SE6329 Homework 2:

Team members: Jeffrey Hayward (jsh190004), Lekshan Jayasinghe(lxr200010), Kaden Tran(stt032000)

## Overview of Approach

There were three main components to this assignment, each of which is discussed below.

### Refactor Movie Rental Application

The movie rental application was refactored to improve the overall quality of the code. The refactoring process was done in several phases in order to minimize and reduce the chance of introducing bugs.

1. Simple changes such as renaming variables, replacing data types, cleaning up comments, and simple method extractions were performed first to clean up the code.
2. Two subclasses to the **Movie** class were created: **RegularMovie** and **ChildrensMovie**. These two subclasses were chosen because they represent different types of movies and can have different behavior or attributes, such as cost.
3. Two subclasses to the **Rental** class were created: **StandardRental** and **NewReleaseRental**. As discussed in class, ‘new release’ could be associated with either the movie or the rental. For our purposes, it made sense to make a ‘new release’ a type of rental as opposed to a type of movie.
   1. Time. A movie may initially be a new release and then that may change after some initial period of time. How is a rental affected when the movie changes from new release? For instance, if someone rents a new release on Sept 30 and the movie is no longer a new release on Oct 1, how is the rental affected? At this point that business rule is not known.
   2. Cost. Rental cost for new releases seems to be independent of the type of movie so it does not make sense to include that information in the Movie class.
   3. Late fees. Late fees are a rental attribute, not a movie attribute, because you have to know the duration of the rental. Making a subclass of Rental for new releases keeps that calculation within the rental space.
   4. Frequent renter points. Calculating frequent renter points requires information about the rental that is not known by the movie, such as the number of days rented. Making a new release rental type simplifies the process of calculating frequent renter points.
   5. Clarity. Both approaches, creating a new release subclass of Movie and a creating a subclass of Rental were tried and the Rental subclass, NewReleaseRental, produced a cleaner, more readable solution.

Because these changes were made across several steps, some of the changes made in earlier steps were again changed, or sometimes removed because they became unnecessary due to later change. As a result, some of the changes listed below may not appear in the final code.

### Create main() Method

**Main.java** was added to the project and a **main()** method was created to create movies, create rentals, and generate statements.

### Add Method to Print Statement in XML.

The homework requirements requested a new method to print a statement in XML format. **Customer.statementxml()** was added for this purpose. The original **Customer.statement()** method remains and generates a statement in the original format.

The Strategy pattern was used to allow the same rental calculations to be used to generate both regular and XML statements. A new interface, Receipt, was added to the application and the statement process was modified to use this interface to generate elements of the receipt and two classes, **TextReceipt** and **XmlReceipt**, implement this interface. **Customer.statement**() uses **TextReceipt** to generate output and **Customer.statementxml**() uses **XmlReceipt** to generate output.

## Summary of refactoring:

1. At least 3 method extraction operations: #8, #9, #10, #11
2. At least 3 creation of 3 new classes: #18. #19, #23, #26, #27
3. At least 3 moving method operations: #20. #24, #25, #28
4. At least 3 renaming operations: #2, #3, #4, #5
5. 1-2 replacements of data types: #6, #7, #32

## Other refactoring types:

1. Convert magic numbers to constants: #1
2. Clean-up comment: #12, #13, #14, #15
3. Make instance variable final: #16, #17,#21, #22
4. Add exception handling: #29, #30, #31

# REFACTORING TO REMOVE CODE SMELLS

1. Refactoring type: Convert magic numbers to constants

Change made:

* Created constants RENTAL\_COST, RENTAL\_PERIOD in ChildrenMovie, RegularMovie class
* Created NEW\_RELEASE\_RENTAL\_COST, FREQUENT\_RENTER\_POINTS\_PER\_NEW\_RELEASE, BONUS\_POINTS\_MINIMUM\_RENTAL in NewReleaseRental class
* Created LATE\_FEE, FREQUENT\_RENTER\_POINTS\_PER\_MOVIE in StandardRental class

Explanation: These values were stored as constants (I.e. magic numbers) in the body of the source code and were converted to constant values to improve readability and maintainability.

1. Refactoring type: Rename variable

Change made: **totalAmount** was renamed to **amountOwed** in Customer class.

Explanation: the name **totalAmount** was ambiguous

1. Refactoring type: Rename variable

Change made: **result** wasrenamed to **receipt** in Customer class

Explanation: **result** is a generic name and did not represent that the contents of the variable contained a receipt of the customer’s rentals.

1. Refactoring type: Rename variable

Change made: **thisAmount** was renamed to **rentalAmount** in Customer class

Explanation: **thisAmount** was ambiguous and it was not clear that the value stored in thisAmount was the amount of the rental.

1. Refactoring type: Rename variable

Change made: **each** wasrenamed to **rental** in Customer class

Explanation: **each** was ambiguous and it was not clear that the value stored in each was an instance of Rental.

1. Refactoring type: Convert primitive to object

Change made: Converted **amountOwed** (**totalAmount**) from **double** to **Double in Customer class**

Explanation: Using the object type makes code that uses this value more readable, especially when string conversions are used.

1. Refactoring type: Upgrade data type

Change made: Changed \_**rentals** from **Vector** to generic **ArrayList** in Customer class

Explanation: The **Vector** data type in Java is often considered to be obsolete. The **ArrayList** is generally considered a good replacement. Additionally, operations such as iterating through an **ArrayList** are simplier and easier to read.

1. Refactoring type: method extraction

Change made: Extracted the switch statement in **Customer.statement()** into a separate method that calculates the rental amount: **calculateRentalAmount()**

Explanation: The switch statement performs a single function which is to calculate the rental cost, along with late fees, for a given movie.

1. Refactoring type: method extraction

Change made: Extracted frequent rental points calculation into a separate method: **calculateFrequentRenterPoints**()

Explanation: The calculation of frequent rental points is a separate calculation that can be done independently from other operations in the **Customer.statement**() method. Extracting this method will allow its implementation to change.

1. Refactoring type: method extraction

Change made: Extracted code that adds information about single movie to the receipt to: **makeReceiptLineItem**() in Customer class.

Explanation: Building the receipt contents is a separate function

1. Refactoring type: method extraction

Change made: Extracted code that adds information to the receipt footer to **makeReceiptFooter**()

Explanation: Building the footer contents is a separate function

1. Refactoring type: Clean-up comment

Change made: Removed comment “add footer lines” in Customer class.

Explanation: This comment added no value because it duplicated what the code was doing.

1. Refactoring type: Clean-up comment

Change made: Removed comment “add frequent renter points” in Customer class.

Explanation: This comment was redundant because it simply restated the next line of code.

1. Refactoring type: Cleanup comment

Change made: Removed comment “add bonus for a two day new release rental” in Customer class

Explanation: After refactoring the conditional, **isMultiDayNewReleaseRental**, the comment became redundant because it restated the code.

1. Refactoring type: Cleanup comment

Change made: Restated the comment “determine amounts for each line” to “determine cost of each movie rented” in Customer class.

Explanation: The original comment was not clear.

1. Refactoring type: Make instance variable final

Change made: Made \_**name** final in Customer class.

Explanation: This value should not change after the object is created. Making the variable final protects against side effects and also makes the programmer’s intent about the lifetime of the value clear.

1. Refactoring type: Make instance variable final

Change made: Made \_**rentals** final in Customer class

Explanation: This value should not change after the object is created. Making the variable final protects against side effects and also makes the programmers intent about the lifetime of the value clear.

1. Refactoring type: Class extraction

Change made: Created **ChildrensMovie** subclass of **Movie**

Explanation: A children’s movie is a type of Movie (ChildrensMove IS-A Movie)

1. Refactoring type: Class extraction

Change made: Created **RegularMovie** subclass of **Movie**

Explanation: A regular movie is a type of Movie (RegularMovie IS-A Movie)

1. Refactoring type: moving method

Change made: Moved **Customer.calculateRentalAmount(**) into Rental class (as **calculateAmount**())

Explanation: the amount of a movie rental is associated with the rental itself. The customer does not need to know about prices and late fee amounts and calculations.

1. Refactoring type: Make instance variable final

Change made: Changed Rental.\_movie to ‘final’

Explanation: This value should not change after the object is created. Making the variable final protects against side effects and also makes the programmer’s intent about the lifetime of the value clear.

1. Refactoring type: Make instance variable final

Change made: Changed Rental.\_daysRented to ‘final’

Explanation: This value should not change after the object is created. Making the variable final protects against side effects and also makes the programmer’s intent about the lifetime of the value clear.

1. Refactoring type: Class extraction

Change made: extract Receipt() into a class

Explanation: It can be separate concern and it can be a class by itself.

1. Refactoring type: moving method

Change made: moved makeReceiptLineItem() from Customer into Receipt class

Explanation: building receipt line item is a function of Receipt class.

1. Refactoring type: moving method

Change made: moved receiptFooter() from Customer into Receipt class

Explanation: building receipt footer is a function of Receipt class.

1. Refactoring type: Class extraction

Change made: extracted NewReleaseRental into subclass of Rental

Explanation:

1. Refactoring type: Class extraction

Change made: extracted StandardRental into subclass of Rental

Explanation: All type of movie rentals are considered StandardRental except New Release. It can be separate concern and it can be a class by itself.

1. Refactoring type: moving method

Change made: moved getTitle () from Movie into rental class

Explanation: Moving the getTitle() method into Rental helped remove a method-chain inside Customer

1. Refactoring type: Add exception handling

Change made: Added exception handling for movie title in Movie class

Explanation: Movie title cannot be null

1. Refactoring type: Add exception handling

Change made: Added exception handling for movie parameter and daysRented in Rental class

Explanation: Movie parameter cannot be null and daysRented must be one or greater.

1. Refactoring type: Add exception handling

Change made: Added exception handling for customer name in Customer class

Explanation: Customer name cannot be null

1. Refactoring type: Convert primitive to object

Change made: Converted frequentRenterPoints from intto Integer in Receipt, TextReceipt, and XmlReceipt class.

Explanation: Using the object type makes code that uses this value more readable, getting rid of primitive obsession code smell.