University of Brighton

CI553 OBJECT-ORIENTED DEVELOPMENT AND TESTING

Software Project from Codebase

Ellie Antonowicz

10.01.25

Contents

Introduction

In this assignment I will be improving the given codebase ‘miniStore’, adding new features and improving its GUI. ‘miniStore’ is an online catalogue store, similar to ‘Argos’; users search for an item in the catalogue, then go to the till where the cashier checks the items are available and buys them. After all items wanted are bought, they get packaged in the packing area. In the program there is also a space for workers to check and add stock.

Objectives

I plan to implement the following features:

- Change the GUI colours
- Make the GUI larger
- Improve the basket (stack identical items and order items by product number)
- The ability to search by the name of products
- The ability to bulk buy
- Change Observer to PropertyChangeListener

If I am able to, I would also like to add more features like:

- Dark Mode
- Show all items available
- Sound effects
- Add more graphics to each window

Development, Management and Tools

Life-Cycle Models

Software has many different life cycle models. In reality, software will be continuously maintained and developed so use the model in figure 1. However, in my assignment once finalised, it will get no more updates, so I have decided to use a 'waterfall' life-style model (figure 2).

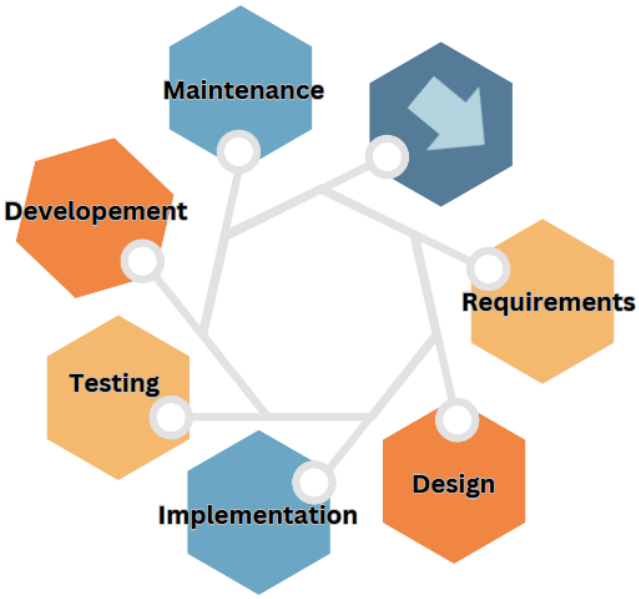


Figure 1

Requirements:
I spent a large amount of time understanding what the requirements for the codebase improvements were and how I would complete this assessment to a high standard.

Planning:
I planned how I would meet those requirements and what improvements I would make.

Coding:
The main development of the program. Adding new features, changing UIs and adding comments.

Testing:
Testing my program using different testing techniques including white box, black box and Junit testing.

Writing Report:
Putting all the components together, showing code, screen shots and reviewing my mini-Store.

Finalise:
Editing report and small edits to the code.

Hand in:
Release repository on GitHub and hand in final report to MyStudies.

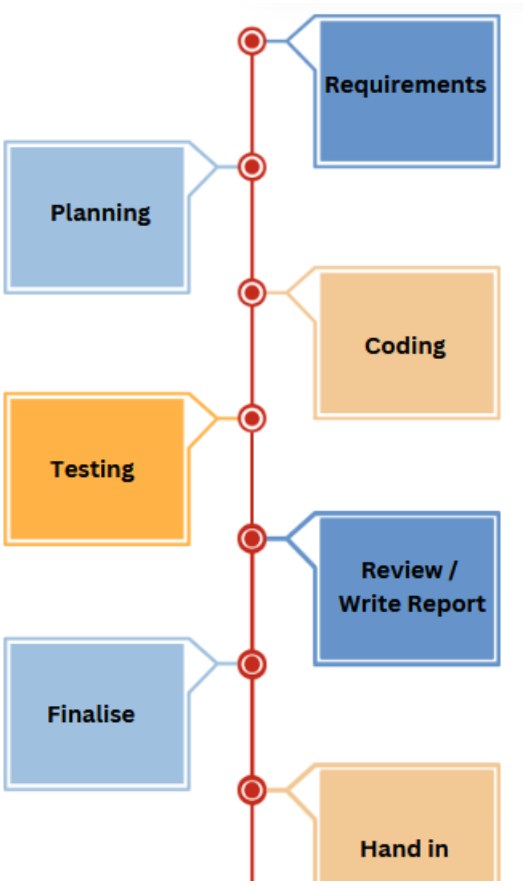


Figure 2

Development

GUI and Visuals

The program before any additions is shown in figure 5. The first thing I wanted to change was the GUI as I knew it would be a simple way to greatly change the feel of the program.

When deciding on what I wanted my store to be like, I wanted it to have a 'retro' aesthetic, I would achieve this by using colours and making the program simple.

I wanted to base the colour pallet off the graphic in figure 3[1]. The graphic is of a retro tech store, and I wanted to replicate what I thought its catalogue shop would look like. I tried multiple different palettes[2] (shown in figure 4), however I decided to stick with the original design from the retro tech store graphic and use blue as the main colour, cream as the text box colour and the orange for buttons.

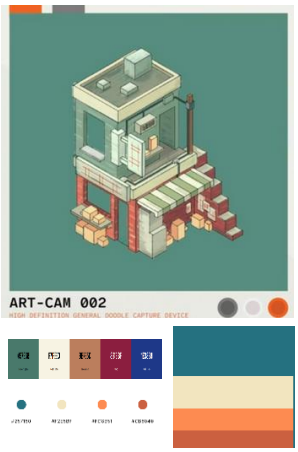


Figure 3

The code I used for each element:

```
cp.setBackground(new Color(37, 113, 128)); // Colour Teal
```

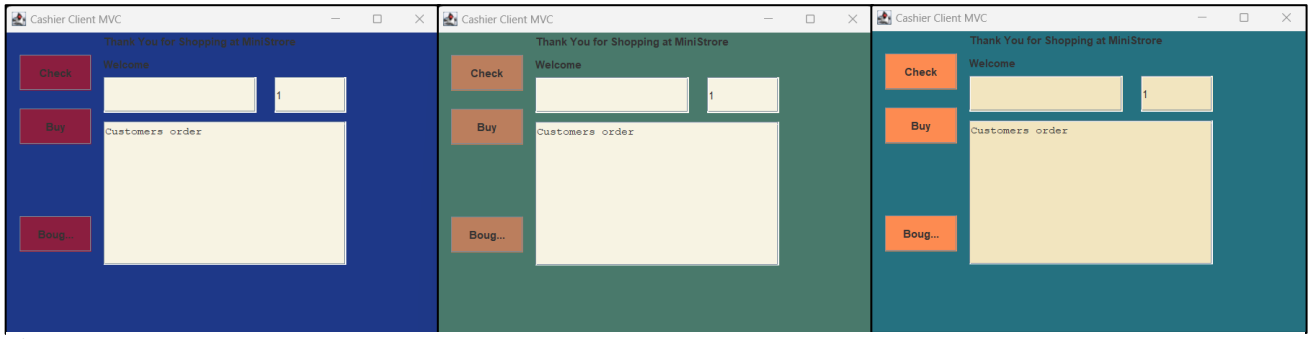
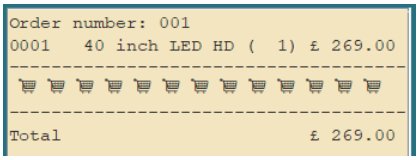


Figure 4

Another small visual change I made was the addition of some shopping trolly icons in the output/receipt. Not only does this make it more charming, but also separates the total price from the individual prices.



I also made the panes bigger (300x400 ➔ 375x500).

As shown below these small changes make a huge difference to the layout. And makes the program much more visually appealing.

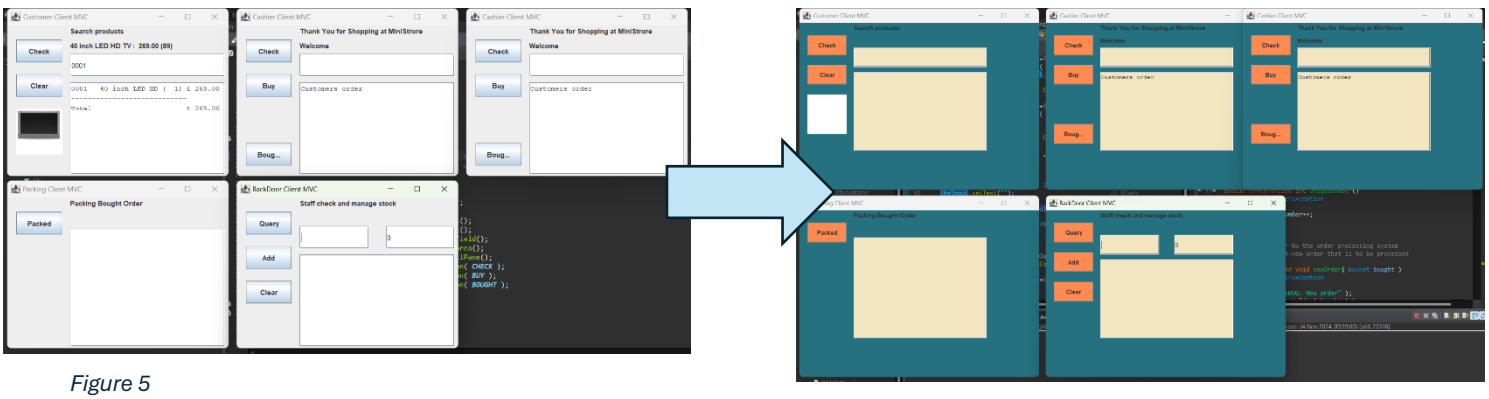
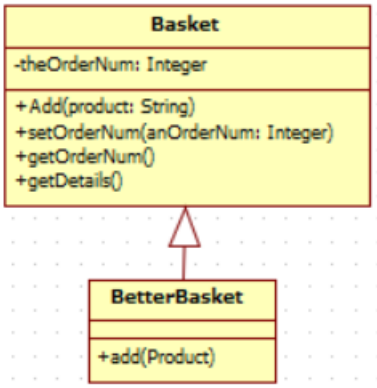


Figure 5

BetterBasket



Before any additions, Basket would keep items in an ArrayList, this works if the user only buys one or two items. However, if the customer buys more products, then the output quickly becomes very messy and hard to read. BetterBasket extends Basket to make multiple of the same products stack and shows them in order of product number. This makes it easier to see what items are in the basket and the quantity.


```java
package catalogue;

import java.io.Serializable;

/**
 * BetterBasket
 *
 * @author  Ellie
 * @version 1.0
 */

public class BetterBasket extends Basket implements Serializable
{
  private static final long serialVersionUID = 1L;

  @Override
  public boolean add( Product pr )
  {
      for(Product prInList: this) { // if product is the same as a product already in list then merge together
          if(prInList.getProductNum().equals(pr.getProductNum())) {
              int quantity = pr.getQuantity()+prInList.getQuantity();
              prInList.setQuantity(quantity);
              return (true);
          }
      }

    super.add( pr );     // Call add in ArrayList
    Collections.sort(this);  //sorts by num
    return (true);           // return
  }

}
```

BetterBasket.java

```java
protected BetterBasket makeBasket()
{
  return new BetterBasket();
}


protected Basket makeBasket()
{
  return new Basket();
}
```

CustomerModel.java and CashierModel.java

Search product by name

Prior to improvements, customers could only search for products by their product number. This is not user-friendly as most people wouldn't know any of the product numbers, let alone find a product they want. Searching for a product by name is a much more user-friendly way of finding items. To do this, I added a 'Search' button the customer pane and made the class NameToNumber. NameToNumber uses a HashMap[3] as an index so you can find one value using another value.

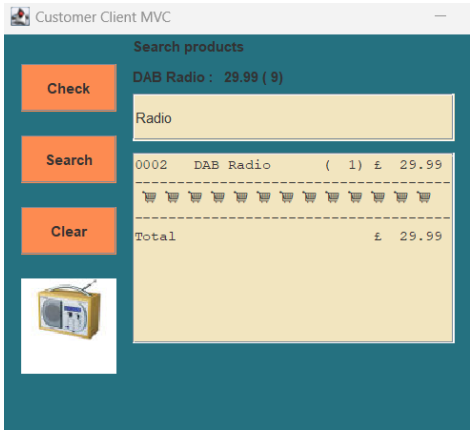
```
package clients.customer;
import java.util.HashMap;

public class NameToNumber extends HashMap<String, String> {

    NameToNumber() {
        put("0001", "TV");
        put("0002", "Radio");
        put("0003", "Toaster");
        put("0004", "Watch");
        put("0005", "Camera");
        put("0006", "Music player");
        put("0007", "USB drive");
    }

//https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
    public <T, E> T getNumberByName(Map<T, E> map, E value) {
        for (Entry<T,E> entry : map.entrySet()) {
            if(Objects.equals(value, entry.getValue())) {
                return entry.getKey();
            }
        }
        return null;
    }
}
```

```
// Name
 /**
  *
  * @param CheckByName - can search items using name
  * @author Ellie
  *
  */
public void doCheckByName(String name) {
    NameToNumber nameToNumber = new NameToNumber();
    String pn = nameToNumber.getNumberByName(nameToNumber, name);
    model.doCheck(pn);  }
```

```
case "doClear":
    thePicture.clear();          // clear picture
    theOutput.setText( null );   // clear text -
    break;
```

Clear

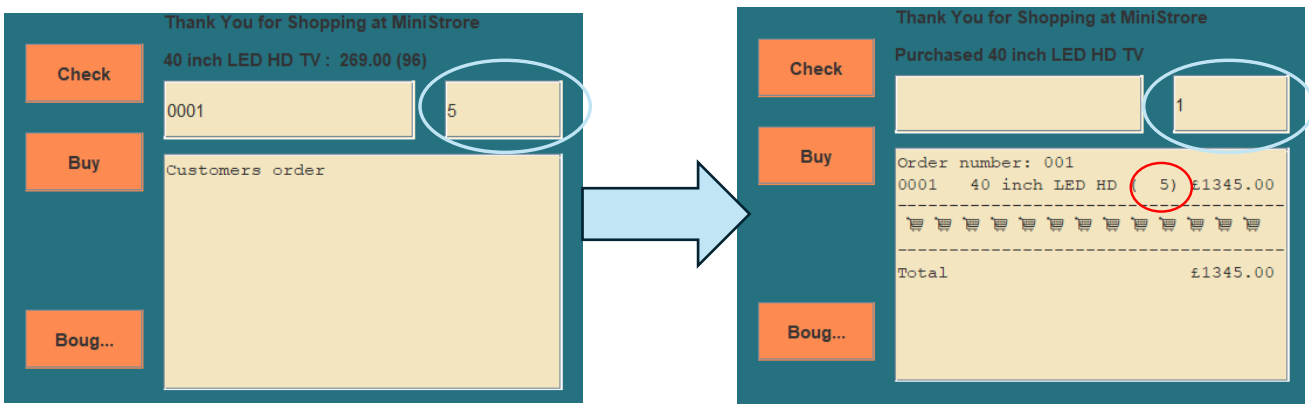
Made all outputs clear, as before only the image would disappear. At first, I tried using .clear but it didn't work. Eventually I found that a different section of code uses .setText("msg"); so I used this with null and it worked. When I added the search button, I originally forgot to move the Clear button down so it disappeared behind Search. I fixed this by moving the clear button and picture down using:




```
theBtClear.setBounds( 16, 25+60*2, 90, 40 );    // Clear button
theBtClear.addActionListener(                   // Call back code
  e -> cont.doClear() );
cp.add( theBtClear );                           //  Add to canvas
theBtClear.setBackground(new Color(253, 139, 81)); // Colour (Orange)
```

Buying multiple items

Previously, the program only allowed you to buy one item at a time, this would be very time consuming if a user wanted to bulk buy. I added a feature that allows customers to enter the amount of each item that they want, the display will reset back to one when they buy it.



```
//                     x   y  length  hight
buyQuantity.setBounds( 300, 50, 80, 40 );         // Input Area
buyQuantity.setText("1");                           // Blank
cp.add( buyQuantity );                             //  Add to canvas
buyQuantity.setBackground(new Color(242, 229, 191)); // Colour (Beige)
```

Changing observable to propertyChange

Behavioural patterns manage standard ways for classes to communicate with one another. Observer pattern was in the code base, this allows links to be made between objects without needing both to be dependent on each other.

This needs to be changed to PropertyChangeSupport as Observer is now deprecated.

I was only able to implement in customer, even after multiple attempts I was unable to get it working on any other clients (All code that I attempted is left in the code but is commented out).

```
public void update( Observable modelC, Object arg )
{
  CustomerModel model   = (CustomerModel) modelC;
  String        message = (String) arg;
  theAction.setText( message );
  ImageIcon image = model.getPicture();  // Image of product
  if ( image == null )
  {
    thePicture.clear();                  // Clear picture
  } else {
    thePicture.set( image );             // Display picture
  }
  theOutput.setText( model.getBasket().getDetails() );
  theInput.requestFocus();               // Focus is here
}
```

```
@Override
public void propertyChange(PropertyChangeEvent evt) {
    String proName = evt.getPropertyName();
    String oldValue = (String) evt.getOldValue();
    String newValue = (String) evt.getNewValue();
    theAction.setText(newValue);
    switch(proName) {
    case "doCheck":
        ImageIcon image = model.getPicture();
        if ( image == null )
        {
          thePicture.clear();                  // Clear picture
        } else {
          thePicture.set( image );             // Display picture
        }
        theOutput.setText( model.getBasket().getDetails() );
        theInput.requestFocus();               // Focus is here

        break;

    case "doClear":
        thePicture.clear();          // clear picture
        theOutput.setText( null );   // clear text -
        break;
    }
```

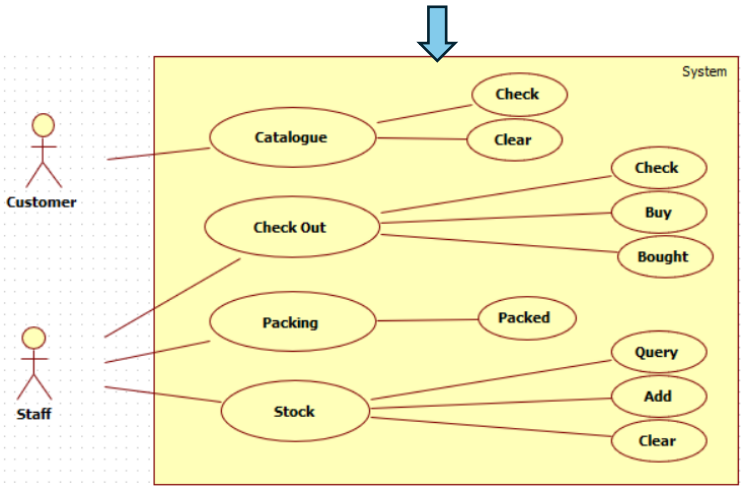
My code can be viewed on GitHub using this link:
https://github.com/EllieAnt/CI553-miniStore

Original code base can be found here:
https://github.com/Shine-SJF/CI553-CW-miniStore

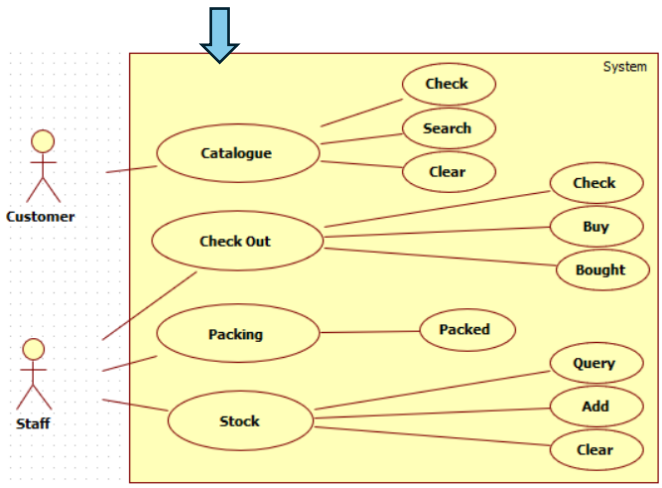
Unified Modelling Language

Unified modelling language (UML) is a language for specifying, visualising and constructing the diagrams and documentation that describe a software system[4]. UML helps designing object-oriented (OO) systems and is an industry standard for OO modelling. UML can be informal like a sketch or be a complete blueprint. For my assignment I used WhiteStarUML to make use case, class and sequence diagrams.

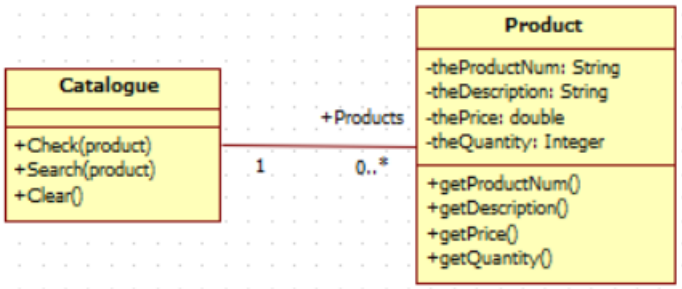
Use case Diagrams: Starting Codebase

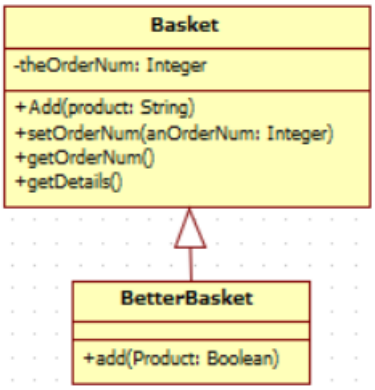


Final Codebase

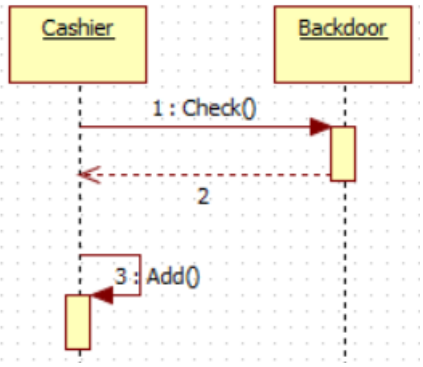


Class Diagrams:





Sequence diagram:

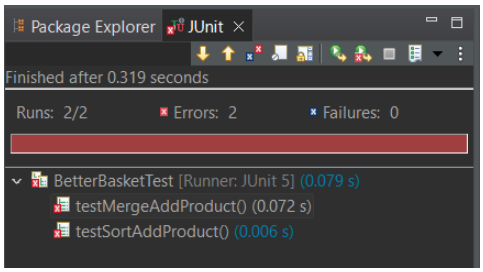


Testing

Testing is an important step of software development as it shows a system works how intended and will detect errors before deployment. Tests only show the presence, not the absence of errors.

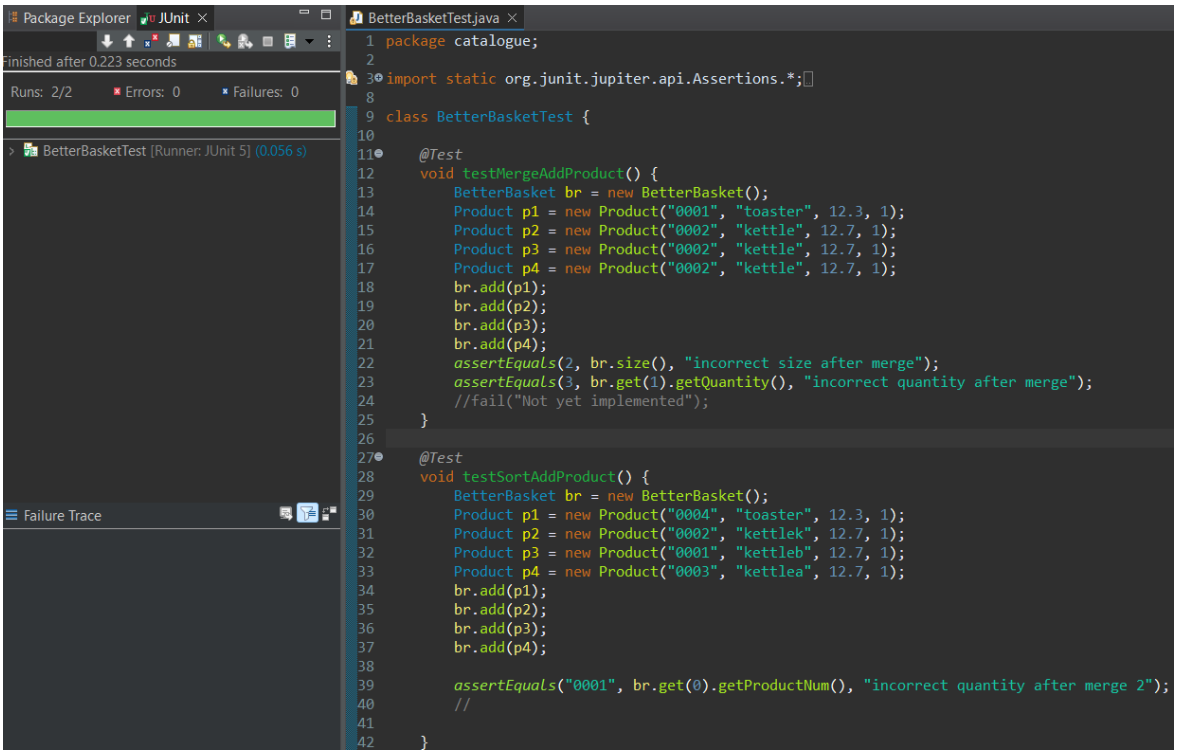
Automated testing

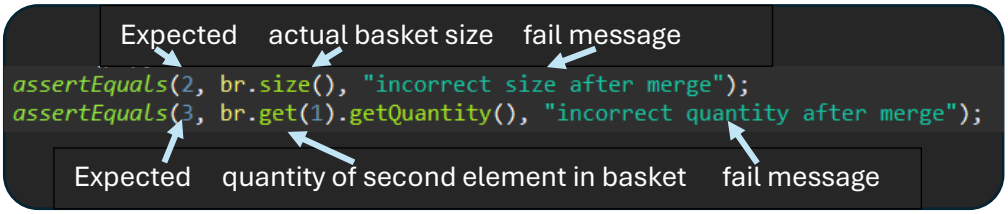
I used Junit[5] to test BetterBasket, to do this I wrote a script that tests the software. When testing I used artificial data to make sure the basket does what is intended to do. In this test I created 4 products (one toaster and three kettles) and added them together (Creating toaster: 1 and kettle: 3).



assertEquals(expected, actual, message)

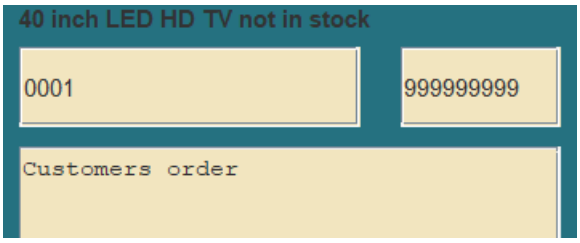
If expected and actual are not equal then the failure message will send. For example in testMergeAddProduct() assertEquals(3, br.get(1).getQuantity(), “incorrect quantity after merge”); means if quantity of the second item in the basket is not 3 then fail.





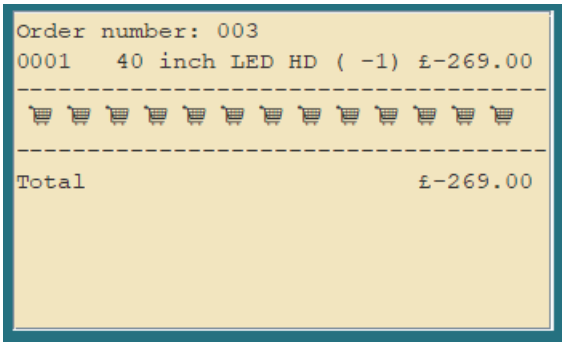
Manual testing

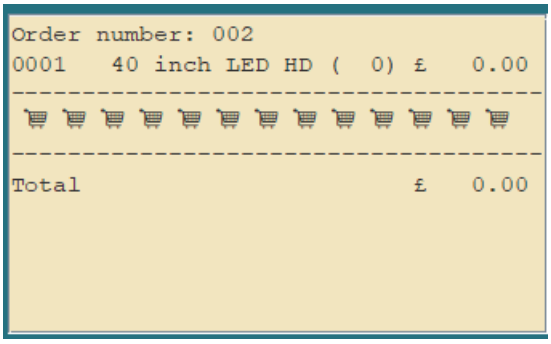
<u>White Box testing</u>



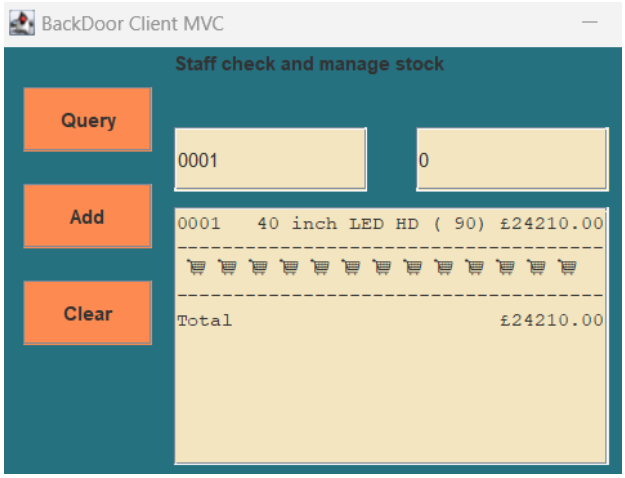
After each addition I would test the part of the program that I had changed to make sure it worked. Once I had finished with additions, I manually tested the whole system. In this example, I tested extreme values, when trying to buy 999999999 TVs, a message is displayed that says, “not in stock”.

In the cashier client, if you try to buy zero or negative amounts of items then the order will still go through.



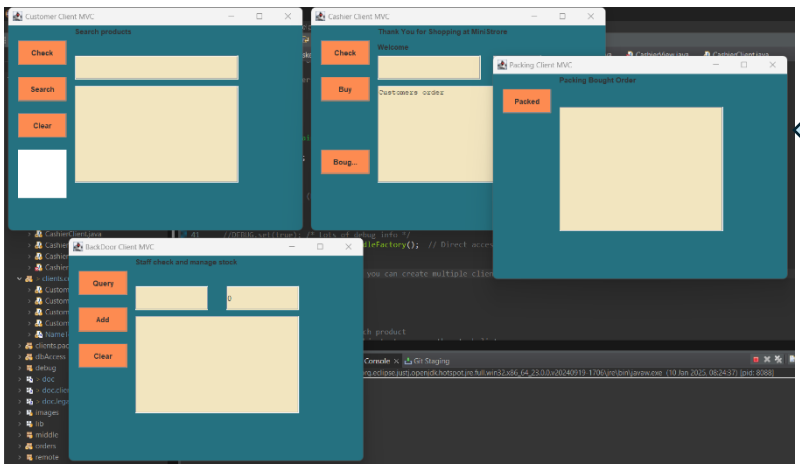


Another bug I found was that when adding stock, if you try to add zero items then it will automatically add 90. But if you try to add -1 then a message will show stating “Invalid quantity”.



<u>Black Box testing</u>

I gave my program to a tester who is not on this course so would not have any knowledge about the codebase. Their feedback was “I like the simplicity and the receipt design, but I don’t like that the two cashier clients overlap”. After this feedback I tried to move the second cashier window and tried deleting the second window all together, however it still did not look neat, so I decided to keep it as it is.



What the screen looks like without second cashier

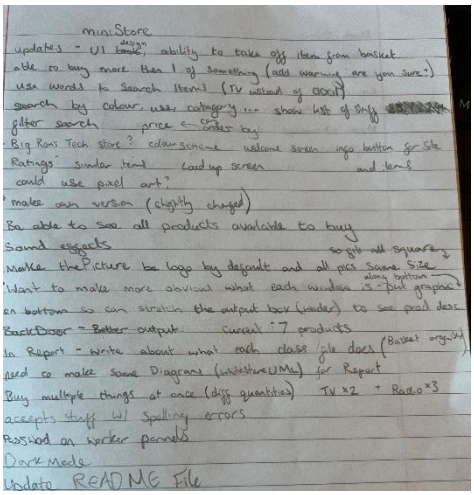
Documentation

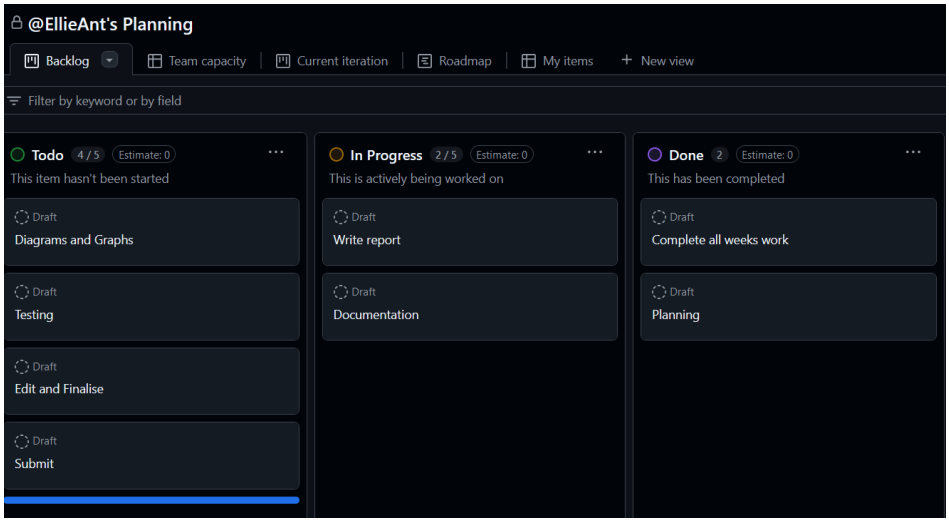
I used JavaDoc to document my code, as well as standalone tags in comments such as @author and @param.

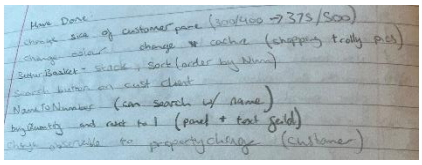
```
/**
 * BetterBasket
 *
 * @author  Ellie
 * @version 1.0
 */
```

Planning

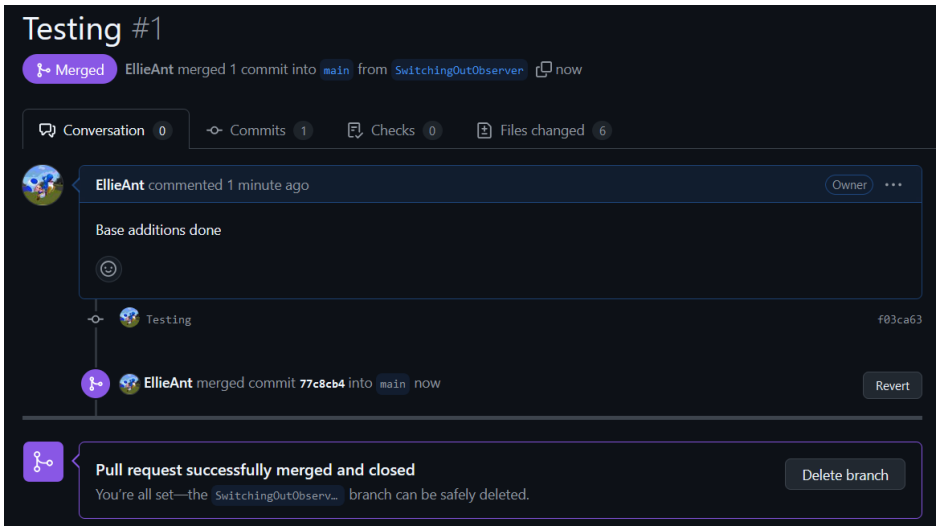
Planning is an important part of the development process; I did most of my planning in my notes as for me, I prefer to be able to quickly write stuff down instead of having to load software to plan an idea. After I have written my plan in my notes, I used GitHub Project tool that allows me to add tasks to a list and move them once they are completed. This tool is great for keeping track of what has been done, and what needs to be done, especially if it's a group project. Alternative tools for planning include Tello and google docs, but I decided for this individual project, its most efficient to use my notes and GitHub to keep everything in one place.

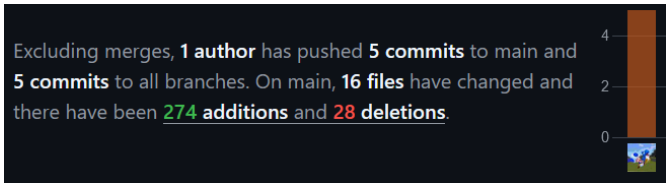


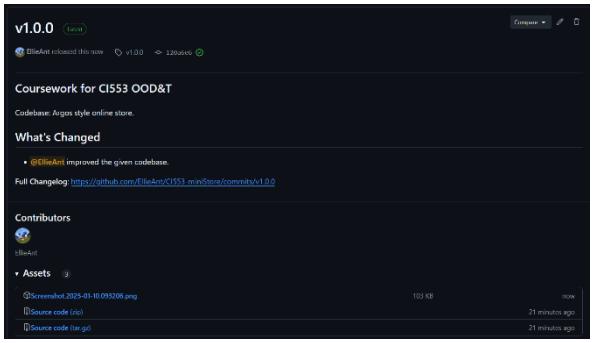




Management and Delivery





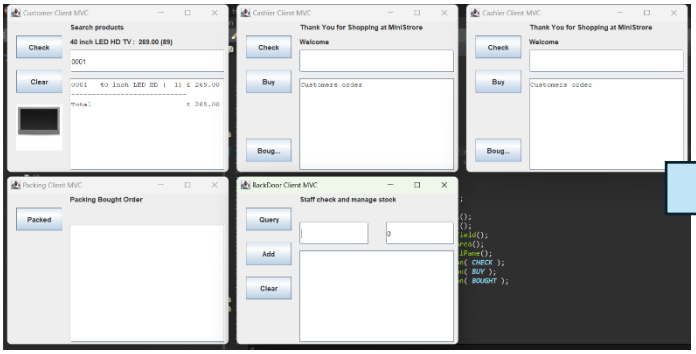


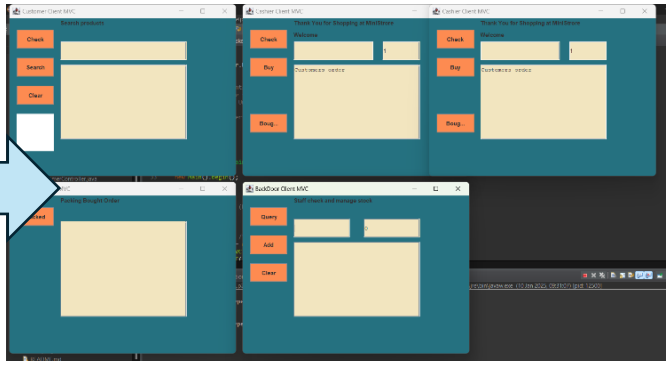
Using GitHub allows me to keep my code safe in a centralised online location and allows me to use version control which is helpful for if I want to revert my code back to when it was last working.

GitHub also allows me to keep track of branches, and merge branches once complete.

Codebase given

Final mini-Store





Critical Review

Positive features

Styling

I am happy with the changes I made to the GUI as it looks a lot nicer than the codebase and fits the theme I wanted. If I had more time, I would add unique graphics along the bottom of each window so you can tell just by looking at it what part of the program it is (catalogue, till, packing, stock). I would have also liked to add a welcome screen with music.

Basket Improvement

BetterBasket was the most significant improvement I made to the program. Being able to merge items together made the program much more usable.

Areas of Improvement

Test Driven Development

Test Driven Development[6] (TDD) is a software development method where you write tests before writing the code. This ensures code is always functional, high quality and reduces bugs.

More features

I would have liked to add more features, however due to time limitations and code breaking I wasn't able to add them. Ideas included: dark mode, being able to see all products available to buy, sound effects and a password for worker windows.

Observer

As stated in the 'Changing observable to propertyChange' section, I was not able to change observer in all the clients.

Conclusion

This was the first time I have coded in a program this large, so at first this was a very intimidating assignment which I kept putting off. However, as I worked on the program more, I started to understand it better and could navigate around it quicker. I learnt a lot about GitHub and using eclipse as this was my first time using eclipse and using GitHub in this way. I now understand object-oriented programming, development and testing a lot better than I did prior. I have never used UML before and enjoyed using WhiteStarUML, it was a lot more enjoyable and useful then I thought it would be.

I found this project challenging, but if I were to do it again, I would be able to tackle it a lot better and would be more confident in my abilities.

Estimated Grade

Development (40%): B

Tools (20%): A

Management (20%); B

Report quality (20%): B

References

Shine-SJF (2024). *GitHub - Shine-SJF/CI553-CW-miniStore: CW codebase for CI553*. [online] GitHub. Available at: https://github.com/Shine-SJF/CI553-CW-miniStore.

1. Mumbo Jumbo (2024). *Hermitcraft 10: Episode 13 - FIRST BASE BUILD!* [online] YouTube. Available at: https://www.youtube.com/watch?v=ay8ZnJe48rY [Accessed 10 Jan. 2025]. Time Stamp: 3:08.
2. colorhunt.co. (n.d.). *Color Hunt - Color Palettes for Designers and Artists*. [online] Available at: https://colorhunt.co/.
3. W3schools (2019). *Java HashMap*. [online] W3schools.com. Available at: https://www.w3schools.com/java/java_hashmap.asp.
4. Shan, S. (n.d.). *Lectures*. University of Brighton
5. Junit.org. (2017). *Assertions (JUnit 5.0.1 API)*. [online] Available at: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html#assertEquals-int-int-java.lang.String- [Accessed 10 Jan. 2025].
6. Bhakhra, S. (2020). *Test Driven Development (TDD)*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/test-driven-development-tdd/.

Find original codebase here:

Shine-SJF (2024). *GitHub - Shine-SJF/CI553-CW-miniStore: CW codebase for CI553*. [online] GitHub. Available at: https://github.com/Shine-SJF/CI553-CW-miniStore.

My code can be found here:

EllieAnt (2025). *GitHub - EllieAnt/CI553-miniStore: Course Work codebase for CI553*. [online] GitHub. Available at: https://github.com/EllieAnt/CI553-miniStore [Accessed 10 Jan. 2025]. Available at: https://github.com/EllieAnt/CI553-miniStore.