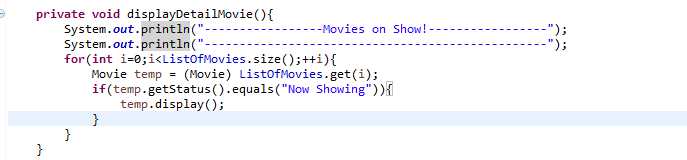
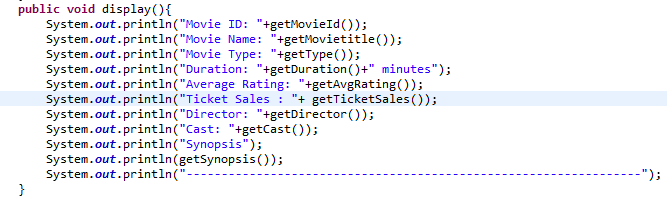
**Design Principles**

Each object is defined via independent class that interacts with well-defined interface. Our program is modularised to achieve loose coupling and high cohesion. By adhering to the design principles stated below, we can ensure that our written code is reusable, extensible and maintainable.

To illustrate loose coupling, high cohesion and good encapsulation, a segment of code used in UserLogin.Java is used as example:

­­­Figure 1.1: *displayDetailMovie() from UserLogin.java*  
Figure 1.2: *display() from Movie.Java*

**Loose Coupling**

*UserLogin.Java* does not need to implement the *display()* function in order to execute *displayDetailMovie()*. The *Movie.display()* function ( implemented via *DisplayInterface* interface) is carried out by a instance of Movie class( ie. *temp)*. This means that if *Movie.display()* is modified, the *displayDetailMovie()* will still work properly as it only needs to call the *Movie.display()* function. On the other hand, neither does the *Movie.display()* function require *displayDetailMovie()* to work.

**High Cohesion**  
 *displayDetailMovie()*  does not need to track data and manage display of the *Movie* class. This is taken care by the *Movie* class. For example, if an addition attribute(ie. Release Date) is added to the Movie class, *UserLogin.Java* does not need to be modified to print that extra attribute. Instead, *Movie.Display()* decides whether to print that attribute or not.

**Good Encapsulation**  
  
*displayDetailMovie()* does not need to know inner workings of *Movie.Display()* (ie. the displayType) since there is no required parameter to pass in. The amount of details to print is manage by *Movie. displayDetailMovie()* does not need to know the data type of detail, nor the number of details. Furthermore, in each object class, all its private attributes are only accessible with provided public get-methods.

**Solid Principle**  
**Single Responsibility Principle (SRP)**

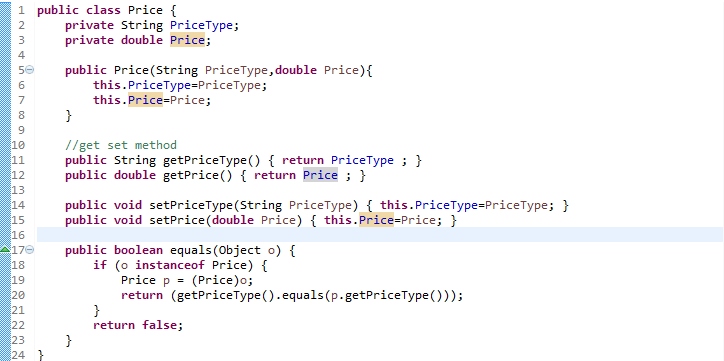
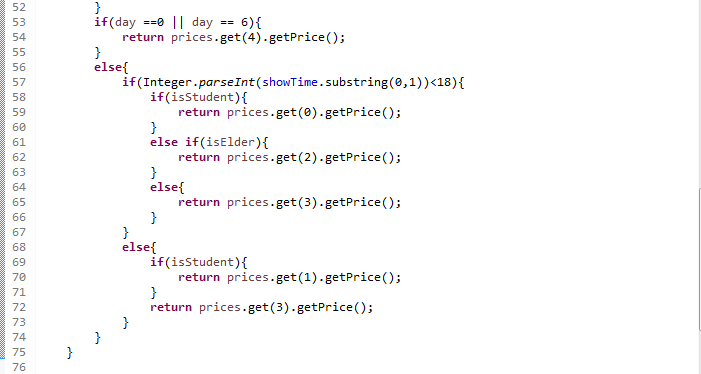
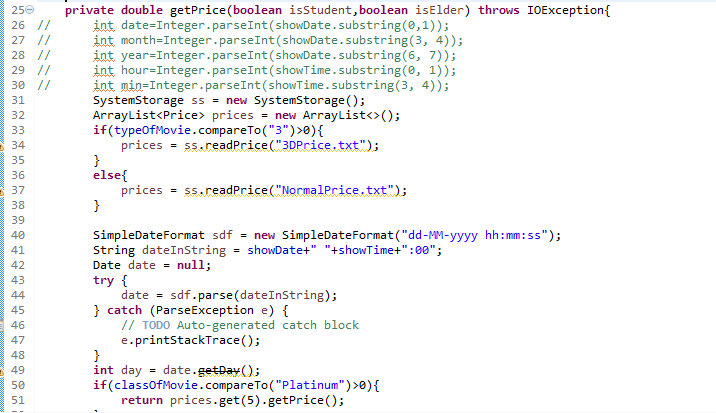
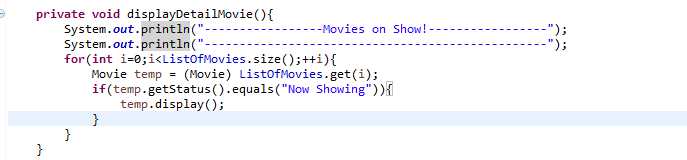


Figure 2.1: *the complete Price class*

** Figure 2.2 :  *getPrice() method in a separate class*

As illustrated in figure 2.1 and figure 2.2. Each class is only responsible for displaying its own attribute along with providing get and set functions. In the event where more complicated methods are required for changing one of its attributes (ie. calculating ticket price), a separate class is used. This limits the number of responsibilities of each class and reduces the axis of change.

**Open-Closed Principle (OCP)**

­­­Figure 3.1: *displayDetailMovie() from UserLogin*

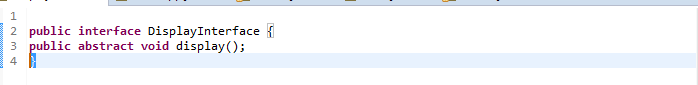
**

Figure 3.2: *The DisplayInterface* Interface

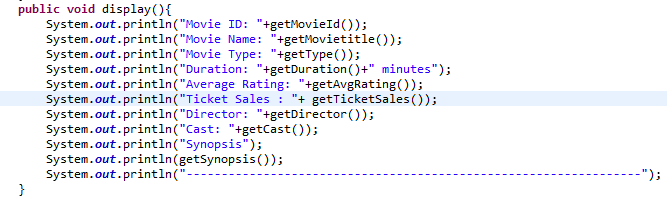
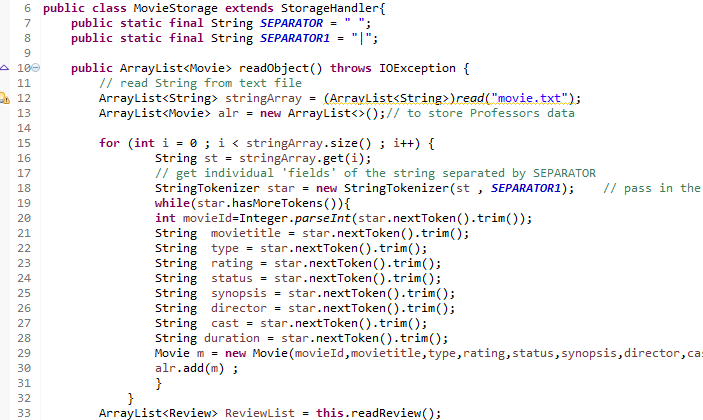
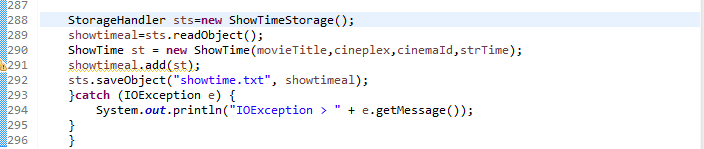
**

Figure 3.3: *display() from Movie.Java*

An abstraction of the *Movie* class is used in displayDetailMovie(). We can modify *Movie.display()* prints without changing the source code of *displayDetailMovie() since displayDetailMovie() merely calls Movie.Display().*

**Liskov Substitution Principle(LSP)**

Figure 4.1: *MovieStorage is a subclass of StorageHandler*

****Figure 4.2: *MovieStorage is passed*

The code in figure 4.2 continues to function properly when a derivative of *StorageHandler* class(*MovieStorage* class) is passed. This because the pre-condition is not stronger than the base case(no parameter passed in for *readObject()* ) and the post-condition is no weaker than the base class (no parameter returned). Hence, *MovieStorage* is substitutable for its base class *StorageHandler*.

**Interface Segregation Principle (ISP)**

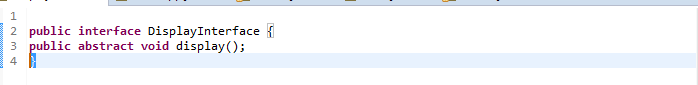
**

Figure 5.1: *The DisplayInterface* Interface, only one method is specified

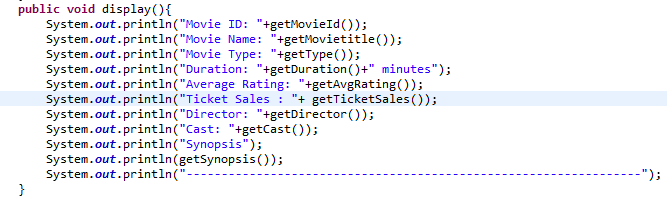
**

Figure 5.3: *display() from Movie.Java*

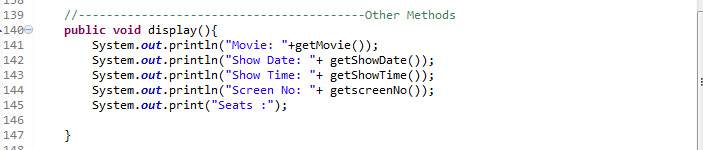
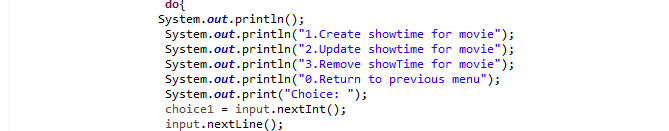
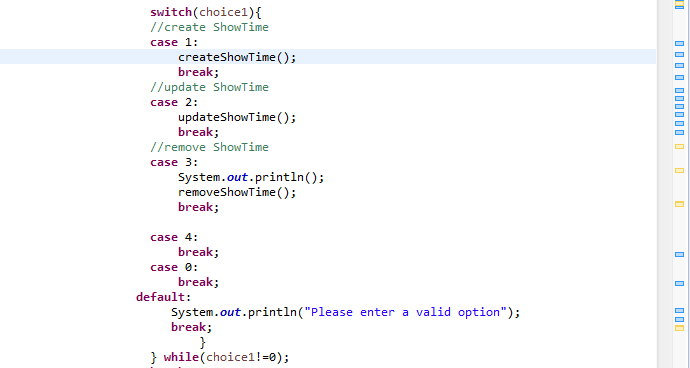
 Figure 5.4: *display() from Ticket.Java*

Figure 5.5: UML diagram showing the classes that implement *DisplayInterface.*

Each interface only implements a single function. Classes that makes use of each function is implements that interface according. This allows for easier categorisation and the user can easily know which interface functions is available for each class.

**Dont Repeat Yourself (DRY)**

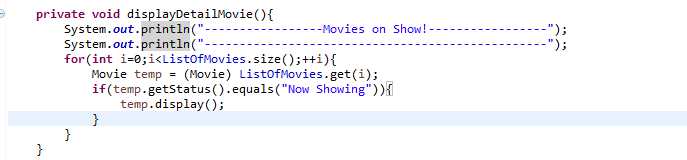
****

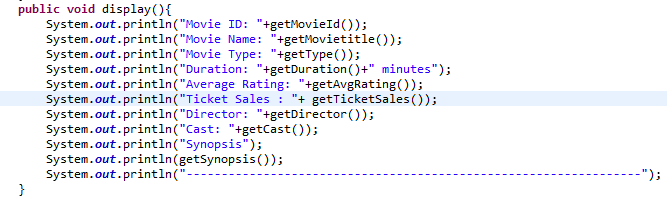
****** Figure 6.1: *a segment of updateShowTime() from UserLogin.java*

Each function is represented by a single, unambiguous, authoritative representation. There is no duplication of code and the name of function clearly defines it purpose.(ie. *removeShowTime*() removes show time as the name implies) This allows for easy maintenance and avoids logical contradiction.

**Dependency Injection Principle (DIP)**

Dependency injection separates the creation of a client's dependencies from the client's behavior.

­­­Figure 1.1: *displayDetailMovie() from UserLogin.java*

Figure 1.2: *display() from Movie.Java*

Using figure1.1 as an example, the high level module(*UserLogin.java)* does not depend on the low level module (*Movie.java).* Instead, both depends on abstractions, in this case, a instance of Movie, *temp.* Abstraction does not depend on detail, instead details depend on abstraction. In this case, each instance/abstraction of movie *temp* is distinct and separate and pass distinct parameter into *display()* via its *get* methods.