# CPSC 501 – Assignment 1

**Artem Golovin**
30018900

## Overview

The project that was refactored for this assignment is iKeirNez/assessment-loan-system. To preform refactoring on the project, the following fork was created awave1/assessment-loan-system. The fork contains two branches: `master` and `refactor`. The initial work is left untouched in `master` branch and the refactoring was done in `refactor` branch. To simplify the access, `refactor` branch is default (main) branch.

Original project did not include any README or instructions how to get up and running with the project, therefore I included README, describing the steps necessary to build, run, and test the code. To make it easier to manage dependencies and to build the project, I added support for gradle.

### Refactoring structure

The refactorings that were made in the project, can be categorized into three categories: **minor**, **medium** and **major**. **Minor** refactorings were usually made together with **major** and **minor** refactorings, thus no commits were directly dedicated to **minor** refactorings.

## Major changes

This section explains and displays the number of **major** changes that were made to the project during refactoring process. Each major change is associated with commit hash.

### Using Visitor pattern to eliminate `instanceof` checking (2534ae1e)

This substantial refactoring was done at the end when I noticed a block of `instanceof` checks that were doing the same thing. To eliminate such code smell and make the code more robust, we make use Visitor pattern. That is, we'll create a special Visitor class that will "visit" all required methods, thus removing `instanceof` checking and abstracting the functionality. To implement this pattern, I added an abstract method to abstract `Item` class:

```
1  public abstract class Item {
2    // ...
3    public abstract SelectItem<? extends Item> accept(MenuVisitor
         visitor);
4  }
```

As you can see, there's a parameter that will need to be passed, `MenuVisitor visitor`. `MenuVisitor` is our visitor that will be calling specific functions, declared in the `MenuVisitor` class. For example:

```
1  // MenuVisitor.java
2  public class MenuVisitor implements Visitor {
3    // ...
4    @Override
5    public SelectItem<Book> addBook(Book book) {
6      return new SelectItem<>(book, BOOK_MANAGER, BOOK_MANIPULATOR);
7    }
8  }
9
10 // Book.java
11 public class Book extends Item {
12   // ...
13   @Override
14   public SelectItem<Book> addBook(Book book) {
15     return new SelectItem<>(book, BOOK_MANAGER, BOOK_MANIPULATOR);
16   }
17 }
```

Therefore, now instead of performing explicit `instanceof` checks, we can do this:

```
1  MenuVisitor menuVisitor = new MenuVisitor();
2  menuVisitor.build(menu, stockRepo.getAll());
```

and `MenuVisitor` will take care of which object to create.

As was stated above, this improves the code by removing excessive `instanceof` checking with much cleaner pattern that could be used in other potential places where a lot of `instanceof`'s is used. This improves the readability of the code by encapsulating the details of implementation.

To perform this refactor, `MenuVisitor` and `Visitor` were introduced. `MenuVisitor` class implements the `Visitor` interface that contains "visit" methods. Therefore, we introduced a new class and a new interface to Replace Conditional with Polymorphism.

The following files were changed and added as a result of this refactoring:

- src/main/java/com/keirnellyer/glencaldy/item/Book.java
- src/main/java/com/keirnellyer/glencaldy/item/Disc.java
- src/main/java/com/keirnellyer/glencaldy/item/Item.java
- src/main/java/com/keirnellyer/glencaldy/item/Journal.java
- src/main/java/com/keirnellyer/glencaldy/item/Video.java
- src/main/java/com/keirnellyer/glencaldy/menu/option/stock/EditStockOption.java
- src/main/java/com/keirnellyer/glencaldy/menu/option/stock/SelectItem.java
- src/main/java/com/keirnellyer/glencaldy/util/MenuVisitor.java
- src/main/java/com/keirnellyer/glencaldy/util/Visitor.java

To test this code, I added test `Item`s that allowed to build the menu with these items and to test `visit` methods. For example,

```
1  Menu menu = new Menu("Test Menu");
2  menuVisitor.build(menu, stockRepository.getAll());
3
4  assertFalse(menu.getItems().isEmpty());
5
6  for (int i = 0; i < stockRepository.getAll().size(); i++) {
7    assertNotNull(menu.getItems().get(Integer.toString(i)));
8  }
```

The rest of the methods were simply tested to make sure that they do not return `null`, since `Menu` class relies on this `MenuVisitor` class.

With this refactoring applied, code smell has been removed and the code became more OO. We no longer do `instanceof` checks, which is considered bad design. As well as in the future, this codebase can now use Visitor pattern thus getting rid of "100 line" if statements.

### Abstracting busy waiting for user input (bd538f0)

The application relies on input from keyboard. In classes, where the user input is required, the original code included use of `do { ... } while(...);` loops, to wait for valid user input. User input is required in `Property<T>` (and its subclasses), `Menu` and `Controller`. As a result, `do { ... } while(...);` loop has been moved to separate class called `ConsoleInput<T>`, in method `public Optional<T> waitForInput(InputWait<T> inputWait)`. `InputWait<T>` is an interface with only one method, passing it as a parameter allows us to pass anonymous lambda function (e.g. `arg -> {/* function body */}`) as a parameter, instead of implementing a method, that was declared in the interface. The following is the implementation of `waitForInput` method:

```
public Optional<T> waitForInput(InputWait<T> inputWait) {
  Optional<T> fetchedObj;
  do {
    fetchedObj = inputWait.getInput(this.scanner);
  } while (!fetchedObj.isPresent());

  return fetchedObj;
}
```

When `waitForInput` is called, we have to supply instance of `InputWait`, for example:

```
// Set the scanner
setScanner(scanner);
Optional<Option> option = waitForInput(scnr -> {
  /*
   * Do all the necessary things here, using Scanner scnr variable
   */

  // Return Optional object result
  return Optional.of(obj);
});

```

```
//... Optional<Option> option can later be safely unwrapped and used
```

By abstracting the `do{...} while()` loop into a separate method, we got rid of duplicated code and made it more readable. Also other simple classes that need to wait for user input can now inherit this class and call `waitForInput`.

To do this refactoring, **Replace Method with Method object** technique was used. For example, with this technique applied, we got rid of `processLogin` method in `User` class, so the following code:

```
private User processLogin() {
  User user;
  do {
    System.out.println("Please enter your username.");
    String username = scanner.next();

    System.out.println("Please enter your password.");
    String password = scanner.next();

    user = model.getUserRepository().getExact(username, password);

    if (user == null) { // invalid credentials
      System.out.println("Invalid credentials, please try again.");
    }
  } while (user == null);

  return user;
}
```

was replaced with:

```
Optional<User> user = waitForInput(s -> {
  User usr;
  System.out.println("Please enter your username.");
  String username = scanner.next();

  if (user != null) {
    System.out.println("Please enter your password.");
    String password = scanner.next();
    usr = model.getUserRepository().getExact(username, password);
    if (usr == null) { // invalid credentials
      System.out.println("Invalid credentials, please try again.");
```

```
    }

    return Optional.of(usr);
});
```

The following files were changed and added as a result of this refactoring:

- src/main/java/com/keirnellyer/glencaldy/manipulation/property/type/Property.java
- src/main/java/com/keirnellyer/glencaldy/menu/Menu.java
- src/main/java/com/keirnellyer/glencaldy/runtime/Controller.java
- src/main/java/com/keirnellyer/glencaldy/runtime/Controller.java
- src/main/java/com/keirnellyer/glencaldy/util/ConsoleInput.java
- src/main/java/com/keirnellyer/glencaldy/util/InputWait.java

To test `ConsoleInput<T>`, I had to mock user input, using `ByteArrayInputStream`. There are four tests for this class, located in `ConsoleInputTest`. Two tests are used to test `ConsoleInput<User>` and one more to test and display functionality using built in object `ConsoleInput<String>`.

Adding `ConsoleInput<T>` allows us to add ability to use busy-waiting on any class, by utilizing Java Generics.

### Add `UserInfo` class to eliminate long parameter list for `User` classes (fcaddaa8, 8fb88595)

This refactoring affected all of the `User` classes, so `User` and its child classes. All of these classes required a lot of parameters when creating a new object. To eliminate long parameter list, I introduced a `UserInfo` class. `UserInfo` class uses chaining methods to build information about a required user type. It also contains every possible field that any given `User` might use. By doing so, we can use it in all of the `User` child classes. As a result, `User` now has a `UserInfo` field and every set method now has to also set the value for `UserInfo` field.

To perform this refactoring, I applied "Extract Class" technique. Here's the snippet of `UserInfo` class

```
public class UserInfo {
  private String username;
  private String password;
  private LocalDate birthDate;
  private String phoneNumber;
```

```
  private String address;
  private int staffId;
  private String email;
  private String extension;

  // ... setters and getters

  // Example of a setter
  // As you can see, it returns this to enable chaining setters
  public UserInfo setUsername(String username) {
    this.username = username;
    return this;
  }
}
```

This data class allowed to replace long parameter constructors like these:

```
public Administrative(String username, String password, String
    address, String phoneNumber, LocalDate birthDate, int id, String
    email, String extension) {
     super(username, password, address, phoneNumber, birthDate, id,
         email, extension);
}
```

with this:

```
public Administrative(UserInfo info) {
    super(info);
}
```

The following files were changed as a result of this refactoring:

- src/main/java/com/keirnellyer/glencaldy/manipulation/user/StaffProperties.java
- src/main/java/com/keirnellyer/glencaldy/user/Administrative.java
- src/main/java/com/keirnellyer/glencaldy/user/Casual.java
- src/main/java/com/keirnellyer/glencaldy/user/Member.java
- src/main/java/com/keirnellyer/glencaldy/user/Staff.java
- src/main/java/com/keirnellyer/glencaldy/user/User.java

- src/main/java/com/keirnellyer/glencaldy/user/UserInfo.java

The testing of this code was done in commit 8fb88595. To test `UserInfo`, I created instance of `UserInfo` object and passed it to one of child classes of `User` and then, using Java `equals` I compared the object I created with instance of it in `User` subclass:

```
@Test
void shouldCreateAdministrativeObjectUsingUserInfo() {
  UserInfo userInfo = new UserInfo();
  userInfo
    .setUsername("user")
    .setPassword("pass")
    .setAddress("addr")
    .setPhoneNumber("1243")
    .setBirthDate(LocalDate.of(2019, 3, 3))
    .setStaffId(1)
    .setEmail("email@email.com")
    .setExtension("ext");

  assertNotNull(userInfo);

  Administrative admin = new Administrative(userInfo);

  assertNotNull(admin);
  assertEquals(userInfo, admin.getUserInfo());
}
```

As was stated before, this code gets rid of long constructors. With `UserInfo`, it's easy to create and store information about `Users`

## Medium changes

### Replace `Repository<K, V>` interface with abstract class (67287c0); Introduce `add` to Repository classes and `getExact` to UserRepository class (aef6c56)

Initial code included `Repository<K, V>` as an interface, however, it didn't make sense for it to be an interface, because there was duplicated code in its child classes. Therefore, by abstracting `ArrayList<V> getAll()` and `public void add(V value)`, we can reuse this abstract class on any other repository

classes, if they need to be created in the future. Another chages that were added, include introducing `getExact` method in the `UserRepository`. This allowed up to remove `findUser` from `Controller` class, which clearly did not belong there. As well as adding `add(V... values)`, which helped to remove multiple calls when trying to add many things to a repository.

The `Repository<K, V>` needed to become an abstract class to eliminate code duplication throughout `Repository` children.

To do the refactorings, I used Pull Up Method technique as well as Move Method. As was stated before, the following methods were removed from `StockRepository` and `UserRepository` and were moved to parent `Repository`:

```
1 public void add(V value) {
2   repoContents.add(value);
3 }
4
5 public ArrayList<V> getAll() {
6   return repoContents;
7 }
```

where `repoContents` is an `ArrayList` that contains repository values. An example of Move Method is the following snippet:

```
1 public User getExact(String username, String password) {
2   return this.repoContents
3     .stream()
4     .filter(item -> item.getUsername().equals(username) &&
          item.getPassword().equals(password))
5     .findFirst()
6     .orElse(null);
7 }
```

That method allowed to remove similar method from `Controller` class, called `findUser`.

The following files were affected by the refactoring:

- src/main/java/com/keirnellyer/glencaldy/repository/Repository.java
- src/main/java/com/keirnellyer/glencaldy/repository/UserRepository.java
- src/main/java/com/keirnellyer/glencaldy/runtime/Controller.java
- src/main/java/com/keirnellyer/glencaldy/runtime/Model.java

To test this refactoring, I created a `UserRepositoryTest`, where I was able to test both `add` and `getExact` methods.

This refactor brought better structure to the code by removing duplicates and moving methods to where they actually belong.

### Refactor menu.option, extract UserTypeOption to separate class, introduce UserType and StockItemType enum, modify accessors (b6b0c98, 8a898a0)

This refactoring replaces hardcoded Strings with `UserType` and `StockItemType` enums, which allows us to reuse the values throughout the code, instead of having to wonder which values to use. On top of that, `UserTypeOption` and `StockItemOption` were extracted into separate classes, since it didn't make sense to keep them inlined.

Menu options relied on hardcoded values, which is a code smell, since such values could be potentially reused. Therefore, by using Extract Class technique, I extracted the values into enums with specific string values attached. For example, here's the `UserType` enum:

```
1  public enum UserType {
2    CASUAL("Casual"),
3    FULL("Full"),
4    STAFF("Staff"),
5    ADMIN("Administrative");
6    private String type;
7
8    UserType(String type) {
9        this.type = type;
10   }
11
12   @Override
13   public String toString() {
14       return this.type;
15   }
16 }
```

This enum has a specific string value attached to each of enum variables. By implementing these enums, I had to replace the parameter type for `UserTypeOption` to replace `String type` to `UserType type`.

This refactoring affected the following files:

- src/main/java/com/keirnellyer/glencaldy/menu/option/stock/CreateStockOption.java
- src/main/java/com/keirnellyer/glencaldy/menu/option/stock/StockItemOption.java

- src/main/java/com/keirnellyer/glencaldy/menu/option/stock/StockItemType.java
- src/main/java/com/keirnellyer/glencaldy/menu/option/user/CreateUserOption.java
- src/main/java/com/keirnellyer/glencaldy/menu/Option.java
- src/main/java/com/keirnellyer/glencaldy/menu/option/user/CreateUserOption.java
- src/main/java/com/keirnellyer/glencaldy/menu/option/user/EditProfileOption.java
- src/main/java/com/keirnellyer/glencaldy/menu/option/user/UserType.java
- src/main/java/com/keirnellyer/glencaldy/menu/option/user/UserTypeOption.java

These changes did not require unit testing since all that was changed is String values were replaced with the enums that have the same values.

This refactoring makes the code more readable and less prone to errors, like typos.

### Add abstract `processInput` method (ad3147)

This refactoring abstracts the `processInput` method that was declared twice in `CasualProperties` and `StaffProperties` by adding abstract `processInput` to `UserProperties`.

```
1 public abstract UserInfo processInput(InputResult result);
```

The code used duplicated method declaration and wasn't very object oriented, therefore, it was fixed by adding the `processInput`. This was done using the Pull Up Method technique.

The following files were changed as a result of this refactoring:

- src/main/java/com/keirnellyer/glencaldy/manipulation/user/CasualProperties.java
- src/main/java/com/keirnellyer/glencaldy/manipulation/user/StaffProperties.java
- src/main/java/com/keirnellyer/glencaldy/manipulation/user/UserProperties.java