgment

I would like to express my acknowledgement and my warmest thanks to my supervisor **Ts. Dr. Ho Sin Ban**. His guidance and tips regarding the project guidelines have helped me through the different stages during this project. Also, His stimulating suggestions and encouragements have also allowed me to write this report and complete it.

I also highly appreciate my colleagues who helped me by giving me ideas, tips, or even clarifying some doubts. I also thank all the people who willingly assisted me with their abilities.

Finally, I would like to thank my family members who played an important role in convincing me to pursue this project. I would like to thank my parents, who gave me all the love and guidance needed. For me, they are the ultimate role models to look up to in my life.

Contents

1. Abstract 1
2. Introduction to the project 2
   2.1 Introduction 2
   2.2 Problem Statement 2
   2.3 Project Objectives 3
   2.4 Project Scope 4
3. Literature Review 4
4. Body 6
   4.1 Question 1 6
   4.2 Question 2 17
   4.3 Question 3 26
5. Conclusion and suggestions 33
   5.1 Conclusion 33
   5.2 Suggestions 33
6. Bibliography 34

1. Abstract

Applying design patterns in software development process tend to be proven solutions to problems that are occurring frequently. This is because it helps to achieve flexibility and maintainability in the software design.

First in this report we will look at the background study of behavioral design patterns to understand the basics more. Also, we will investigate different types of behavioral design patterns without going into details

Also in this report, we discussed three behavioral design patterns: Observer, Visitor and Command. We demonstrate sample runs and explained about the benefits and drawbacks of apply those design patterns in the case study questions provided in tutorial 6

The result of this report was that these design patterns tend to be important as they are used to establish relationships between objects. First of all, the Observer Design pattern allows the new DVD subscribers to be added to the DVD release program by using minimum effort without need to modify the DvdReleaseByCategory class hence, it achieves easy extensibility of the program. Additionally, the visitor design patterns make it easy to add new features to the Blurb & TitleInfo program without the need of modifying the implementation of the underlying objects. Lastly, the command design pattern offers decoupling of objects. Therefore, this makes it easy to change and manipulate the DvdStarsOnOff program

2. Introduction to the project

2.1 Introduction

The importance of design patterns in software development tends to be an important factor to consider. Design patterns provide acceleration to the process of developing software. They provide solutions to frequently repeated problems faced by software developers.

In this report, we will be learning about three types of behavioral design pattern: Observer, Visitor and Command. Moreover, we will demonstrate how these design patterns can be used in real life practice by providing sample runs for programs from Tutorial 6 questions A, B and D. Furthermore, we will analyze the result of the sample run by providing screenshots of the output. Also, we will provide the mapping table for the components of the design patterns to the components of the program. Additionally, we will provide both the general UML class structure as well as the applied UML class structure according to the question for better understanding. Next, we will discuss the advantages and disadvantages of using the design pattern in each of the questions. Finally, we will provide conclusions and suggestions for future work related to the design patterns discussed and how they can be improved.

2.2 Problem Statement

This report focuses on the need to identify solutions that are flexible and reusable at the same time. This will help address the design problems that are common in Object Oriented programming.

In this report we will discuss the Observer, Visitor and Command design patterns. These patterns tend to provide efficient and effective ways to establish relationships between different objects.

However, the main problem lies in understanding these design patterns. Due to their complex nature and structure, many sources on the internet tend to use large programs to explain the idea behind these patterns. As a result, in this report we will focus on providing simple, clear and concise programs to explain the design patterns.

2.3 Project Objectives

The objectives of this project are as follows:

- Provide students with clear, simple and easy to understand explanations of the Observer, Visitor and Command design patterns
- Provide students with better understanding of the advantages and disadvantages of Observer, Visitor and Command design patterns
- Provide real life example demonstration of simple sample runs to help students understand the design patterns in a better way
- To help students use these patterns in their final year project so that they use a more flexible and reusable solution in their codes

2.4 Project Scope

The scope of this report is to introduce the methodology behind using Observer, Visitor and Command design patterns. The report will demonstrate sample runs of these patterns by using real life examples.

The report will not include advanced programs or large projects that used these design patterns. Instead, we will try to make it simple for students to learn the basics by using the simplest examples. Hence, this will provide a solid background knowledge for final year students who will apply these design patterns in their future projects.

The report also acts as a source and a step-by-step guidance to anyone who wants to learn behavioral design patterns and effective ways of using them.

3. Literature Review

Firstly, Design Patterns is a set of solutions that are reusable to a frequently occurring software design problem that software developers face during the development stage. It is basically a template to describe how a particular problem can be interpreted and solved. “Design Patterns tend to solve reoccurring software problems by providing solutions to overcome them.” (Wedyan & Abufakher, 2019). Also, they improve the quality of the software by suggesting better design decisions in the software development stage. Additionally, “benefits of applying design patterns include reducing development cost, improving code quality, and standardizing the integration and maintenance processes. Therefore, using design patterns is becoming a common practice to build both commercial software and open-source products. " (Liguo Yu, Yingmei Li, & Srini Ramaswamy, 2018)

Moving on, Behavioral Design Pattern are design patterns that focuses more on the patterns that are identical in the communication between the objects. Also "Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects." (McGahagan, 2013). Moreover, Behavioral Design Patterns tend to make the use of inheritance property to allocate the behaviors between the classes. Also, encapsulated is used to delegate the behaviors to the objects. Furthermore, the types of Behavior Design Pattern include:

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

Moreover, Observer Pattern is a Behavior Design Pattern that is applicable in scenarios where there is a one-to-many relationship between the objects. Hence, if any object state changes the other objects related to it must be automatically notified. "This object-oriented design scheme ensures that an observer object or a set of observer objects automatically perform appropriate actions when required to do so by an observable object.” (Eales, 2005).

Furthermore, Visitor Pattern is a Behavior Design Pattern which replaces the implementation algorithm of an element class. Hence, the behavior of the algorithm might vary from one visitor class to another. The “Visitor lets you define a new operation without changing the classes of the elements on which it operates.” (Jeanmart, Gu´eh´eneuc, Sahraoui, & Habra, 2014)

Last but not least, the Command Pattern is a Behavior Design Pattern in which a request is allocated for an object in a form of command and delivered to the object which is responsible for invoking it. Then the object that invokes the request will search for the appropriate object that can take on to handle the request. Finally, the request will be passed to the appropriate object for execution. The command design pattern is from the behavioral category. “This pattern is used to encapsulate the client's request (in the form of the method name and the value of the method's parameters) as an object. Moreover, it also promotes the invocation of a method on an object.” (Hussain, et al., 2019)

4. Body

4.1 Question 1

The Observer Design Pattern is usually used to establish one-to-many relationships between objects. This means the objects that are dependent on the subject will be notified automatically once any object is modified. Due to this characteristic, the Observer Pattern tends to fall under the category of Behavioral Design Pattern.

Secondly, let’s try to first understand the components of an Observer Pattern so we can understand better how the code in **Tutorial 6 Section A** is being executed. An Observer Pattern consists of the following components:

- **Subject:** This identifies and knows the observer in the system. This component tends to have methods to link and establish a connection with observers to an object which is the client
- **Observer:** This component creates an interface for the objects that must be informed if there are any changes in the subject state
- **Concrete Object:** This component keeps the state of interest of Concrete Observer stored in it. Then it sends a notification to the Observers when the state changes
- **Concrete Observer:** This component keeps a reference to the Concrete Subject. It also keeps a state that should be aligned with the Concrete Subject state of interest. It also uses the Observer interface for updating and keeping the state and keeping it consistent.
- **Concrete Subject:** This component informs the observers if any change has happened that could result in making the observer state inconsistent with the Concrete Subject state.
- **Concrete Observer:** This component requests information from the subject to confirm the updates occurred in an object

Next, lets map the observer pattern components in a mapping table:

	Name in DP	Actual Name

1.	Subject	**DvdReleaseByCategory** - the class which is observed
	observersAggregationVariable	**subscriberList**
	attach(Observer)	**addSubscriber(DvdSubscriber)**
	detach(Observer)	**removeSubscriber (DvdSubscriber)**
	notify()	**notifySubscribersOfNewDvd(DvdRelease),** **notifySubscribersOfUpdate(DvdRelease)**
2.	Observer	**DvdSubscriber** - the observer
	update()	**updateDvdRelease(DvdRelease, String)**
3	Client	**TestDvdObserver**

Table 1 shows the mapping table for Tutorial 6 Section A

Now, since we understood the components of the Observer Pattern, let's look at the structure of the UML Class Diagram of an Observer Pattern

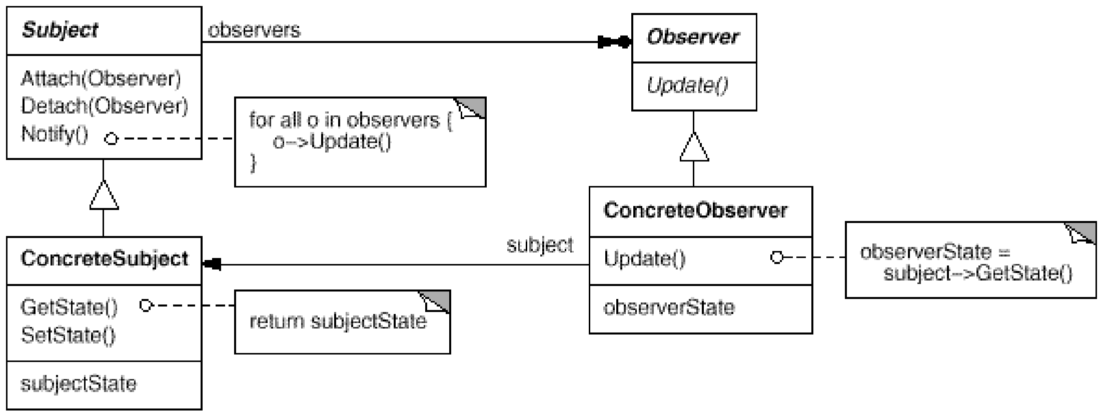


Figure 4-1 shows the general UML class diagram for Observer Design Pattern (Observer Pattern, 2022)

Moving on further, let's apply the above class diagram to our Tutorial 6 Section A case study:

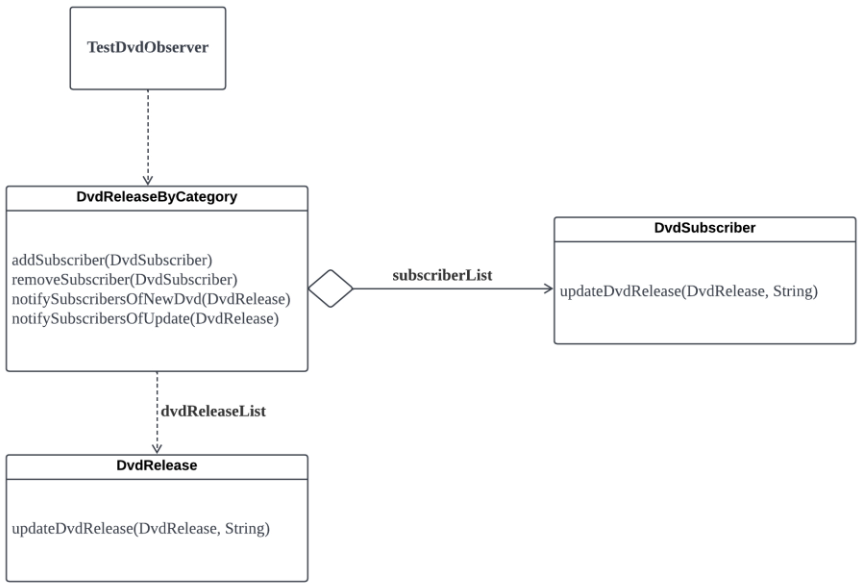
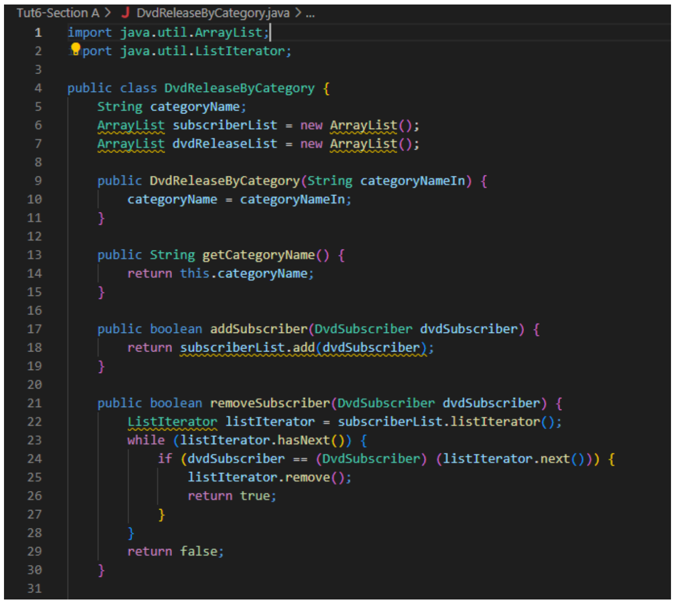

TestDvdObserver
DvdReleaseByCategory
addSubscriber(DvdSubscriber)
removeSubscriber(DvdSubscriber)
notifySubscribersOfNewDvd(DvdRelease)
notifySubscribersOfUpdate(DvdRelease)
subscriberList
DvdSubscriber
updateDvdRelease(DvdRelease, String)
dvdReleaseList
DvdRelease
updateDvdRelease(DvdRelease, String)

Figure 4-2 shows the UML class diagram for Tutorial 6 Section A

Moving on, as we now have a clearer idea about the Observer Pattern and its structure, we can now look and understand the execution of the program in **Tutorial 6 Section A** which is basically a DVD Release Program. It's quite an interesting program to understand.



```java
import java.util.ArrayList;
import java.util.ListIterator;

public class DvdReleaseByCategory {
    String categoryName;
    ArrayList subscriberList = new ArrayList();
    ArrayList dvdReleaseList = new ArrayList();

    public DvdReleaseByCategory(String categoryNameIn) {
        categoryName = categoryNameIn;
    }

    public String getCategoryName() {
        return this.categoryName;
    }

    public boolean addSubscriber(DvdSubscriber dvdSubscriber) {
        return subscriberList.add(dvdSubscriber);
    }

    public boolean removeSubscriber(DvdSubscriber dvdSubscriber) {
        ListIterator listIterator = subscriberList.listIterator();
        while (listIterator.hasNext()) {
            if (dvdSubscriber == (DvdSubscriber) (listIterator.next())) {
                listIterator.remove();
                return true;
            }
        }
        return false;
    }

```

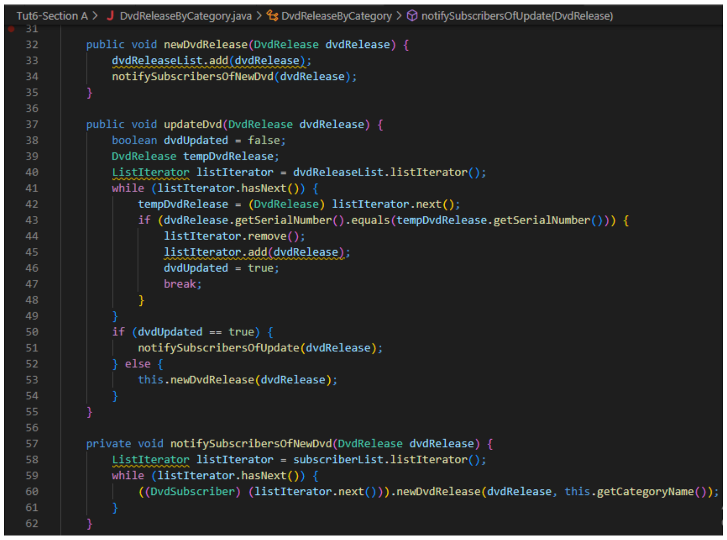
Figure 4-3 shows the code snippet of DvdReleaseByCategory class

First, as observed from the above figure, we will need to create a Subject which will be the class that is observed in this program. The class created will be DVDReleaseByCategory. This class will accept parameters for the constructors which will be used to initialize the Category Name attribute of the DVD release.

Also, this class will have two arrays which is subscriberList as well as dvdReleaseList. The subscriberList and dvdReleaseList are the observer's aggregation variables.

Additionally, the class will also have methods to addSubscriber(attach) and removeSubscriber (detach). The add method subscribes the DvdSubscriber **(**Observer) to the DvdReleaseByCategory (Subject). As a result, if there are any changes in the subject state, it can be updated. The remove method unsubscribes the DvdSubscriber from the DvdReleaseByCategory. Therefore, it gets updated about the changes in the subject state.

These methods will accept a class type DvdSubscriber which is basically the observer in this case.



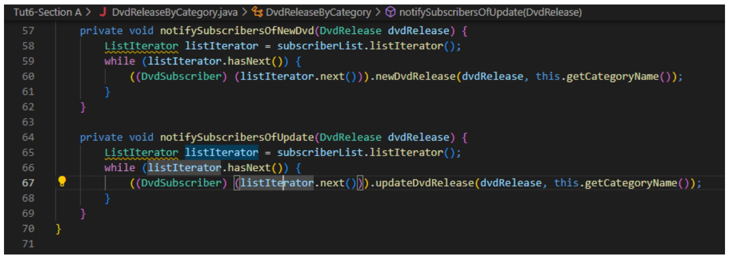
Tut6-Section A > DvdReleaseByCategory.java > DvdReleaseByCategory > notifySubscribersOfUpdate(DvdRelease)

```java

    public void newDvdRelease(DvdRelease dvdRelease) {
        dvdReleaseList.add(dvdRelease);
        notifySubscribersOfNewDvd(dvdRelease);
    }

    public void updateDvd(DvdRelease dvdRelease) {
        boolean dvdUpdated = false;
        DvdRelease tempDvdRelease;
        ListIterator listIterator = dvdReleaseList.listIterator();
        while (listIterator.hasNext()) {
            tempDvdRelease = (DvdRelease) listIterator.next();
            if (dvdRelease.getSerialNumber().equals(tempDvdRelease.getSerialNumber())) {
                listIterator.remove();
                listIterator.add(dvdRelease);
                dvdUpdated = true;
                break;
            }
        }
        if (dvdUpdated == true) {
            notifySubscribersOfUpdate(dvdRelease);
        } else {
            this.newDvdRelease(dvdRelease);
        }
    }

    private void notifySubscribersOfNewDvd(DvdRelease dvdRelease) {
        ListIterator listIterator = subscriberList.listIterator();
        while (listIterator.hasNext()) {
            ((DvdSubscriber) (listIterator.next())).newDvdRelease(dvdRelease, this.getCategoryName());
        }
    }
```

Figure 4-4 shows the code snippet of DvdReleaseByCategory Class

Next, as observed in Fig 4-4, the method newDvdRelease will accept parameter of class type DvdRelease**.** This method will simply add the DVD release to the dvdReleaseList which is a list storing all the DVD releases.

After that, a method called updateDvd will be used to update the DVD releases. This method accepts a parameter of class type DvdRelease and will perform an update only if the serial number exists and matches the intended DVD release to update. The method will remove the current DVD release and replace it with the updated one.


Tut6-Section A > DvdReleaseByCategory.java > DvdReleaseByCategory > notifySubscribersOfUpdate(DvdRelease)

```java
    private void notifySubscribersOfNewDvd(DvdRelease dvdRelease) {
        ListIterator listIterator = subscriberList.listIterator();
        while (listIterator.hasNext()) {
            ((DvdSubscriber) (listIterator.next())).newDvdRelease(dvdRelease, this.getCategoryName());
        }
    }

    private void notifySubscribersOfUpdate(DvdRelease dvdRelease) {
        ListIterator listIterator = subscriberList.listIterator();
        while (listIterator.hasNext()) {
            ((DvdSubscriber) (listIterator.next())).updateDvdRelease(dvdRelease, this.getCategoryName());
        }
    }
}
```

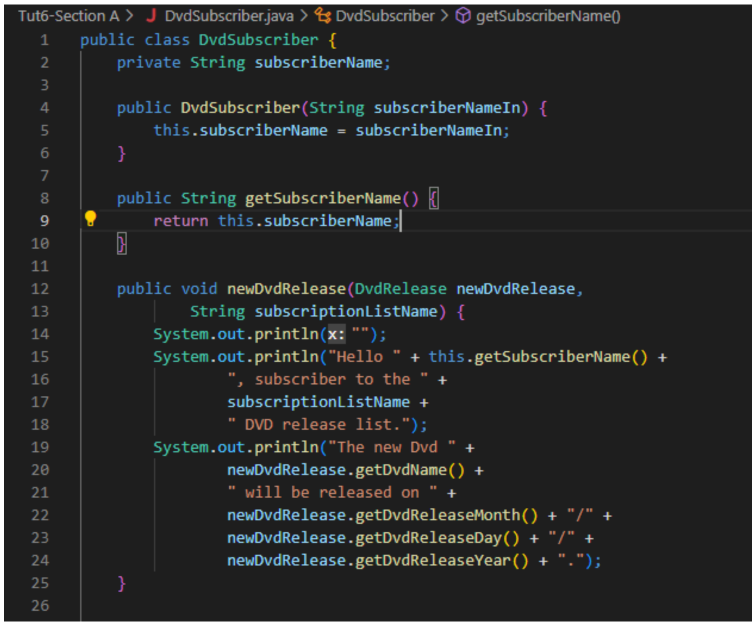
Figure 4-5 shows the code snippet of DvdReleaseByCategory Class

In case an update occurs, there will be a Boolean value that is triggered and a method to notify subscribers about the update which will be notifySubscribersOfUpdate will get triggered too.

Else, if the serial number does not exist, we will just create a new DVD release and notify the subscribers of it by using the notify notifySubscribersOfNewDvd**.**

As we can see, the notifySubscribersOfUpdate and notifySubscribersOfNewDvd are basically the notify components in the Observer Pattern. These methods update all the DvdReleaseByCategory objects by iterating through the list and use each list element method to update.

As we have now understood the code execution for the DvdReleaseByCategory which is a Subject component, let’s move on and look at the code execution in the Observer component which will be the DvdSubscriber in our case. This class will provide an update method. This method will be called by DvdReleaseByCategory class to notify it if there are any changes that has occurred in the state of the subject.

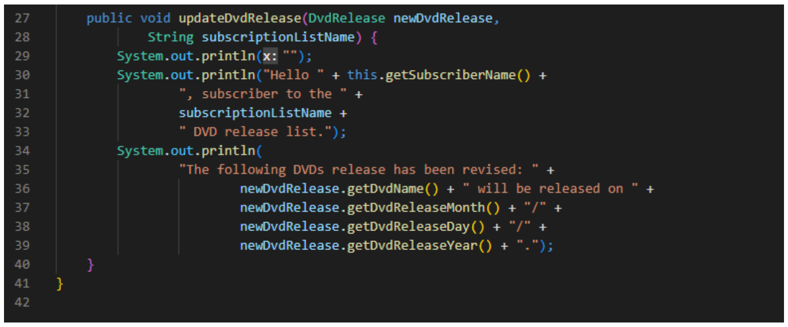

Tut6-Section A > DvdSubscriber.java > DvdSubscriber > getSubscriberName()

```java
public class DvdSubscriber {
    private String subscriberName;

    public DvdSubscriber(String subscriberNameIn) {
        this.subscriberName = subscriberNameIn;
    }

    public String getSubscriberName() {
        return this.subscriberName;
    }

    public void newDvdRelease(DvdRelease newDvdRelease,
            String subscriptionListName) {
        System.out.println(x: "");
        System.out.println("Hello " + this.getSubscriberName() +
                ", subscriber to the " +
                subscriptionListName +
                " DVD release list.");
        System.out.println("The new Dvd " +
                newDvdRelease.getDvdName() +
                " will be released on " +
                newDvdRelease.getDvdReleaseMonth() + "/" +
                newDvdRelease.getDvdReleaseDay() + "/" +
                newDvdRelease.getDvdReleaseYear() + ".");
    }

```

Figure 4-6 shows the code snippet for DvdSubscriber Class

First, as observed in the above figure, the constructor of the observer class will accept a parameter of type string. This string will be the subscriber's name and will be used to initialize the subscriberName attribute in the DvdSubscriber class.

Next, the newDvdRelease method will take the DvdRelease object, which will contain the information about the new release, and subscriptionListName**.** This method will identify which category of subscriptions list the new DVD release falls into. After that the method prints the all the details of the new DVD release such as release name, month, day and so on.

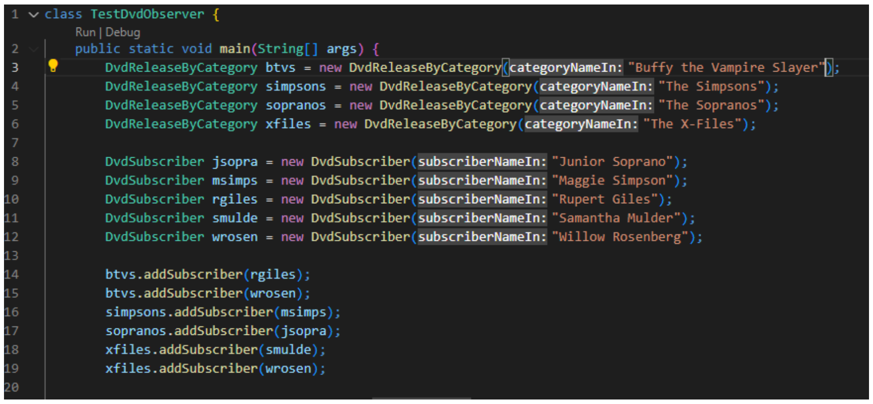


```java
    public void updateDvdRelease(DvdRelease newDvdRelease,
            String subscriptionListName) {
        System.out.println(x: "");
        System.out.println("Hello " + this.getSubscriberName() +
                ", subscriber to the " +
                subscriptionListName +
                " DVD release list.");
        System.out.println(
                "The following DVDs release has been revised: " +
                        newDvdRelease.getDvdName() + " will be released on " +
                        newDvdRelease.getDvdReleaseMonth() + "/" +
                        newDvdRelease.getDvdReleaseDay() + "/" +
                        newDvdRelease.getDvdReleaseYear() + ".");
    }
}
```

Figure 4-7 shows the code snippet of DvdSubscriber Class

After that, the updateDvdRelease method will notify the user about any updates conducted in any of the DVDs. The method is like newDvdRelease; however, it prints the updated information about the DVD. As you may notice, this is basically the update method in the observer component.

Now since we have understood the execution of the major code parts in the program, let's look at the TestDvdObserver to run the program and see the output.

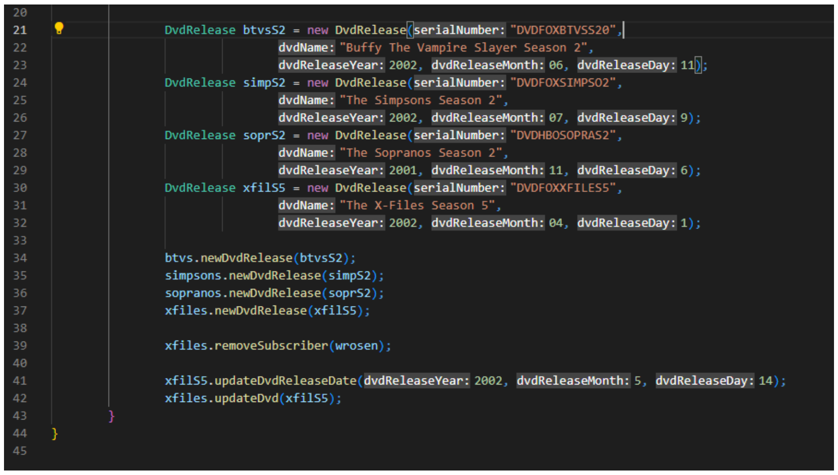


```java
class TestDvdObserver {
    Run | Debug
    public static void main(String[] args) {
        DvdReleaseByCategory btvs = new DvdReleaseByCategory(categoryNameIn: "Buffy the Vampire Slayer");
        DvdReleaseByCategory simpsons = new DvdReleaseByCategory(categoryNameIn: "The Simpsons");
        DvdReleaseByCategory sopranos = new DvdReleaseByCategory(categoryNameIn: "The Sopranos");
        DvdReleaseByCategory xfiles = new DvdReleaseByCategory(categoryNameIn: "The X-Files");

        DvdSubscriber jsopra = new DvdSubscriber(subscriberNameIn: "Junior Soprano");
        DvdSubscriber msimps = new DvdSubscriber(subscriberNameIn: "Maggie Simpson");
        DvdSubscriber rgiles = new DvdSubscriber(subscriberNameIn: "Rupert Giles");
        DvdSubscriber smulde = new DvdSubscriber(subscriberNameIn: "Samantha Mulder");
        DvdSubscriber wrosen = new DvdSubscriber(subscriberNameIn: "Willow Rosenberg");

        btvs.addSubscriber(rgiles);
        btvs.addSubscriber(wrosen);
        simpsons.addSubscriber(msimps);
        sopranos.addSubscriber(jsopra);
        xfiles.addSubscriber(smulde);
        xfiles.addSubscriber(wrosen);

```

Figure 4-8 shows the code snippet for TestDvdObserver Class

First of all, we need to create the subjects which are the categories of the released DVDs. We basically need to create the object of type DvdReleaseCategory which is the subject class. Then we initialize the category name of the DVD release.

After that we create the subscribers of the DVD using the class DvdSubsriber which will be the concrete objects of the Observer component which is DvdSubscriber in our case. Moving on, we add subscribers to each category that they belong to.



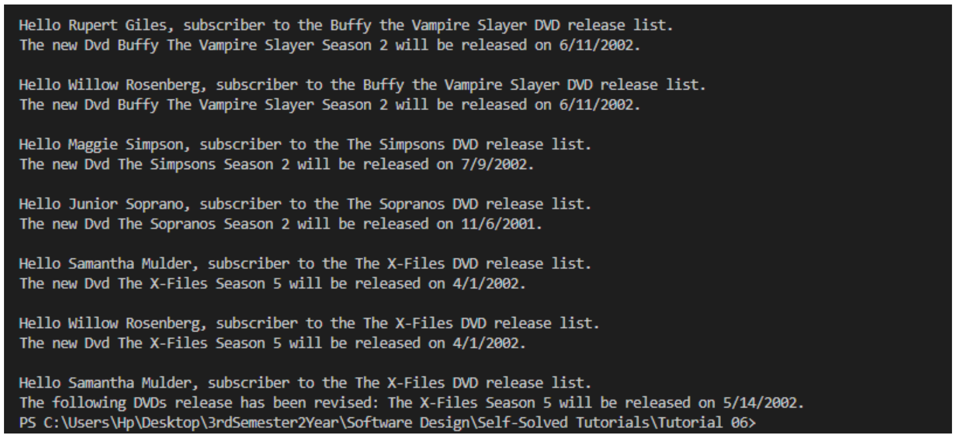
```
        DvdRelease btvsS2 = new DvdRelease(serialNumber: "DVDFOXBTVSS20",
                        dvdName: "Buffy The Vampire Slayer Season 2",
                        dvdReleaseYear: 2002, dvdReleaseMonth: 06, dvdReleaseDay: 11);
        DvdRelease simpS2 = new DvdRelease(serialNumber: "DVDFOXSIMPSO2",
                        dvdName: "The Simpsons Season 2",
                        dvdReleaseYear: 2002, dvdReleaseMonth: 07, dvdReleaseDay: 9);
        DvdRelease soprS2 = new DvdRelease(serialNumber: "DVDHBOSOPRAS2",
                        dvdName: "The Sopranos Season 2",
                        dvdReleaseYear: 2001, dvdReleaseMonth: 11, dvdReleaseDay: 6);
        DvdRelease xfilS5 = new DvdRelease(serialNumber: "DVDFOXXFILES5",
                        dvdName: "The X-Files Season 5",
                        dvdReleaseYear: 2002, dvdReleaseMonth: 04, dvdReleaseDay: 1);

        btvs.newDvdRelease(btvsS2);
        simpsons.newDvdRelease(simpS2);
        sopranos.newDvdRelease(soprS2);
        xfiles.newDvdRelease(xfilS5);

        xfiles.removeSubscriber(wrosen);

        xfilS5.updateDvdReleaseDate(dvdReleaseYear: 2002, dvdReleaseMonth: 5, dvdReleaseDay: 14);
        xfiles.updateDvd(xfilS5);
    }
}
```

Figure 4-9 shows the code snippet of TestDvdObserver Class

Next, we create DvdRelease objects. We will input all the details of the DVD release as shown in the figure above.

Finally, we can now add the new DVD releases to their respective categories. We can also perform removeSubscriber, updateDvdReleaseDate methods to manipulate the subscribers and DVD release details in the DVD categories class.

Finally, we can run the program and see the outputs:



```
Hello Rupert Giles, subscriber to the Buffy the Vampire Slayer DVD release list.
The new Dvd Buffy The Vampire Slayer Season 2 will be released on 6/11/2002.

Hello Willow Rosenberg, subscriber to the Buffy the Vampire Slayer DVD release list.
The new Dvd Buffy The Vampire Slayer Season 2 will be released on 6/11/2002.

Hello Maggie Simpson, subscriber to the The Simpsons DVD release list.
The new Dvd The Simpsons Season 2 will be released on 7/9/2002.

Hello Junior Soprano, subscriber to the The Sopranos DVD release list.
The new Dvd The Sopranos Season 2 will be released on 11/6/2001.

Hello Samantha Mulder, subscriber to the The X-Files DVD release list.
The new Dvd The X-Files Season 5 will be released on 4/1/2002.

Hello Willow Rosenberg, subscriber to the The X-Files DVD release list.
The new Dvd The X-Files Season 5 will be released on 4/1/2002.

Hello Samantha Mulder, subscriber to the The X-Files DVD release list.
The following DVDs release has been revised: The X-Files Season 5 will be released on 5/14/2002.
PS C:\Users\Hp\Desktop\3rdSemester2Year\Software Design\Self-Solved Tutorials\Tutorial 06>
```

Figure 4-10 shows the output after executing TestDvdObserver Class

Finally, let’s identify the advantages and disadvantages of Observer pattern based on Tutorial 6 Section A program.

Advantages**:**

1. **Extensibility:** The Observer design pattern allows new DVD Subscribers (Observers) to be added to the DVD Release Program effortlessly without the need for modifying the **DvdReleaseByCategory** (Subject). As a result, this will make it easy to add new enhancements to the program, for example, playback controls, audio options and so on.
2. **Performance:** The Observer design pattern tends to improve the DVD Release Program performance. This is because it only notifies the DvdSubscribers (Observers) who experience state changes. Therefore, this reduces the load of work of the DvdReleaseByCategory (Subject).

Disadvantages**:**

1. **Complexity:** Using the Design Pattern in a DVD release program may lead to the program in some cases. For example, the DvdReleaseByCategory (Subject) needs to maintain the list of subscribers (Observers) and notify in case of any state changes. Hence, if there are a significant number of subscribers, there will be also numerous state changes, therefore, this may lead to complexity in the program
2. **Large number of subscribers may lead to slow performance of program:** if the number of subscribers is large notifying them will be time-consuming. Also, if the DvdSubscriber (Observer) performs many operations when receiving the notifications, it results in an overhead to the DVD program performance.

4.2 Question 2

Firstly, in the visitor pattern, there will be a visitor class that manipulates the algorithm of a class with each execution. This implies that an algorithm may be executed in various ways by applying different visitor classes. The element object will have to approve the visitor object to allow the visitor to handle the operations on it.

Next, lets understand the components in this design pattern:

- **Visitor:** this is basically the interface / abstract class that will give the definition of the operations that can be executed on the elements of the object
- **Concrete visitor:** this is a class that applies the implementation of the visitor interface and executes the necessary operations on the object elements
- **Element:** this is an interface / abstract class that identifies the accept method which basically accepts the visitor

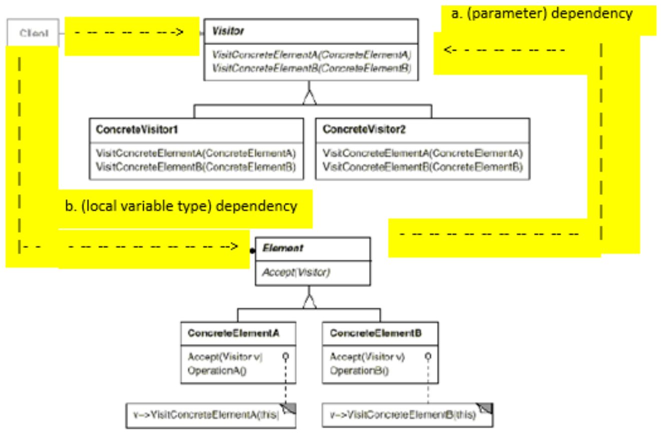
- **Object structure:** this a class that stores several elements and it also provides a method that allow the visitors to visit its elements

Moving on, since we have understood the components of the Visitor Design Pattern, now let's map the components according to Tutorial 6 – Section B question:

	Name in DP	Actual Name
1.	Element/AbstractVisitee	***AbstractTitleInfo***
	accept(Visitor)	***accept(TitleBlurbVisitor)***
2.	ConcreteElementA, B, C	**BookInfo, DVDInfo, GameInfo**
	operationA()	**setAuthor(String), setISBN(String)**
	operationB()	**setStar(String),** **setEncodingRegion(char)**
	operationC()	**setTitleName(String)**
3.	Visitor/AbstractVisitor	***TitleBlurbVisitor***
4.	ConcreteVisitor	**TitleLongBlurbVisitor,** **TitleShortBlurbVisitor**
	visitConcreteElement(ConcreteElementA;B;C)	**visit(BookInfo), visit(DVDInfo),** **visit(GameInfo)**
5.	Client	**TestTitleVisitor**

Table 2 shows the mapping table for Tutorial 6 Section B

Now, let's look first at a general structure of a Visitor component:



a. (parameter) dependency
b. (local variable type) dependency
Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
ConcreteVisitor2
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Figure 4-11 shows the general UML class diagram for Tutorial 6 Section B

Next, lets draw a UML class diagram by applying the above structure according to **Tutorial 6 Section B question**

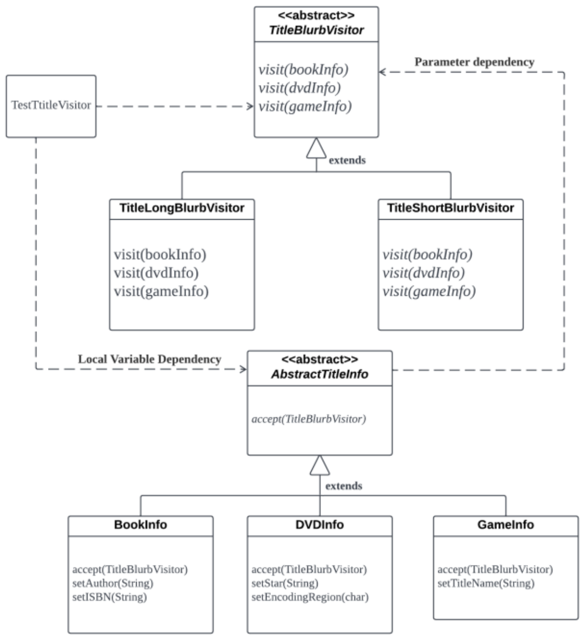
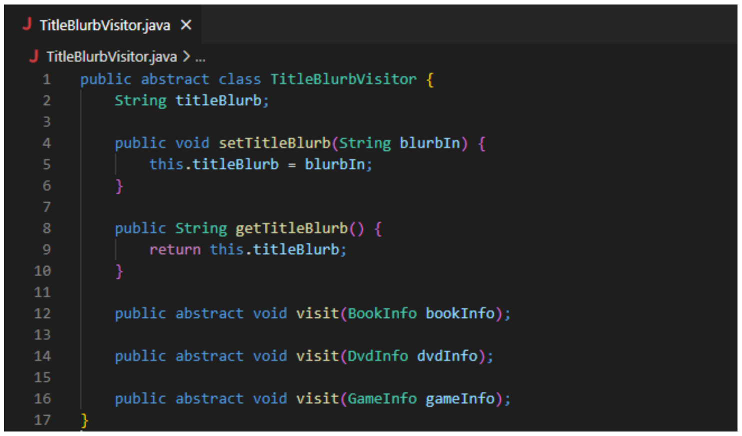


Figure 4-12 shows the UML class diagram for Tutorial 6 Section B

Moreover, as we now have a clearer idea about the Visitor Pattern and its structure, we can now look and understand the execution of the program in **Tutorial 6 Section B** which is a Blurb and Title Info Program.



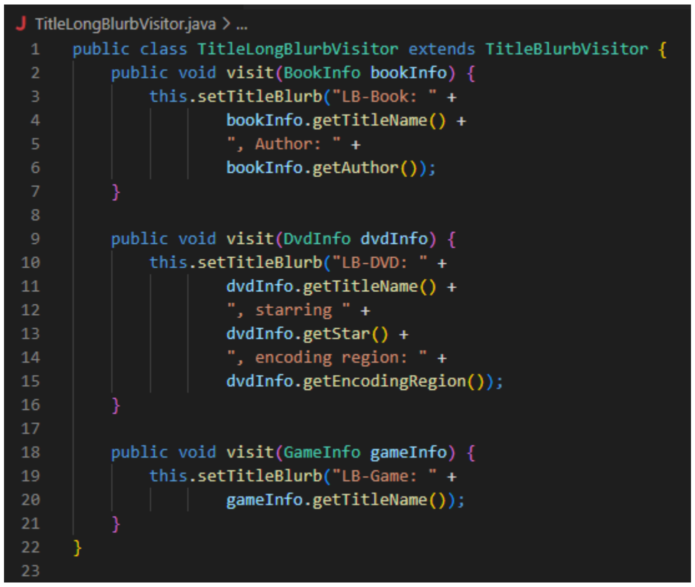
TitleBlurbVisitor.java

```java
public abstract class TitleBlurbVisitor {
    String titleBlurb;

    public void setTitleBlurb(String blurbIn) {
        this.titleBlurb = blurbIn;
    }

    public String getTitleBlurb() {
        return this.titleBlurb;
    }

    public abstract void visit(BookInfo bookInfo);

    public abstract void visit(DvdInfo dvdInfo);

    public abstract void visit(GameInfo gameInfo);
}
```

Figure 4-13 shows the code snippet of TitleBlurb Visitor Class

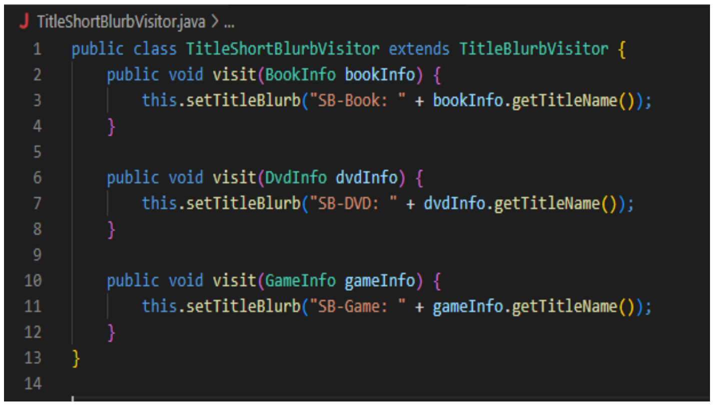
First, the visitor component will be used as an abstract class. This class will define several operations to be executed on the elements object. Here we define 3 visit methods, which are: visit(bookInfo), visit(dvdInfo) and visit(gameInfo). These 3 methods are called visit methods. They are abstract because in each concrete visitor, the algorithm will be executed in a different behavior.

Moving on, now we can look at the Concrete Visitors that will implement those Abstract class methods. In our case study, TitleLongBlurbVisitor and TitleShortBlurbVisitor will implement the abstract methods defined in the TitleBlurbVisitor.



```java
TitleLongBlurbVisitor.java > ...
public class TitleLongBlurbVisitor extends TitleBlurbVisitor {
    public void visit(BookInfo bookInfo) {
        this.setTitleBlurb("LB-Book: " +
                bookInfo.getTitleName() +
                ", Author: " +
                bookInfo.getAuthor());
    }

    public void visit(DvdInfo dvdInfo) {
        this.setTitleBlurb("LB-DVD: " +
                dvdInfo.getTitleName() +
                ", starring " +
                dvdInfo.getStar() +
                ", encoding region: " +
                dvdInfo.getEncodingRegion());
    }

    public void visit(GameInfo gameInfo) {
        this.setTitleBlurb("LB-Game: " +
                gameInfo.getTitleName());
    }
}

```

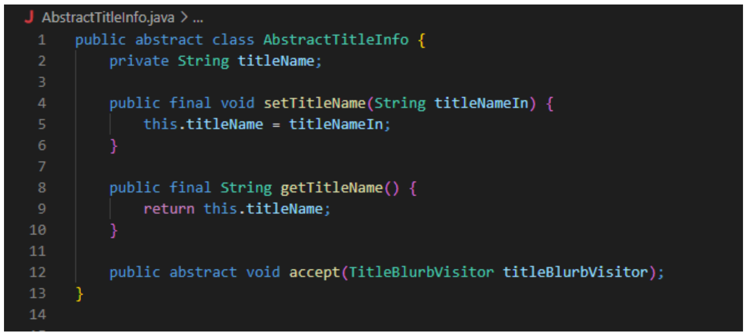
Figure 4-14 shows the code snippet for TitleLongBlurbVisitor Class



```java
TitleShortBlurbVisitor.java > ...
public class TitleShortBlurbVisitor extends TitleBlurbVisitor {
    public void visit(BookInfo bookInfo) {
        this.setTitleBlurb("SB-Book: " + bookInfo.getTitleName());
    }

    public void visit(DvdInfo dvdInfo) {
        this.setTitleBlurb("SB-DVD: " + dvdInfo.getTitleName());
    }

    public void visit(GameInfo gameInfo) {
        this.setTitleBlurb("SB-Game: " + gameInfo.getTitleName());
    }
}

```

Figure 4-15 shows the code snippet for TitleShortBlurbVisitor Class

Next, the AbstractTitleInfo (Element) will define an operation called accept. This operation will take the TitleBlurbVisitor (Visitor) as an argument for it and will give the green light to access the element and execute the operations. This will be implemented by its Concrete Classes which will represent the objects that the visitor will operate. According to our case study, these classes will be BookInfo, DVDInfo, GameInfo.

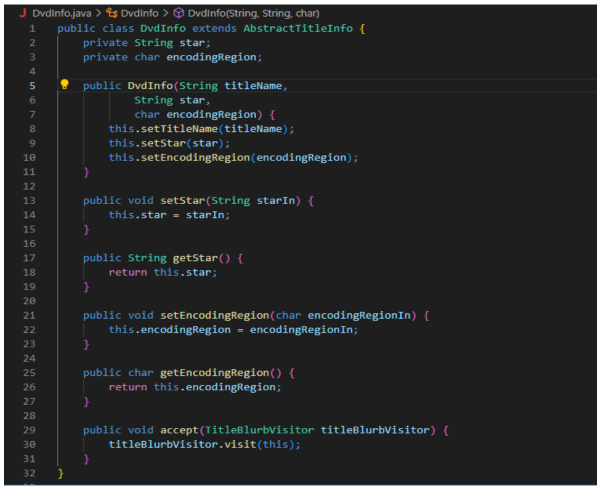


AbstractTitleInfo.java > ...

```java
public abstract class AbstractTitleInfo {
    private String titleName;

    public final void setTitleName(String titleNameIn) {
        this.titleName = titleNameIn;
    }

    public final String getTitleName() {
        return this.titleName;
    }

    public abstract void accept(TitleBlurbVisitor titleBlurbVisitor);
}
```

Figure 4-16 shows the code snippet for AbstractTitleInfo Class

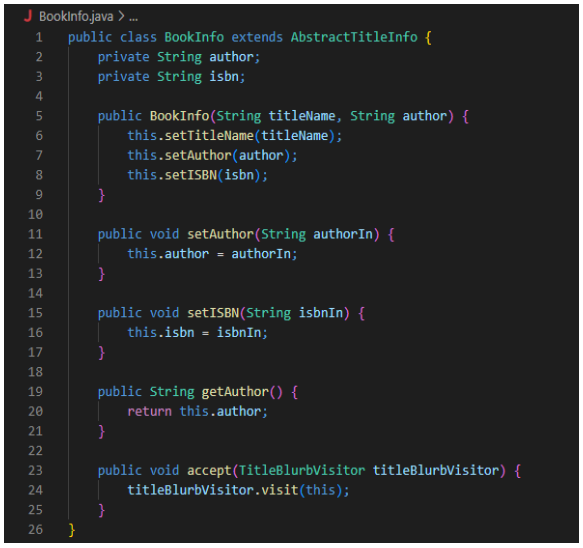
Next the Concrete Classes of the AbstractTitleInfo that are listed above will implement the accept method:



DvdInfo.java > DvdInfo > DvdInfo(String, String, char)

```java
public class DvdInfo extends AbstractTitleInfo {
    private String star;
    private char encodingRegion;

    public DvdInfo(String titleName,
            String star,
            char encodingRegion) {
        this.setTitleName(titleName);
        this.setStar(star);
        this.setEncodingRegion(encodingRegion);
    }

    public void setStar(String starIn) {
        this.star = starIn;
    }

    public String getStar() {
        return this.star;
    }

    public void setEncodingRegion(char encodingRegionIn) {
        this.encodingRegion = encodingRegionIn;
    }

    public char getEncodingRegion() {
        return this.encodingRegion;
    }

    public void accept(TitleBlurbVisitor titleBlurbVisitor) {
        titleBlurbVisitor.visit(this);
    }
}
```

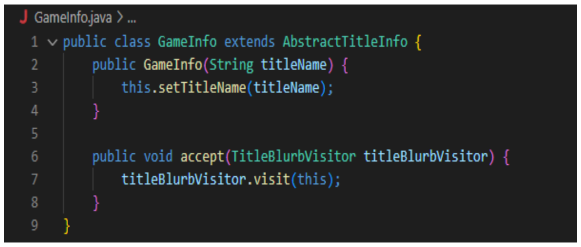
Figure 4-17 shows the code snippet of DvdInfo Class



BookInfo.java > ...

```java
public class BookInfo extends AbstractTitleInfo {
    private String author;
    private String isbn;

    public BookInfo(String titleName, String author) {
        this.setTitleName(titleName);
        this.setAuthor(author);
        this.setISBN(isbn);
    }

    public void setAuthor(String authorIn) {
        this.author = authorIn;
    }

    public void setISBN(String isbnIn) {
        this.isbn = isbnIn;
    }

    public String getAuthor() {
        return this.author;
    }

    public void accept(TitleBlurbVisitor titleBlurbVisitor) {
        titleBlurbVisitor.visit(this);
    }
}
```

Figure 4-18 shows the code snippet for BookInfo Class

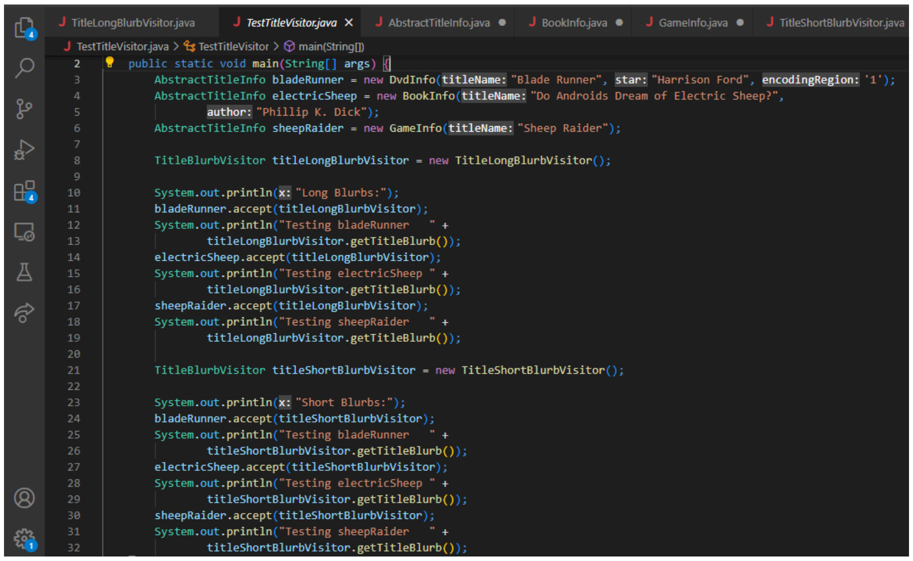


GameInfo.java > ...

```java
public class GameInfo extends AbstractTitleInfo {
    public GameInfo(String titleName) {
        this.setTitleName(titleName);
    }

    public void accept(TitleBlurbVisitor titleBlurbVisitor) {
        titleBlurbVisitor.visit(this);
    }
}
```

Figure 4-19 shows the code snippet of GameInfo Class

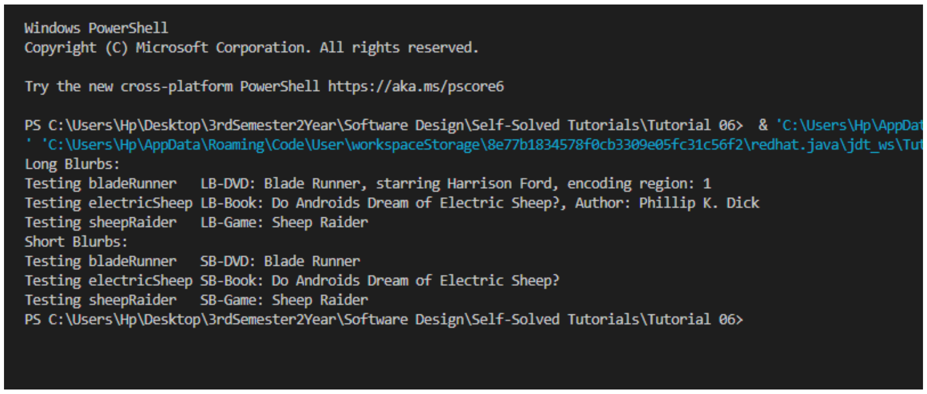
Finally, now we can run the Client Code which according to our case study will be TestTitleVisitor. Here, we would create the TitleBlurbVisitor objects, and we would pass to it the TitleInfo such as (BookInfo or GameInfo) we would like it to operate on. The client would simply perform the operations on TitleInfo without the need to get to know the details of how the operations are executed.


TitleLongBlurbVisitor.java
TestTitleVisitor.java
AbstractTitleInfo.java
BookInfo.java
GameInfo.java
TitleShortBlurbVisitor.java
TestTitleVisitor.java > TestTitleVisitor > main(String[])

```java
    public static void main(String[] args) {
        AbstractTitleInfo bladeRunner = new DvdInfo(titleName: "Blade Runner", star: "Harrison Ford", encodingRegion: '1');
        AbstractTitleInfo electricSheep = new BookInfo(titleName: "Do Androids Dream of Electric Sheep?",
                author: "Phillip K. Dick");
        AbstractTitleInfo sheepRaider = new GameInfo(titleName: "Sheep Raider");

        TitleBlurbVisitor titleLongBlurbVisitor = new TitleLongBlurbVisitor();

        System.out.println(x: "Long Blurbs:");
        bladeRunner.accept(titleLongBlurbVisitor);
        System.out.println("Testing bladeRunner   " +
                titleLongBlurbVisitor.getTitleBlurb());
        electricSheep.accept(titleLongBlurbVisitor);
        System.out.println("Testing electricSheep " +
                titleLongBlurbVisitor.getTitleBlurb());
        sheepRaider.accept(titleLongBlurbVisitor);
        System.out.println("Testing sheepRaider   " +
                titleLongBlurbVisitor.getTitleBlurb());

        TitleBlurbVisitor titleShortBlurbVisitor = new TitleShortBlurbVisitor();

        System.out.println(x: "Short Blurbs:");
        bladeRunner.accept(titleShortBlurbVisitor);
        System.out.println("Testing bladeRunner   " +
                titleShortBlurbVisitor.getTitleBlurb());
        electricSheep.accept(titleShortBlurbVisitor);
        System.out.println("Testing electricSheep " +
                titleShortBlurbVisitor.getTitleBlurb());
        sheepRaider.accept(titleShortBlurbVisitor);
        System.out.println("Testing sheepRaider   " +
                titleShortBlurbVisitor.getTitleBlurb());
```

Figure 4-20 shows the code snippet for TestTitleVisitor Class

Lastly, we can now run the TestTitleVisitor to see the output:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Hp\Desktop\3rdSemester2Year\Software Design\Self-Solved Tutorials\Tutorial 06>  & 'C:\Users\Hp\AppDat
' 'C:\Users\Hp\AppData\Roaming\Code\User\workspaceStorage\8e77b1834578f0cb3309e05fc31c56f2\redhat.java\jdt_ws\Tut
Long Blurbs:
Testing bladeRunner   LB-DVD: Blade Runner, starring Harrison Ford, encoding region: 1
Testing electricSheep LB-Book: Do Androids Dream of Electric Sheep?, Author: Phillip K. Dick
Testing sheepRaider   LB-Game: Sheep Raider
Short Blurbs:
Testing bladeRunner   SB-DVD: Blade Runner
Testing electricSheep SB-Book: Do Androids Dream of Electric Sheep?
Testing sheepRaider   SB-Game: Sheep Raider
PS C:\Users\Hp\Desktop\3rdSemester2Year\Software Design\Self-Solved Tutorials\Tutorial 06>
```

Figure 4-21 shows the code snippet of TestTitleVisitor Class

Moreover, after the program execution we can now analyze the program to identify the advantages and disadvantages of using Visitor Design Pattern in the case study of Tutorial 6 Section B:

Advantages:

- **Ease of adding new features / extending the Blurb & TitleInfo program:** This means that we would be able to add new operations under, for example, DVD Information without modifying the implementation of the underlying objects. An example may include adding new feature to display additional information about a DVD title
- **Easy to maintain the Blurb & TitleInfo program:** As visitor design pattern usually promotes the idea of separating the implementation of operations from the object structure, this helps to achieve maintainability of the program. This means that we can reduce the risk of possible bugs appearing while accessing DVD or game information. In the worst scenario, the bug can be easily identified as the codes of implementation are separate from the object structure. Hence, easy to maintain and faster to fix bugs.

Disadvantages:

- **Rigid program:** Any change in the Blurb & TitleInfo program hierarchy will require modifying the TitleBlurbVisitor and the implementation of the accept methods in all of GameInfo, DvdInfo and BookInfo. This tends to be a lot of work to do by developers if the project is huge.
- **Complex:** There are separate visitor classes that implement TitleBlurbVisitor class. Also, the accept method needs to be implemented whenever we add a new visitor class. For future TitleBlurbVisitor enhancements, the code might be complex as there will be several additional visitor classes such as BlurayInfo. The more visitor classes added, the more accept methods added and more complex it gets.

4.3 Question 3

First of all, the command design pattern transforms a request into a separate object that stores all the necessary information concerning the request. As a result, the client can pass requests in terms of method arguments.

Moreover, let's take a closer look and understand the different components of command design pattern:

- **Command:** This is an interface that defines a method to execute requests
- **Concrete Command:** This is a class performs that the implementation of the command interface. It also provides the definition of the binding connecting the receiver and action
- **Client:** This is an object that defines a command and sets its receiver
- **Invoker:** This is an object that tell the command component to execute the request
- **Receiver:** This is an object that understands on how to carry out and execute the request

Moving on, since we have understood the components of the Command Design Pattern, now let's map the components according to **Tutorial 6 – Section D question**:

	Name in DP	Actual Name
1.	Command	***CommandAbstract****, abstract class*
	execute()	***execute()***

2.	ConcreteCommand	**DvdCommandNameStateOn**, **DvdCommandNameStateOff**
	receiverAggregationVariable	**dvdName**
3.	Receiver	**DvdName**
	action()	**setNameStarsOn(), setNameStarsOff()**
4.	Client	**TestCommand**

Table 3 shows the mapping table for Tutorial 6 Section D

Now, since we understood the components of the Command Pattern, let's look first at the general structure of the UML Class Diagram:

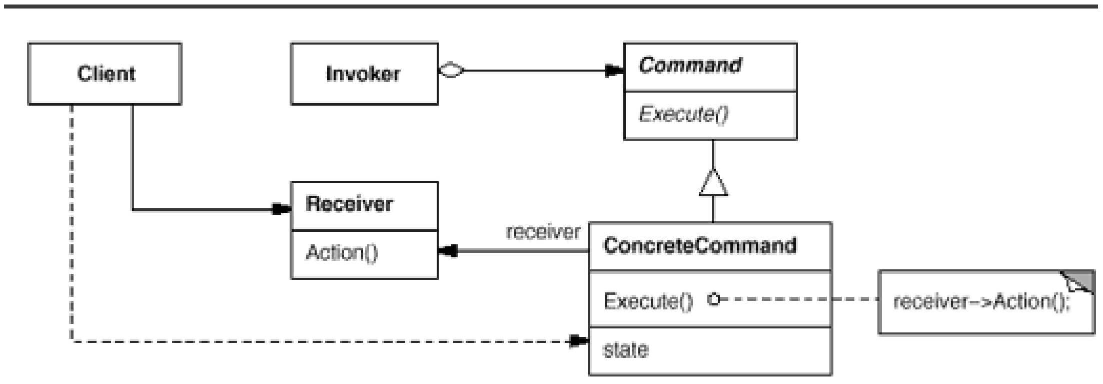


Figure 4-22 shows the general UML class diagram for Tutorial 6 Section D (Caballero, 2019)

Moving on further, let's apply the above class diagram to our Tutorial 6 Section D case study:

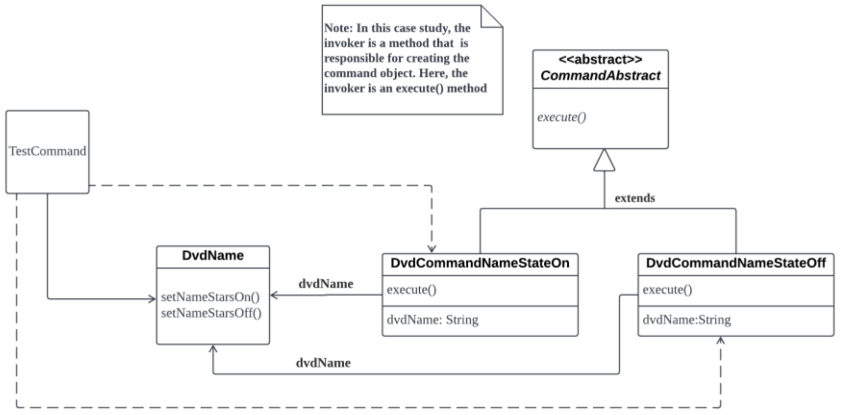
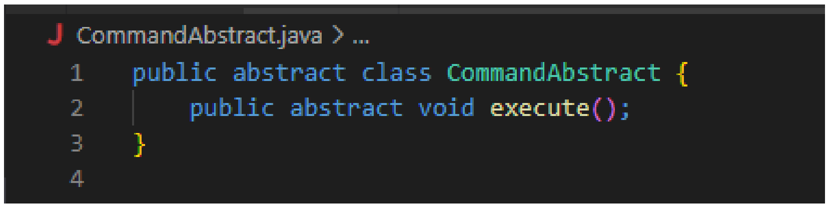


Figure 4-23 shows the UML class diagram for Tutorial 6 Section D

Furthermore, as we now have a better idea about the Command Pattern and its structure, we can now look and understand the execution of the program in **Tutorial 6 Section D** which is a DvdStarsOnOff Program

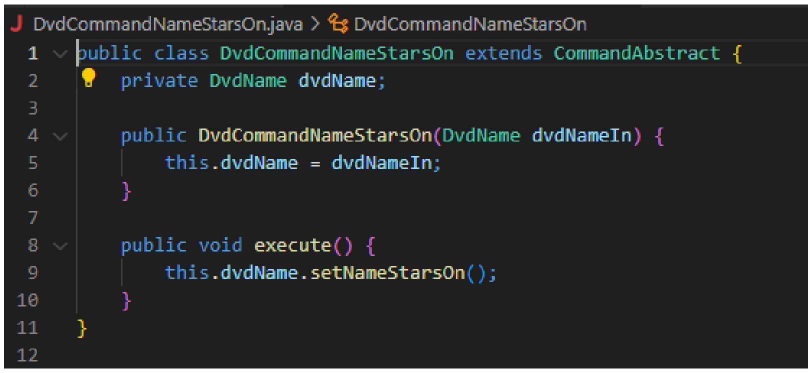
Firstly, the CommandAbstract (Command) class will declare a method for executing request. This method will be implemented by DvdCommandNameStateOn and DvdCommandNameStateOff (Concrete Commands) which will establish a connection between the DvdName (receiver) and the execute action.



```java
public abstract class CommandAbstract {
    public abstract void execute();
}

```

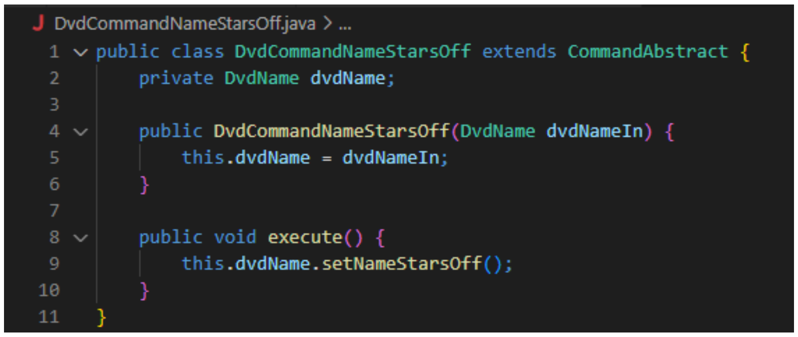
Figure 4-24 shows the code snippet for CommandAbstract Class

Then, the DvdCommandNameStateOn and DvdCommandNameStateOff will implement the execute method:


DvdCommandNameStarsOn.java > DvdCommandNameStarsOn

```java
public class DvdCommandNameStarsOn extends CommandAbstract {
    private DvdName dvdName;

    public DvdCommandNameStarsOn(DvdName dvdNameIn) {
        this.dvdName = dvdNameIn;
    }

    public void execute() {
        this.dvdName.setNameStarsOn();
    }
}
```

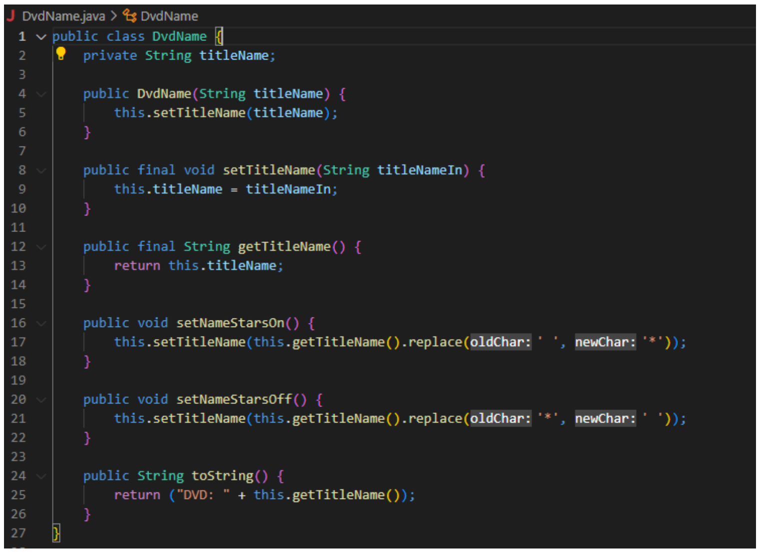
Figure 4-25 shows the code snippet of DvdCommandNameStarsOn Class


DvdCommandNameStarsOff.java > ...

```java
public class DvdCommandNameStarsOff extends CommandAbstract {
    private DvdName dvdName;

    public DvdCommandNameStarsOff(DvdName dvdNameIn) {
        this.dvdName = dvdNameIn;
    }

    public void execute() {
        this.dvdName.setNameStarsOff();
    }
}
```

Figure 4-26 shows the code snippet of DvdCommandNameStarsOff Class

After that, the DvdName (receiver) has the knowledge on how to execute the request. It has all the information and resources required to execute the operations that was defined by the request:

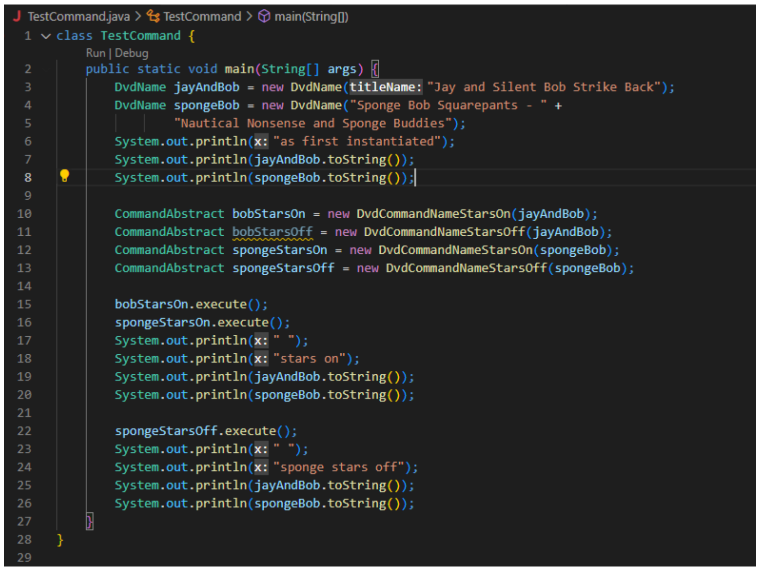


DvdName.java > DvdName

```java
public class DvdName {
    private String titleName;

    public DvdName(String titleName) {
        this.setTitleName(titleName);
    }

    public final void setTitleName(String titleNameIn) {
        this.titleName = titleNameIn;
    }

    public final String getTitleName() {
        return this.titleName;
    }

    public void setNameStarsOn() {
        this.setTitleName(this.getTitleName().replace(oldChar: ' ', newChar: '*'));
    }

    public void setNameStarsOff() {
        this.setTitleName(this.getTitleName().replace(oldChar: '*', newChar: ' '));
    }

    public String toString() {
        return ("DVD: " + this.getTitleName());
    }
}
```

Figure 4-27 shows the code snippet of DvdName Class

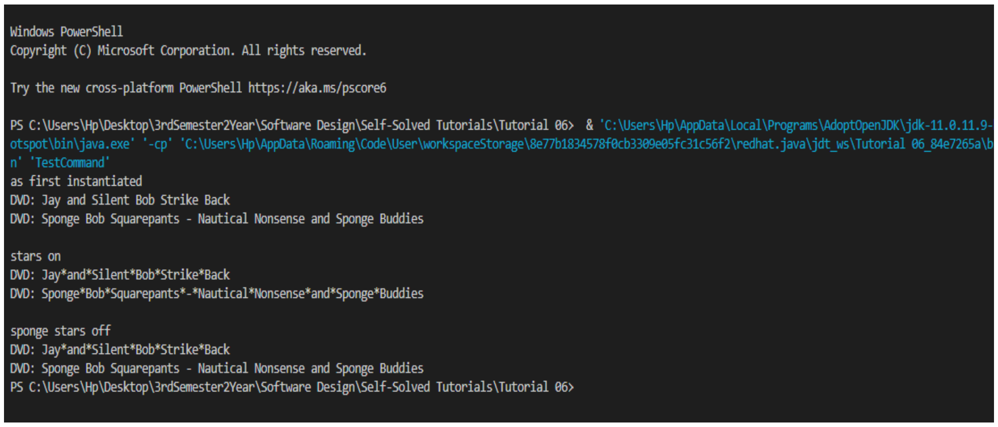
Finally, the TestCommand (client) is basically an object that creates the command and will set the receiver. Here we will specify the request to be executed and the object that will execute the request. In this class, we do not need to know the details of the request that is executed because all these details are encapsulated in the DvdCommandNameStarOn and DvdCommandNameStarOff (command) objects.



```java
TestCommand.java > TestCommand > main(String[])
class TestCommand {
    Run | Debug
    public static void main(String[] args) {
        DvdName jayAndBob = new DvdName(titleName: "Jay and Silent Bob Strike Back");
        DvdName spongeBob = new DvdName("Sponge Bob Squarepants - " +
                "Nautical Nonsense and Sponge Buddies");
        System.out.println(x: "as first instantiated");
        System.out.println(jayAndBob.toString());
        System.out.println(spongeBob.toString());

        CommandAbstract bobStarsOn = new DvdCommandNameStarsOn(jayAndBob);
        CommandAbstract bobStarsOff = new DvdCommandNameStarsOff(jayAndBob);
        CommandAbstract spongeStarsOn = new DvdCommandNameStarsOn(spongeBob);
        CommandAbstract spongeStarsOff = new DvdCommandNameStarsOff(spongeBob);

        bobStarsOn.execute();
        spongeStarsOn.execute();
        System.out.println(x: " ");
        System.out.println(x: "stars on");
        System.out.println(jayAndBob.toString());
        System.out.println(spongeBob.toString());

        spongeStarsOff.execute();
        System.out.println(x: " ");
        System.out.println(x: "sponge stars off");
        System.out.println(jayAndBob.toString());
        System.out.println(spongeBob.toString());
    }
}
```

Figure 4-28 shows the code snippet of TestCommand Class

Last but not least, we can run the code of TestCommand (client) and see the output:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Hp\Desktop\3rdSemester2Year\Software Design\Self-Solved Tutorials\Tutorial 06>  & 'C:\Users\Hp\AppData\Local\Programs\AdoptOpenJDK\jdk-11.0.11.9-
otspot\bin\java.exe' '-cp' 'C:\Users\Hp\AppData\Roaming\Code\User\workspaceStorage\8e77b1834578f0cb3309e05fc31c56f2\redhat.java\jdt_ws\Tutorial 06_84e7265a\b
n' 'TestCommand'
as first instantiated
DVD: Jay and Silent Bob Strike Back
DVD: Sponge Bob Squarepants - Nautical Nonsense and Sponge Buddies

stars on
DVD: Jay*and*Silent*Bob*Strike*Back
DVD: Sponge*Bob*Squarepants*-*Nautical*Nonsense*and*Sponge*Buddies

sponge stars off
DVD: Jay*and*Silent*Bob*Strike*Back
DVD: Sponge Bob Squarepants - Nautical Nonsense and Sponge Buddies
PS C:\Users\Hp\Desktop\3rdSemester2Year\Software Design\Self-Solved Tutorials\Tutorial 06>
```

Figure 4-29 shows the output of TestCommand Class

Moreover, after the program execution we can now analyze the program to identify the advantages and disadvantages of using Command Design Pattern in the case study of Tutorial 6 Section D:

Advantages:

- **Decoupling objects:** The Command Design pattern decouples the TestCommand (Client) object that actually executes the operation from the DvdName (receiver) which is the object that performs the operation. Hence, this makes it easy to change or extend DvdStarsOnOff program
- **Encapsulation:** All the details of the request that is executed are encapsulated in the DvdCommandNameStarOn and DvdCommandNameStarOff (command) objects. Therefore, this makes it easy to execute requests for the client.

Disadvantages:

- **Too many classes possible:** There might be an increase in number of classes needed for each command. For example, DvdCommandNameStarEdit, and so on. As a result, with future enhancements to DvdStarsOnOff program, there might be too many classes which will make it difficult to maintain the program.
- **Client might not understand the program:** The client might find difficulty in understanding DvdStarsOnOff program due to the high indirection level of connection between the TestCommand and the DvdName due to encapsulation of many details and steps.

5. Conclusion and suggestions

5.1 Conclusion

In a nutshell, the Observer, Visitor and Command design patterns tend to be important and useful patterns that are used to solve many common problems in the software design.

To give a clear and concise explanation of these design patterns, we demonstrated the explanations with sample runs of real-life examples. Therefore, we believe that the discussion provided a practical way of understanding which will be better for the students to understand. Additionally, we followed the explanation of each question with benefits and drawbacks to make the student identify when these design patterns are appropriate to use.

Finally, we hope that we achieved the goal of this project which is providing simple, clear, and concise explanations by providing practical examples to students.

5.2 Suggestions

For future work, we suggest that we provide more complex and advanced examples of Observer, Visitor and Command design patterns for better understanding when developing a large project. Additionally, we also recommend discussing other types of behavioral design pattern such as Chain of Responsibility Pattern as this will provide a more solid foundation for the students. Finally, we look forward to providing interactive practice exercises for students such as quizzes to achieve better understanding of applying these design pattern

6. Bibliography

Eales, A. (2005, January 1). The Observer Pattern Revisited. *Wellington Institute of Technology*, 1-2.

Caballero, C. (2019). *Command Design Pattern*. Retrieved 12 22, 2022, from betterprogramming.pub: https://betterprogramming.pub/the-command-design-pattern-2313909122b5

Hussain, S., Keung, J., Sohail, M. K., Khan, A. A., Ahmad, G., Mufti, M. R., & Khatak, H. A. (2019, April 2). Methodology for the quantification of the effect of patterns and anti-patterns association on the software quality. *The Institute Of Engineering and Technology*, 1-3.

Jeanmart, S., Gu´eh´eneuc, Y.-G., Sahraoui, H., & Habra, N. (2014, October 17). Impact of the Visitor Pattern on Program Comprehension and Maintenance. *University of Montreal*, 1-3.

Liguo Yu, Yingmei Li, & Srini Ramaswamy. (2018, March 19). Design Patterns and Design Quality: Theoretical Analysis, Empirical. *nternational Journal of Secure Software Engineering*, 1.

McGahagan, J. (2013, January 1). Behavioral Designs Patterns and Agile Software Development. *University of Maryland*, 1-2.

Observer Pattern. (2022). Retrieved 12 20, 2022, from www.cs.mcgill.ca: https://www.cs.mcgill.ca/~hv/classes/CS400/01.hchen/doc/observer/observer.html

Wedyan, F., & Abufakher, S. (2019, September 9). Impact of design patterns on software quality: a systemic literature review. *The Institute of Engineering and Technology*, 1-2.