

**Assignment Report**

**Object Oriented Design Patterns**

Joseph Tierney B00092923 Lecturer: Orla McMahon

Eoghan de Bhal B00092942 COMP H3012

Object Oriented Design Patterns

Table of Contents

[Abstract 3](#_Toc510962113)

[Design of the Application 4](#_Toc510962114)

[Abstract Factory Method 5](#_Toc510962115)

[Command Pattern 8](#_Toc510962116)

[Composite Pattern 9](#_Toc510962117)

[Singleton Pattern 11](#_Toc510962118)

[Conclusion 13](#_Toc510962119)

# Abstract

This paper describes the technologies and approach that can be used to create and maintain a film application using four object-oriented design patterns. The patterns used in this assignment are; the abstract factory method, the command pattern, the composite pattern and the singleton pattern. The paper will discuss the design of the application, going into detail about why certain styles and layouts were used. It will also go into detail about how the application works, with the major focuses being on how the design patterns that were chosen were applied to this application. The paper will also illustrate the use of the design patterns with the aid of a UML diagram.

# Design of the Application

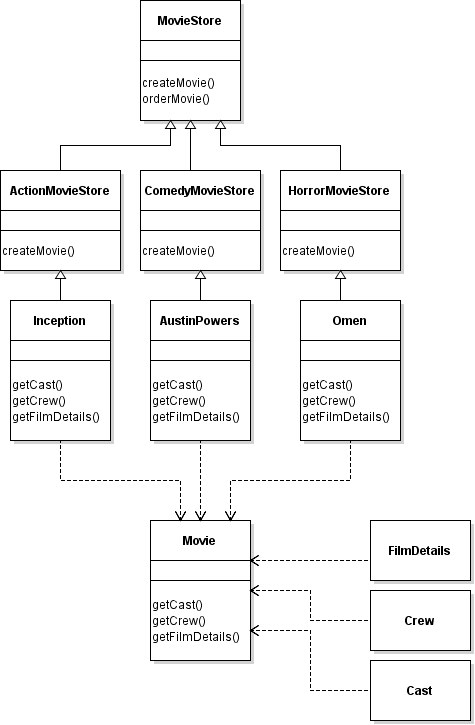
When designing the application, we kept the user in mind with one simple question, is everything clear and easy to use? We accomplished this by making use of JFrames in Java. The application includes various features, such as; buttons, labels, a menu bar, text fields, images and a list. The simplistic nature of this design makes the application easy to navigate and understand what it is doing.

To start off, when running the application, the user is greeted with a popup message welcoming them to the application – this makes use of the singleton pattern. Once the user clicks OK, the Graphical User Interface (GUI) will be displayed. At the top of the GUI, there are two menu items; “File” and “Help”. When the user clicks “File”, a sub-menu titled “Exit” is shown. Clicking on “Exit” will close the application. Clicking “Help” will show a sub-menu titled “About”. Clicking “About” will show a pop-up message showing the user who created the application. At the bottom right of the application there is a “Quit” button, clicking this will also close the application. These features show use of the command pattern.

Following on from this, on the main panel of the application, the user will see an empty form-styled area with film details waiting to be put there. These will remain blank until the user clicks on the left of the panel, where they will see a JTree list. The films in this application are sorted into three categories; Comedy, Action and Horror. Clicking to the left of these categories will expand each of them, revealing a different list of movies. Clicking on any movie will fill the form to the right, displaying the movies poster, some photos from the move, the title, director, release date, runtime, budget, box office, writers, cast and a synopsis of the movie, making use of the abstract factory method.

The text fields in the application have been disabled, this was a design choice made by the both of us early on into production of the application, as we felt it gives it a more professional feel by not allowing users from deleting movie information.

# Abstract Factory Method

****

The Abstract factory method was used to create the movie objects needed for the program. An abstract class named Movie was created with the methods getCast(), getCrew() and getFilmDetails(). These methods are used to return objects of Cast, Crew and FilmDetails based on their respective classes.

An abstract class named MovieStore was created to implement methods createMovie() and orderMovie(). These methods are used to create the completed movie objects comprising of the cast, crew and film details.

The MovieStore abstract class is used to create movies based on their genre. An example of this is the ActionMovieStore. As ActionMovieStore extends MovieStore it uses the createMovie() method to return a movie object based on the movie title. This is done by passing the title of the movie as a string and then checking the title of the movie. If the title of the movie is “Inception” a new Inception movie object is created. The case of the string is ignored to remove any potential issues with the spelling of the title.

In this case of the movie Inception being created, the Inception class is called. This class extends the Movie abstract class so it implements the same methods getCast(), getCrew and getFilmDetails(). These methods are used to create these objects with the correct data held within them.

During the running of this application, each movie is created once the value withing the JTree list has changed. The valueChanged() method in the FilmAppGUI class stores the string contained in the parent path and the child path. The parent path is then checked to see whether it contains “comedy”, “action” or “horror”. Once the parent genre type has been identified a Movie store based on the genre is created. As an example say that “action” is the parent path. A new actionStore is created and calls the orderMovie() method. The child path is the string that is passed within this method. This will then create the correct movie object (i.e Inception). Once the object is created the changeMovieFields() method changes all the fields within the GUI to correspond to that movie.

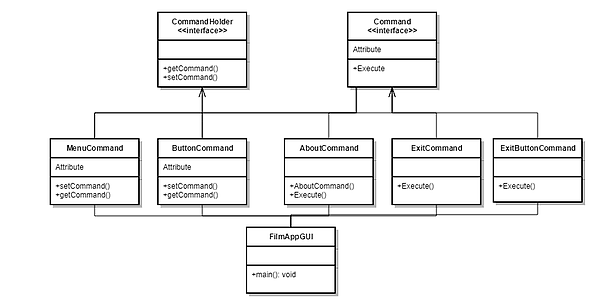


The abstract factory pattern makes creating movies very efficient as new classes can be created without having to change other classes too much. If a new genre of movie was needed for the application all that would need to be done is create a movie store of that genre and create the movie classes. As the movie classes will also be using the Cast, Crew and FilmDetails classes they do not need to be recreated each time. Using the factory pattern also helps to avoid having to create a long list of if statements to check which objects to create.

# Command Pattern

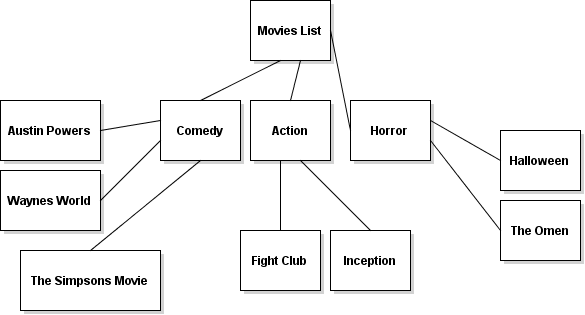
The Command pattern is used to encapsulate an object with all the information needed to perform an action or trigger an event at a later time. For our application, we created an interface called Command which is acting as a command, and we have created an interface called CommandHolder, which sets and gets the command. The purpose of the CommandHolder is so that the UI components can now be implemented through this interface, meaning that a command can now be associated with a particular UI component using the setCommand method. The getCommand method retrieves the command associated with the UI component.

The Invoker, which in this case, are the MenuCommand and the ButtonCommand classes, asks the command to carry out the request. The MenuCommand and ButtonCommand objects use the command pattern to identify which object will execute what command based on what the type of command is. These will distinguish between the whether or not the input is from the menu items or from the buttons. The concrete command classes in this application are the AboutCommand, ExitCommand and ExitButtonCommand classes – these classes implement the command interface, which itself will do the processing of the commands.



# Composite Pattern

The composite pattern is used to create a tree structure to represent hierarchies, in this case movies and their genres. By using the composite pattern, we were able to create a JTree which lets the user choose the genre of movie and the movie itself. This is accomplished by creating a composite of type Genre and then creating a movie leaf for each movie within this genre.

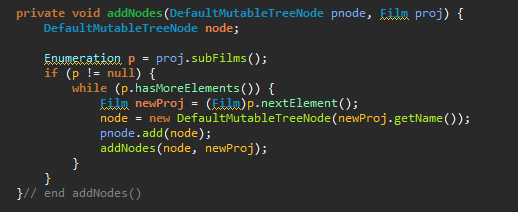


The abstract class AbstractFilm is used as a Component. It implements default behaviour for the interface to all classes. It also declares an interface for accessing and managing the child components.

The BaseFilm class is used to create a genre component. This will allow for the creation of leaves that will store the names of the movies. These will be stored in a Vector called films. The BaseFilm class is the only class that can store child leaf objects.

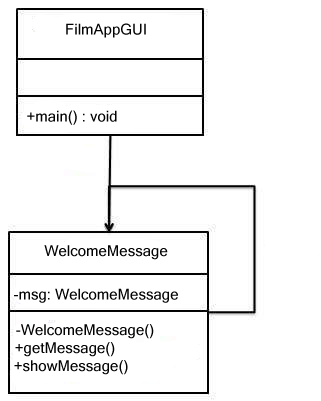
The Film class is used to create the leaf objects. These will store the names of the movies. The Film class must have concrete implementations of the add, remove and getChild methods as it is a leaf. These methods will return an error indication in the form of a string. These errors would be caused as the Film class is a leaf and cannot itself have leaves.

To create the list of movies the makeFilms() method is called during the initialization of the application. This creates the BaseFilm objects and the Film objects. The setTree() method is then called to set these objects within the JTree component. This is done by calling the loadTree()  method which calls the addNodes() method recursively until all the leaves in each node are processed.

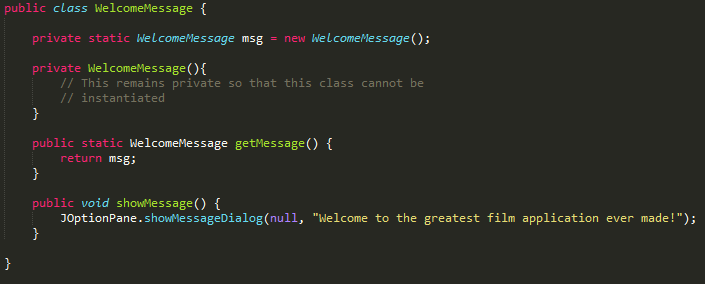


# 

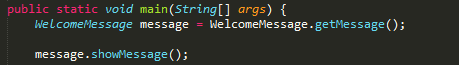
# Singleton Pattern



The singleton pattern involves a single class. This single class, in our case, WelcomeMessage, which will be responsible for creating an object while also making sure that only a single object gets created. This was achieved by creating a static variable inside of the WelcomeMessage class. A static variable is used because it is a variable in which there can only be one instance of, no matter how many other instances of the class there are. In order to prevent instantiating the class more than once, the WelcomeMessage constructor is made private, this makes it so that an instance can be created only from within a static method.



Our FilmAppGUI class uses the WelcomeMessage class to get a WelcomeMessage object.



# Conclusion

Overall, we feel as if we accomplished all the necessary tasks that had been asked of us for this assignment. If we had more time, we would like to go back and attempt to implement the Iterator pattern. We originally decided to use this pattern for scrolling through photos of the various movies, however, that task proved to be difficult, so we opted to use the Singleton pattern as a backup choice.