Benton Luu 10175103

CPSC 501 – Assignment 1: Refactoring

Link to GitHub Repository: https://github.com/bentonluu/CPSC501-LibrarySystem-Refactored

Refactoring 1: Duplicate code -> Pull Up Field and Pull Up Constructor Body
(GitHub SHA: 9b19284922deb43ca7d9fa07a6ccc1580e2a1e8b)

- The code smell that was identified was duplicate code in the two subclasses that extend the superclass. In both subclasses Book and Movie, they have a common 'availability' field which tracks whether the item is available to be borrowed or not. This field fits better being in the superclass RentableItem since the field is a common behaviour between the subclasses.

RentableItem.java

```
public abstract class RentableItem {
        public RentableItem(String title, Boolean availability) {
                this.title = title;
                this.availability = availability;
        }
}
```

Book.java

```
public class Book extends RentableItem {
        private Boolean availability;
        public Book(String bookTitle, int bookID, String authorName, Boolean availability) {
                super(bookTitle);
                this.bookID = bookID;
                this.authorName = authorName;
                this.availability = availability;
        }
}
```

Movie.java

```
public class Movie extends RentableItem {
        private Boolean availability = true;
        public Movie(String movieTitle, int movieID, String directorName, Boolean availability)) {
                super(movieTitle);
                this.movieID = movieID;
                this.directorName = directorName;
                this.availability = availability;
        }
}
```

- The refactoring methods that were used on the code were the *Pull Up Field* and *Pull Up Constructor Body*. The steps followed to implement these refactoring methods were:
    - In the superclass RentableItem, defined the common 'availability' field and set it to protected since it will only be accessed or modified by the subclasses.
    - Added the 'availability' field into the superclass class constructor body and updated the parameter list to indicate to the subclasses that it is a needed field in order for an item to be initialized.

Rentableitem.java

```
pubic abstract class RentableItem {
        protected Boolean availability;
        public RentableItem(String title, Boolean availability) {
                this.title = title;
                this.availability = availability;
        }
}
```

    - Removed the duplicated field instance from both of the subclasses Book and Movie.

Book.java

```
public class Book extends RentableItem {
        public Book(String bookTitle, int bookID, String authorName, Boolean availability) {
                super(bookTitle, availability);
                this.bookID = bookID;
                this.authorName = authorName;
        }
}
```

Movie.java

```
public class Movie extends RentableItem {
        public Movie(String movieTitle, int movieID, String directorName, Boolean availability) {
                super(movieTitle, availability);
                this.movieID = movieID;
                this.directorName = directorName;
        }
}
```

    - Tested the refactored code and verified that it was working as expected before the changes were implemented.

- The refactoring methods affected files: RentableItem.java, Book.java and Movie.java

- To ensure that the code was functioning properly before and after the refactoring changes were applied, JUnit test ‘availabilityTest’ was created that was used to verify that the code was working as expected. The test comprised of a borrower borrowing both a book and movie item and checks to see if the ‘availability’ field of those items are false, meaning that the items were borrowed and are unavailable to be borrowed by someone else. Then the borrower returns both items and the ‘availability’ field is checked to see if it is true, which means the item was returned and is now available to be borrowed by someone else.
- The code was better structured after the refactoring methods such that it removed redundant code in both subclasses Book and Movie that had essentially the same behaviour. By moving the common ‘availability’ field into the superclass and it’s constructor body, it reduced the potential for future bugs since each subclass would not be implementing their own version of the same functionality. It puts the common field in a shared place that can be accessed by both subclasses and will be easier to debug the functionality down the line.
- As a result of this refactoring it suggests two other related refactoring methods that could also be made, the Pull Up Method and Rename Method as there are still methods that relate to the ‘availability’ field that could also be moved into the superclass.

Refactoring 2: Duplicate code -> Pull Up Method and Rename Method
(GitHub SHA: 612f452cd0efd44889cc7bd76f6361f6f8002cb4)

- The code smell that was detected was duplicate code as there are methods that can be removed from the subclasses and moved into the superclass. The methods that access and modify the field 'availability' are both implemented in the subclasses Book and Movie but perform the same function. Since the 'availability' field was moved into the superclass RentableItem in the previous refactoring, it would make more sense to move the methods associated with that field into the superclass as well to keep everything together.

RentableItem.java

```
public abstract class RentableItem {
        protected Boolean availability;

        public RentableItem(String title,
        Boolean availability) {
                this.title = title;
                this.availability = availability;
        }
}
```

Book.java

```
public class Book extends RentableItem {
        public Boolean getAvailability() { return availability; }
        public void bookUnavailable() { availability = false; }
        public void bookAvailable() { availability = true; }
}
```

Movie.java

```
public class Movie extends RentableItem {
        public Boolean getAvailability() { return availability; }
        public void movieUnavailable() { availability = false; }
        public void movieAvailable(){ availability = true; }
}
```

Borrower.java

```
public class Borrower {
        public void requestItem(Object obj, String borrowerName) {
        …
                if (ableToBorrow(obj))
                { … ((Book) obj).bookUnavailable(); …}
        …
                if (ableToBorrow(obj))
                { … ((Movie) obj).movieUnavailable(); …}
        …
        }
        public void returnItem(Object obj, LocalDate returnDate) {
        …
                else { … ((Book) obj).bookAvailable(); … }
        …
                else { … ((Movie) obj).movieAvailable(); … }
        …
        }
}
```

- The two refactoring methods that were applied were the *Pull Up Method and Rename Method.* The steps that were followed to integrate the refactoring methods were:
  - Defined the 'getAvailiabity' method in the superclass RentableItem since it's a shared functionality used by both subclasses.
  - Removed the 'getAvailiabity' method from the subclasses Book and Movie.
  - Ran the JUnit test 'availabilityTest' and verified that it still passed after the modification was made to the code.

RentableItem.java

```
public abstract class RentableItem {
        protected Boolean availability;
        public RentableItem(String title, Boolean availability) {
                this.title = title;
                this.availability = availability;
        }
        public Boolean getAvailability() { return availability; }
}
```

Book.java

```
public class Book extends RentableItem {
        public void bookUnavailable() { availability = false; }
        public void bookAvailable() { availability = true; }
}
```

Movie.java

```
public class Movie extends RentableItem {
        public void movieUnavailable() { availability = false; }
        public void movieAvailable() { availability = true; }
}
```

  - Defined 2 generic setter methods in the superclass RentableItem for the 'availability' field that would replace the subclasses' specific setter methods. The renamed generic setter methods are 'itemUnavailable' and 'itemAvailable'.
  - Removed each of the subclass's specific setter methods, 'bookUnavailable', 'bookAvailable', movieUnavailable', and 'movieAvailable'.

RentableItem.java

```
pubic abstract class RentableItem {
        protected Boolean availability;

        public Boolean getAvailability() { return availability; }
        public void itemUnavailable() { availability = false; }
        public void itemAvailable() { availability = true; }
}
```

Book.java

```
public class Book extends RentableItem {
}
```

Movie.java

```
public class Movie extends RentableItem {
}
```

- Updated the method calls in the Borrower class that use to call the subclass specific setter methods and changed the method call signature to the renamed generic setter methods.

Borrower.java

```
public class Borrower {
        public void requestItem(Object obj, String borrowerName)
        {
        …
                if (ableToBorrow(obj))
                { … ((Book) obj).itemUnavailable(); …}
        …
                if (ableToBorrow(obj))
                { … ((Movie) obj).itemUnavailable(); …}
        …
        }
        public void returnItem(Object obj, LocalDate returnDate) {
        …
                else { … ((Book) obj).itemAvailable(); … }
        …
                else { … ((Movie) obj).itemAvailable(); … }
        …
        }
}
```

- Tested the code to ensure that the existing JUnit test ‘availabilityTest’ still passed.

- The refactoring methods affected files: RentableItem.java, Book.java, Movie.java, and Borrower.java
- Using the same JUnit tested used previously, ‘availabilityTest’, ensured that when an item was borrowed, the ‘availability’ field changed from true to false and when the item is returned it changed from false to true.
- The improvement to the structure of the code is that the maintainability is significantly improved since all the code pertaining to the ‘availability’ field is located in one class now. Prior to the refactoring, each subclass had its own methods to access and modify the ‘availability’ field which would make it difficult to troubleshoot an issue since there were two sources where the issue might have originated from. Furthermore, it cleans up both subclasses’ code making them much smaller as the redundant methods were removed and enables the code to be more easily understood in terms of the specific attributes associated with a subclass.
- With regards to other refactoring methods that could be applied to this section of code from this refactoring, it does seem to enable any further refactoring methods at this moment in time.

Refactoring 3: Feature Envy -> Move Method
(GitHub SHA: c590aea50d50abfc13f57d38531ea9a3c7b94839)

- The code smell that was identified was Feature Envy as there were several methods being called to access fields that originate in another class in order to perform a conditional check and calculation. The method 'checkPastDue' is better suited in the class TransactionRecord instead of the class Borrower since the 'fineAmount' field and 'dueDate' fields are part of the TransactionRecord class. By moving it there, it reduces the number of method calls the Borrower class has to make to the TransactionRecord class.

Borrower.java

```
public class Borrower {
        public void returnItem(Object obj, LocalDate returnDate) {
                …
                if (obj instanceof Book) { …
                        if (checkPastDue(rentedItem.get(i), returnDate) == true) { … }
                        …
                }
                …
                else if (obj instanceof Movie) { …
                        if (checkPastDue(rentedItem.get(i), returnDate) == true) { … }
                        …
                }
        }

        public Boolean checkPastDue(TransactionRecord item, LocalDate returnDate) {
          long days = ChronoUnit.DAYS.between(returnDate, item.getDueDate());

          if (days < 0 && item.getFineAmount() == -1) {
            double fine = 1.5*Math.abs(days);
            item.setFineAmount(fine);
            return true;
          }

          return false;
        }
}
```

TransactionRecord.java

```
public class TransactionRecord {
        private LocalDate dueDate;
        private double fineAmount;

        public double getFineAmount() { return fineAmount; }

        public void setFineAmount(double fineAmount) { this.fineAmount = fineAmount; }

        public LocalDate getDueDate() { return dueDate; }
}
```

- The refactoring method that was applied was the *Move Method*. The steps followed to implement the refactoring method were:
  - Determined that the 'checkPastDue' method should be moved from the Borrower class to the TransactionRecord class.
  - Defined the method 'checkPastDue' method in the TransactionRecord class and copied the source method code.
  - Updated the parameter list for the 'checkPastDue' method to only take the return date of the item since it does not require the TransactionRecord object to be passed anymore.
    - Additionally, replaced the method calls in the 'checkPastDue' method to use actual fields in the TransactionRecord class.

TransactionRecord.java

```
public class TransactionRecord {
        public Boolean checkPastDue(LocalDate returnDate) {
          long days = ChronoUnit.DAYS.between(returnDate, dueDate);

          if (days < 0 && fineAmount == -1) {
            fineAmount = 1.5*Math.abs(days);
            return true;
          }

          return false;
        }
}
```

  - Removed the 'checkPastDue' method from the Borrower class and tested the code to verify it was still working as expected.

Borrower.java

```
public class Borrower {
        public void returnItem(Object obj, LocalDate returnDate) {
                …
                if (obj instanceof Book) { …
                        if (checkPastDue(returnDate) == true) { … }
                        …
                }
                …
                else if (obj instanceof Movie) { …
                        if (checkPastDue(returnDate) == true) { … }
                        …
                }
        }
}
```

- The refactoring method affected files: TransactionRecord.java and Borrower.java

- To test that the ‘checkPastDue’ method was functioning properly before and after the refactoring was applied, created JUnit tests ‘returnItemTest’ and ‘returnItemPastDueTest’. The ‘returnItemTest’ verifies that if a borrower borrows an itemed, they should be able to return the item if it is before the item’s due date. The ‘returnItemPastDueTest’ checks that if a borrower returns an item past it’s due date, the borrower must pay a late fine first before the item can be returned. After the late fine is paid, the borrower would be allowed to return the item.
- After the refactoring, the code structure is more understandable in that it separates the responsibilities of each class. Work that is needed to done on a TransactionRecord object’s field is done in that class and thus reduces the number of method calls the Borrower class would need to make in order to get the information required to perform the conditional check and calculation. In general, it makes more sense for the ‘checkPastDue’ method to be in TransactionRecord class since it was accessing the ‘fineAmount’ and ‘dueDate’ fields which belong to that class. The Borrower class now simply makes one method call to check if an item is able to returned or not.
- In terms of other refactoring methods that could be applied to this section of code, it does seem that this refactoring method allows for any further refactoring methods to be used at this point in time.

Refactoring 4: Duplicate Code and Long Method -> Extract Method
(GitHub SHA: 2e112e69e58091cfec66771c9a1a9698fa4b9e99)

- The code smells that were detected in this program were long method and duplicate code. In several methods of the class Borrower, there was redundant code that would loop through the list of items that a borrower currently had rented and looked through the list to find the TransactionRecord object associated with the item passed through as a parameter. Thus, it resulted in making methods longer than they needed to be and duplicated code that essentially had the same functionally.

Borrower.java

```
public class Borrower {
    public void returnItem(Object obj, LocalDate returnDate) {
       for (int i = 0; i < rentedItem.size(); i++) {
          if (obj instanceof Book) {
             if (rentedItem.get(i).getItemName().equals(((Book) obj).getTitle())) {
                if (rentedItem.get(i).checkPastDue(returnDate) == true) { … }
                else {
                   rentedItem.get(i).setReturnStatus(true);
                   ((Book) obj).itemAvailable();
                   rentedItem.remove(i);
                }
             }
          }
          else if (obj instanceof Movie) {
             if (rentedItem.get(i).getItemName().equals(((Movie) obj).getTitle())) {
                if (rentedItem.get(i).checkPastDue(returnDate) == true) { … }
                else {
                   rentedItem.get(i).setReturnStatus(true);
                   ((Movie) obj).itemAvailable();
                   rentedItem.remove(i);
                }
             }
          }
       }
    }
```

```
    public Boolean ableToBorrow(Object obj) {
    …
       for (TransactionRecord item : rentedItem) {
          if (item.getFineAmount() > 0 ) {
             System.out.println("Item cannot be borrowed as there is
    an outstanding late fine.");
             return false;
          }
       }
    …
    }
    public void paidFine(Object obj) {
       for (TransactionRecord item : rentedItem) {
          if (item.getItemName().equals(((Book) obj).getTitle())) {
             item.setFineAmount(0);
          }
       }
    }
    public void renewItem(Object obj, int num) {
       for (TransactionRecord item : rentedItem) {
          if (item.getItemName().equals(((Book) obj).getTitle())) {
             item.extendDueDate(num);
          }
       }
    }
    public String getItemDetails(Object obj) {
       for (TransactionRecord item : rentedItem) {
          if (obj instanceof Book) {
             if (item.getItemName().equals(((Book) obj).getTitle())) {
                return item.transactionDetails();
             }
          }
       }
       return "Item details not found";
    }
}
```

- The refactoring method that was applied was the *Extract Method*. Steps that were followed for the refactoring were:
  - Created a new method called 'findTransactionRecord' in the Borrower class and extracted code from source method 'returnItem'.

Borrower.java

```
public class Borrower {
    public TransactionRecord findTransactionRecord(List<TransactionRecord> transactionRecords, Object obj) {
      for (TransactionRecord item : rentedItem) {
        if (obj instanceof Book && item.getItemName().equals(((Book) obj).getTitle())) {
          return item;
        }
        else if (obj instanceof Movie && item.getItemName().equals(((Movie) obj).getTitle())) {
          return item;
        }
      }
      return null;
    }
}
```

  - Replaced the extracted code from the source method 'returnItem' with a method call to the 'findTransactionRecord' method and tested that the code still worked as normal.

Borrower.java

```
public class Borrower {
    public void returnItem(Object obj, LocalDate returnDate) {
      TransactionRecord item = findTransactionRecord(rentedItem, obj);

      if (item.checkPastDue(returnDate) == true) {
        System.out.println("Must pay late fine before item can be returned");
      }
      else {
        item.setReturnStatus(true);
        rentedItem.remove(rentedItem.indexOf(item));

        if (obj instanceof Book) {
          ((Book) obj).itemAvailable();
        }
        else if (obj instanceof Movie) {
          ((Movie) obj).itemAvailable();
        }
      }
    }
}
```

- Replaced other methods ('ableToBorrow', 'paidFine', 'renewItem' and 'getItemDetails') that used the same functionality with a method call to the extract method 'findTransactionRecord' and tested the changes to ensure that the code was working properly.

Borrower.java

```
public class Borrower {
    public Boolean ableToBorrow(Object obj) {
    …
      TransactionRecord item = findTransactionRecord(rentedItem, obj);
      if (item != null) {
        if (item.getFineAmount() > 0 ) {
          System.out.println("Item cannot be borrowed as there is an outstanding late fine.");
          return false;
        }
      }
      return true;
    }

    public void paidFine(Object obj) {
      TransactionRecord item = findTransactionRecord(rentedItem, obj);
      item.setFineAmount(0);
    }

    public void renewItem(Object obj, int num) {
      TransactionRecord item = findTransactionRecord(rentedItem, obj);
      item.extendDueDate(num);
    }

    public String getItemDetails(Object obj) {
      TransactionRecord item = findTransactionRecord(rentedItem, obj);

      if (item == null) {
        return "Item details not found";
      }
      return item.transactionDetails();
    }
}
```

- The refactoring method affected file: Borrower.java

- The refactoring method that was integrated into the code was tested using the existing JUnit tests for the other refactoring methods as well as a new JUnit test called ‘renewItemTest’ was created. The ‘renewItemTest’ checks that when a borrower requests for an item to have its due date extended, the associated transaction record would be updated to reflect the new due date of the item. These tests performed ensure that the behaviour of the code after the integration of the refactoring works as it previously did and behaviours such as being able to borrow and return an item and pay a late fine if an item was returned after it’s due date works properly.
- The code structure was enhanced after the refactoring as it reduced the complexity of the code specifically the source method ‘returnItem’ by eliminating the need for multiple nested conditional statements. Additionally, by removing the redundant code in the other methods and replacing it with a single method call to the extracted method, ‘findTransactionItem’, it made the code length shorter which helps with readability of what each method does. Furthermore, by putting the functionality of finding the associated transaction record of an item in a single method rather than in multiple spots in the code, it mitigates the risks of a related error appearing in various parts in the code.
- Since the Extract Method reduced the complexity of the code in the Borrower class, it could be further reduced by using the refactoring method, Extract Class in order to split up the responsible in the class.

Refactoring 5: Larger Class -> Extract Class
(GitHub SHA: 9c6eb573523df798184a7858441f3d0681530474)

- The identified code smell was a larger class as some of the responsibilities in the Borrower class could be split into another class to shorten the length of code. Some fields and methods in the Borrower class would be better suited in another class and objects of the Borrower class would be able to call methods from that class.

Borrower.java

```
public class Borrower {
    public void requestItem(Object obj, String borrowerName) {
        if (obj instanceof Book) {
            if (ableToBorrow(obj, rentedItem)) { … }
        }
        else if (obj instanceof Movie) {
            if (ableToBorrow(obj, rentedItem)) { … }
        }
    }

    public void returnItem(Object obj, LocalDate returnDate) {
        TransactionRecord item = findTransactionRecord(rentedItem, obj);
        …
    }

    public Boolean ableToBorrow(Object obj) {
    …
        TransactionRecord item = findTransactionRecord(rentedItem, obj);
    …
    }

    public void paidFine(Object obj) {
        TransactionRecord item = findTransactionRecord(rentedItem, obj);
        item.setFineAmount(0);
    }

    public void renewItem(Object obj, int num) {
        TransactionRecord item = findTransactionRecord(rentedItem, obj);
        item.extendDueDate(num);
    }

    public String getItemDetails(Object obj) {
        TransactionRecord item = findTransactionRecord(rentedItem, obj);

        if (item == null) {
            return "Item details not found";
        }
        return item.transactionDetails();
    }

    public TransactionRecord findTransactionRecord(List<TransactionRecord>
    transactionRecords, Object obj) {
        for (TransactionRecord item : rentedItem) {
            if (obj instanceof Book && item.getItemName().equals(((Book) obj).getTitle())) {
                return item;
            }
            else if (obj instanceof Movie && item.getItemName().equals(((Movie) obj).getTitle())) {
                return item;
            }
        }
        return null;
    }

}
```

- The refactoring that was applied was the *Extract Class* method. The steps that were followed to implement this refactoring method were:
    - Created a new class named 'Library'. The responsible of this class will be used to perform the checks and update TransactionRecord objects of items that a borrower has rented.

Library.java

```
public class Library() {
    private String name, address;

    public Library(String name, String address) {
        this.name = name;
        this.address = address;
    }
}
```

    - Moved method 'findTransactionRecord' from the Borrower class to the Library class and tested it. Several methods depend on this method so wanted to ensure that moving this method did not break the code.

Library.java

```
public class Library() {
    public TransactionRecord findTransactionRecord(List<TransactionRecord> transactionRecords, Object obj) {
      for (TransactionRecord item : rentedItem) {
        if (obj instanceof Book && item.getItemName().equals(((Book) obj).getTitle())) {
          return item;
        }
        else if (obj instanceof Movie && item.getItemName().equals(((Movie) obj).getTitle())) {
          return item;
        }
      }
      return null;
    }
}
```

- Moved methods ‘getItemDetails’ and ‘renewItem’ into the Library class and tested it with the existing unit tests to ensure code was still functioning as expected.
  - Updated the parameter list of methods getItemDetails’ and ‘renewItem’ to include a Borrower object.

Library.java

```
public class Library() {
    public void renewItem(Object obj, Borrower borrower, int num) {
       TransactionRecord item = findTransactionRecord(borrower.getRentedItem(), obj);
       item.extendDueDate(num);
    }

    public String getItemDetails(Object obj, Borrower borrower) {
       TransactionRecord item = findTransactionRecord(borrower.getRentedItem(), obj);

       if (item == null) {
          return "Item details not found";
       }
       return item.transactionDetails();
    }
}
```

- Moved method ‘ableToBorrow’ into the Library class and tested it with the existing unit tests.
  - Updated the parameter list of method ‘ableToBorrow’ to include a list of TransactionRecord objects.

Library.java

```
public class Library() {
    public Boolean ableToBorrow(Object obj, List<TransactionRecord> rentedItem) {
    …
       TransactionRecord item = findTransactionRecord(rentedItem, obj);
    …
    }
}
```

- Updated the remaining methods in the Borrower class ('requestItem', 'returnItem' and 'paidFine') to call the methods from the Library class.
  - Added a Library object as a parameter for the methods 'requestItem', 'returnItem' and 'paidFine' in order for those methods to call methods in the Library class.
- Added a getter method named 'getRentedItem' in the Borrower class to allow for the Library class to access the Borrower object's 'rentedItem' field.

Borrower.java

```
public class Borrower {
    public List<TransactionRecord> getRentedItem() {
      return rentedItem;
    }

    public void requestItem(Object obj, String borrowerName, Library library) {
      if (obj instanceof Book) {
        if (library.ableToBorrow(obj, rentedItem)) { … }
      }
      else if (obj instanceof Movie) {
        if (library.ableToBorrow(obj, rentedItem)) { … }
      }
    }

    public void returnItem(Object obj, LocalDate returnDate, Library library) {
      TransactionRecord item = library.findTransactionRecord(rentedItem, obj);
      …
    }

    public void paidFine(Object obj, Library library) {
      TransactionRecord item = library.findTransactionRecord(rentedItem, obj);
      item.setFineAmount(0);
    }
}
```

- Tested the refactoring changes and verified that it was working as expected before the changes were implemented.

- The files that were affected as a results of the refactoring were: Borrower.java and TransactionRecord.java

- The code was tested with the existing JUnit tests that were used to test the other refactoring methods but were updated to utilize the new Library class. A library object was initialized and was passed into methods that were updated to require a Library objects as a parameter. To fully ensure that the Library class was created correctly, created a new JUnit test named 'libraryTest'. This test was used to verify that the methods in the Library class were functioning properly with the existing code structure.
- Overall, the structure of the code was improved as the responsibility the Borrower class is spread between two classes now, the Borrower and Library classes. Compared to before the refactoring with only the Borrower class where it handled various responsibilities such as checks and calculations, it simplifies the complexity of the code and provides a better understanding of the purpose of what each class does. The methods that were used to check whether an item was able to be borrowed or not and methods used to calculate a late fine were moved to the Library class to reduce the complexity of the code in the Borrower class and only contains methods that are specific to what a borrower is able to do. Additionally, the length of the Borrower class has significantly reduced making it easier to understand the class and its relationship with the other classes in the program.
- With regards to other refactoring methods that could be applied to this section of code from this refactoring, it does seem to enable any further refactoring methods at this moment in time.