Software Development

(CS12420)

Individual Project 2012

“AberPizza” Till System

Connor Luke Goddard

Student No: 110024253

Aber ID: clg11

Table of Contents

Introduction

In this project I have been set the task of taking an initial UML class design for a till system, and using the provided use-case and class diagrams, extend and complete the design before implementing it using Java code to fulfil all of the use-case requirements specified. I must then thoroughly test my completed system using both automated 'JUnit' tests for the data classes, and test tables for the GUI to ensure that the system fulfils all its requirements and is stable enough to be used in the 'AberPizza' store.

In this document, I will describe and discuss the additions/changes I have made to the initial class design to ensure that all the requirements in the use-case diagram are met. I will also describe my design for the GUI and explain my reasoning for it, and I will also explain the tests I chose and why before finally providing a self-evaluation of how I felt the project went as a whole, and any areas I found particularly difficult/easy and why.

Assumptions

I have made some assumptions about the required system and its specified data when designing the final design for the software. These assumptions are declared here:

- All products (Pizza, Drink and Side) can have a size option (Small, Medium or Large). I am aware that Pizza is explicitly required to have this option, but as Drink and Side were not I have included this option for these products to.

Notes

- Please be aware that the justification for my chosen designs has been incorporated into the class /design approach descriptions.

Changes to the Initial Design & Approach

After analysing the existing design and identifying the main requirements of the system, I have produced updated class diagrams that detail all of the classes that will be built and used to allow the system to fulfil its requirements:

Data Model:

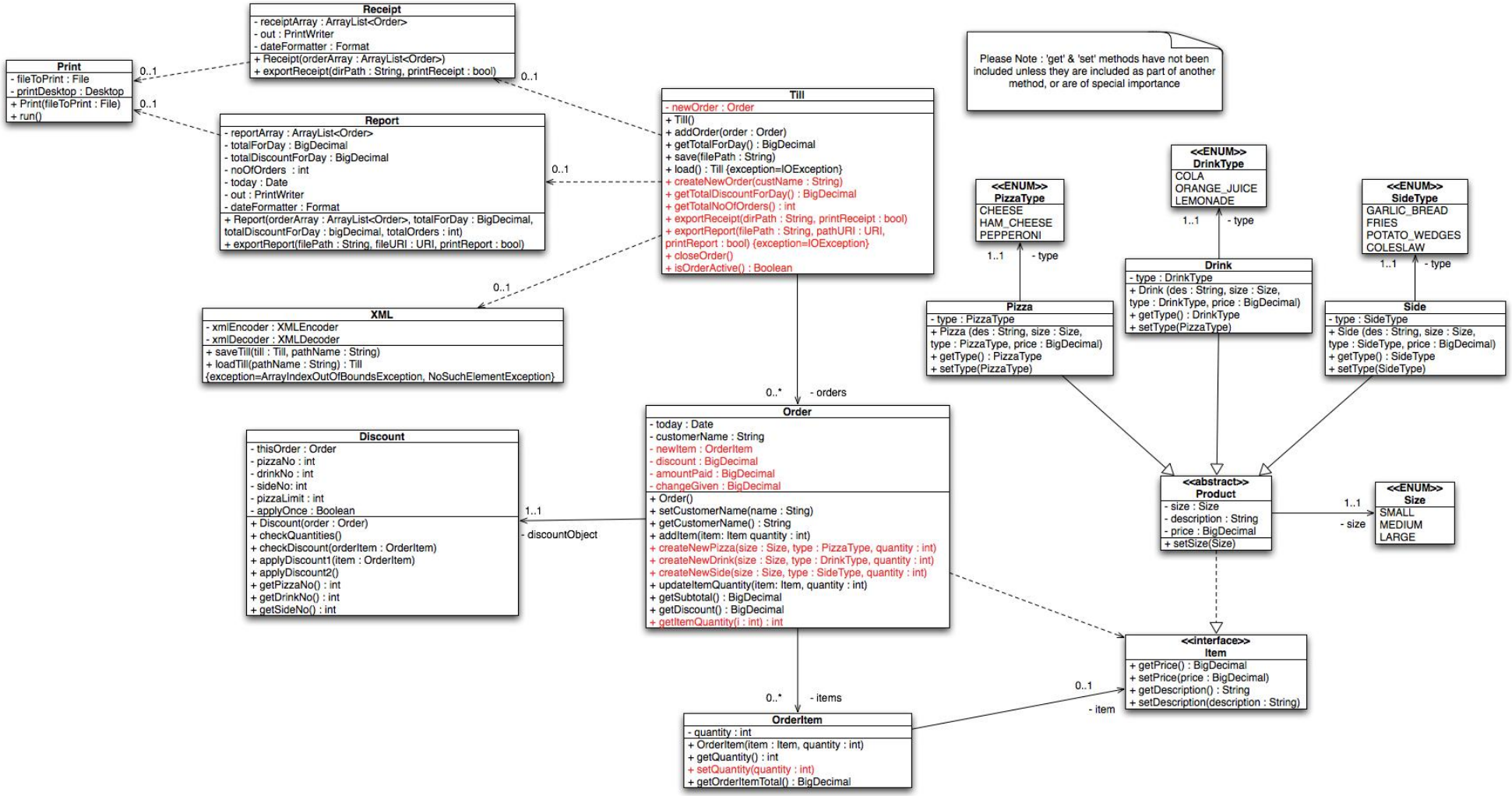


Figure 1.1: Application Data Model Class Diagram

(N.B. Items highlighted in RED are additions to the existing classes in the initial class design)

Till

The most important change to this class is the addition of a 'newOrder' object. This object (which is of type Order) is used to represent a new order in the system, and is created when a new order is created by the user using the GUI. This 'newOrder' object is then populated with the items selected by the user and has its 'subtotal' and 'discount' values updated accordingly. If the user wants to change the quantity of an item, this is always done to an item inside the 'newOrder' object. When the user tenders payment, the 'amountPaid 'and 'changeGiven' values inside 'newOrder' are updated, giving detailed information displayed on the receipt. Finally, the 'newOrder' object is saved in the 'orders' arraylist and then set as 'null' ready for a new order.

The class also contains a 'printReceipt' method, which loops through the arraylist of Orders and outputs a receipt using the data contained within each Order object using a pre-defined HTML template, and a 'generateReport' method which again loops through all stored orders and generated a report detailing totals of the orders and breakdowns using a pre-defined HTML template.

Order

This class contains an object similar to the 'newOrder' object in the Till class. 'newItem' is an object of type OrderItem and is populated with a new item (either Pizza, Drink or Side) and a quantity using either the 'createNewPizza', 'createNewDrink' or 'createNewSide' methods when a user adds an item to the 'newOrder' object.

These three methods respectively create a new object of type Pizza, Drink or Side, and then set size and type of that item setting the 'newItem' object with the item and a quantity, before finally adding that 'newItem' object to the 'items' arraylist of OrderItems and then setting the object as null, ready to take another item. 'newItem' therefore essentially acts as a 'template' for adding a new item to 'items' using the three 'createNew..' methods.

This class also has a link to a Discount class. This link is represented by the variable 'discountObject' which calls the Discount class every time a new item is added to 'items' to determine whether a discount should be applied with the addition of the new item to the order.

I have removed the 'getReceipt' method from this class, and instead placed the method 'exportReceipt' in the Till class. My reasoning for this was that I felt it would be better to perform a "bulk" export of all order receipts (including the newly saved order), as this would mean that if any changes had been made to any existing orders, the receipt for that order and any other orders would be automatically updated just by running this one method, as opposed to having to access every order you wanted to update the receipt for and then running the 'getReceipt' method.

OrderItem

There have been very few changes to this class. The only addition was the 'setQuantity' method which is by the 'updateItemQuantity' method in the Order class to update the quantity of a particular item in an order.

Item (Interface)

Nothing has been changed in this interface. The methods have been implemented by the Product abstract class to allow the different item classes to utilise these common methods.

Product (Abstract)

This abstract class implements the Item interface and is used to hold all the common methods used by each of the three item classes (Pizza, Drink and Side).

These common methods are all of the methods implemented from the interface, with the addition of 'setSize' and 'getSize' methods, which use the 'Size' enumeration class to determine the size of an item which is common between all three different items (See introduction for assumptions).

Pizza/Drink/Side

These three classes are very similar to each other in many ways. They all extend the Product abstract class, allowing them to use all of the common methods in that class; they all represent a particular product (Pizza, Drink or Side respectively), and they all have an additional method that allows the type of the product to be set.

The difference between the classes is the available types that can be used to set the type of a product. As you can see from the class diagram, each class (Pizza, Drink Side) has a link to type enumerations (PizzaType, DrinkType, SideType) respectively. The three classes then use the 'setType' method to set the type of the Pizza/Drink/Side using their respective enumeration class.

By using this approach, individual items cannot be set an incorrect type (e.g. a drink could not have a Pepperoni type set), and also as different products will have different prices, it means that when it comes to setting prices for different products, the system can use a simple *"instanceof"* function to easily determine the product that the item represents, and then set the price accordingly.

Discount

An instance of this class is created by the Object class, and is called every time a new item is added to the 'items'. Its purpose is to determine whether a discount should be applied to the order (using rules contained in methods – specified by the user) and if so, applies this discount (specified in methods) before updating all the item prices and order subtotal accordingly.

The class keeps track of how many of Pizza/Drink/Side objects are stored in the 'items', using the 'checkQuantities' method, and then using the 'checkDiscount' method, uses this data and the information from the new item (parameter) to determine if a discount should be applied using rules specified by the user. If a discount should be applied from one of the rules, then the relevant discount is applied to the order using 'applyDiscount1()/2/etc…'.

The class also has the ability to prevent a particular discount being performed more than once per order using the 'applyOnce' boolean, which is check to see if the particular discount has been applied, before attempting to apply it to the order.

Receipt

The purpose of this class is to generate, and output the bespoke receipts for each of the orders stored in the 'orders' in the Till class. An instance is created in the Till class and is called using the 'printReceipt' method.

The class takes the array list of orders 'orders' as a parameter, and uses this array list to loop through each of the Order objects, and their respective items which are then populated and outputted to separate '.html' files (for each order) using a pre-defined template, producing a professional-looking receipt that can be viewed by a web browser, and printed off as required.

The data that is exported in the receipt includes the time, date, customer name, all order items (including description, size, type and price), an order subtotal, any discounts applied, the amount tendered, and any change that was given.

Report

This class is designed to generate the administrator 'end of day' report that details all the stored orders in the system, and gives an overview of all the money taken, the total discount applied, and the total number of orders, and output it as an easy-to-read '.html' file using a pre-defined template.

Similar to the Receipt class, it takes the array list of orders 'orders' from the Till class as a parameter, which it then loops through to obtain each Order object and its respective data to produce a "breakdown table" of all the orders.

However, it also takes three more parameters, 'totalForDay', 'totalDiscountForDay' and 'totalOrders'. These parameters contain total money taken, total discounts applied, and the total number of orders (calculated by the 'getTotalForDay()' , 'getTotalDiscountForDay()' and 'getTotalNoOrders()' methods inside the Till class) which are then (along with the order breakdown) are outputted into a single '.html' file using a pre-defined template.

XML

The importing and exporting of the Till object is dealt with by this class. Its purpose is to provide the functionality to export, and import the entire system to/from an '.xml' file, which will provide the 'save/load' functionality required by the use case diagram.

"Saving" is achieved by calling the 'saveOrders' method which takes a Till object and a pathname string as parameters. This method then takes the current Till object (which contains the 'orders' (which subsequently contains all order items and data)) and passes it into an XML encoder 'xmlEncoder' which serialises everything and saves the resulting file into the directory as specified by the 'pathname' string.

"Loading" is achieved by calling the 'loadOrders' method when the program starts. This method (which takes a pathname as a parameter) looks in the location specified by the 'pathname' variable, and tries to find an '.xml' file containing the serialised Till object.

If found, the method then returns a new Till object that has been created using the data inside the 'xml' file, which is then used as the Till object for the program. If no '.xml' file is found, or it is not a serialised Till object, then the method returns a new empty Till object which is then used.

This approach to saving/loading the Till is very robust, and allows the entire till system and all its data to be exported very quickly and easily from one place.

Print

Used in conjunction with the Receipt and Report classes to send the newly-created HTML files to print using the host system's printer spooler. The class makes use of the *Desktop* class available in the Java API which allows the application to run specific operations using the host system's default application for that particular operation (e.g. opening a HTML file using the default web browser or in this case, printing a file using the default printer service).

This class is always run using a dedicated thread created by the Report or Receipt classes as required. Using a dedicated thread allows the intensive task of sending a file to print to be run concurrently, preventing the GUI from becoming 'un-responsive' while the operation is carried out, and therefore improving the robustness of the application as a whole.

Graphical User Interface:

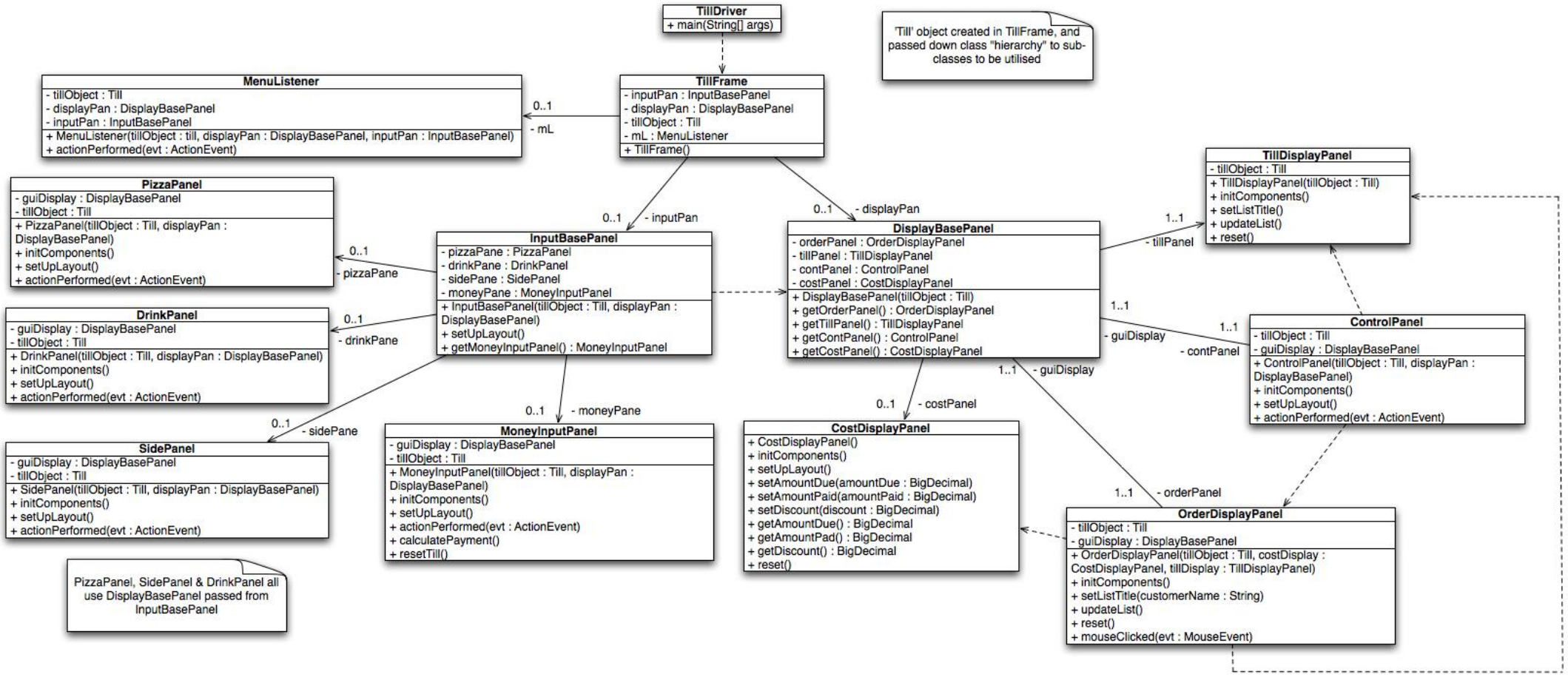


Figure 1.2: Application GUI Class Diagram

TillDriver

Bootstrap for the application. Contains the 'main' method that creates an instance of the main GUI frame, TillFrame.

TillFrame

Main GUI frame in which the panels that form the GUI are contained. The two "base" panels are created here and added to the WEST, and EAST sections of the frame.

The main Till data class is created here, and is either created using data from an existing '.xml' file, or from new. This Till object is passed as a parameter to both the InputBasePanel, and the DisplayBasePanel to allow sub-panels to use the class methods.

The application menu bar is created here also, and so as a result a new MenuListener class is created to allow the menu options to become interactive.

InputBasePanel

One of two 'base' panels designed to contain all sub-panels that allow a user to input information into the application. The four input sub-panels are created here (PizzaPanel, DrinkPanel, SidePanel, and MoneyInputPanel), and all have the Till object (passed down from TillFrame), and the DisplayBasePanel passed as parameters.

Passing the Till object to the sub-panels as a parameter allows them to interact with the main Till data class, which is vital as they are all used by the user to perform tasks within the system, such as adding items to an order, or tendering money and calculating change before closing an order and resetting the till. The panel also passes the DisplayBasePanel (passed down from TillFrame) to the four sub-panels, which is done so that the panels are able to interact with the rest of the GUI once the data manipulation tasks have been completed (e.g. updating the sub-total display once a new item is added to the order).

DisplayBasePanel

Second of the two 'base' panels, that is designed to contain all the sub-panels that display the order, transaction and general till information to the user. The three main display panels (TillDisplayPanel, OrderDisplayPanel, CostDisplayPanel) are created here, with the addition of a general control panel for the application (ControlPanel).

It may seem odd as to why I have placed a panel that accepts user input into the DisplayBasePanel. The reason I have chosen to do this however, is because it provides (I feel) a more intuitive interface by keeping the main controls for the till system away from the more specialist controls contained within the InputBasePanel which will be used quickly when taking an order, and helps to prevent buttons that do more 'system-wide' tasks being pressed accidentally when taking an order quickly, and so helps prevent orders being accidentally cancelled or the system exited during an order.

It is also very easy to provide access to the Till class from the ControlPanel by simply passing it from the TillFrame, into the constructor of ControlPanel (in the same way as InputBasePanel). This is also

the case for the TillDisplayPanel, and the OrderDisplayPanel, as they both require access to the Till object to allow them to display order and item information stored within.

PizzaPanel/DrinkPanel/SidePanel

Contained within the InputBasePanel, the purpose of these panels are to provide the GUI elements to give the user the ability to select the size, type, and quantity of a new pizza, drink or side respectively, before automatically calculating the price and adding the new item to the array list of items (*items*) in an order.

The panels consist of two *JComboBoxes* (one to select a type, and one to select a size), a *JSpinner* (to set the quantity) and finally a *JButton* (which calls the 'createNewPizza/Drink/Side()' methods inside the Order class). The 'Size' and 'Type' combo boxes are populated by arrays that contain the Size and Pizza/Drink/SideType enum entries respectively using the 'values()' method of those enum classes. The 'Quantity' spinner is populated using a *SpinnerNumberModel* that allows a starting value, and minimum value, a maximum value, and a step value to be set, giving complete control over the available quantities of items.

To access the data methods, the panels take a Till object as a parameter, this is provided by the InputBasePanel which passes the Till class created in TillFrame to the panels so that the same Till object is being used universally when adding items.

The panels contain an *ActionListener* that provides the functionality to the button when creating a new item. I chose to implement separate listeners inside the panel classes, as opposed to having a generic listener a separate class, because the 'createNewPizza/Drink/Side()' methods take a different enum for the item type (either PizzaType, DrinkType or SideType) which would have resulted in compatibility issues if I had attempted to use the same listener for all three panels.

The 'SpringLayout' layout manager is used to position the components as it allows for precise positioning on the panel which can be easily modified if required, without setting absolute values for the component positions.

MoneyInputPanel

Also contained within the InputBasePanel, this panel is used to enter amounts of money that a customer gives to pay for an order.

The panel consists of a number of JButtons, of which the majority make up a "number panel" to allow precise monetary values to be entered, another three are used to represent more typical values of money (£50, £20, and £10), and the final button is used to input this amount into the system, allowing the tendering calculations to take place.

I decided to include a "number panel" in the GUI because after doing some research, I discovered that most till systems nowadays are run on touch screen machines, and so I felt that by having this number panel, the till operator would be able to still use the application, even if there is no physical keyboard available.

Using data obtained from the Till object (again a parameter passed from InputBasePanel), this panel uses the 'calculatePayment()' method to determine whether the amount of money entered

by the user is more, or less than the order subtotal, and will then perform the appropriate operation, which would be subtracting the amount of money entered from the subtotal, and then either displaying how much more money is due from the customer, or displaying the amount of change that is owed to the customer, before closing the order and resetting the till.

CostDisplayPanel

Contained inside the DisplayBasePanel, this panel is responsible for displaying the financial information for the current order including the subtotal, the total discount applied, and the amount paid by the customer.

It is the only panel in the GUI that does not use the Till object that is created and passed down from TillFrame. This is because the panel is accessed from the other panels that make up the GUI (including those contained in InputBasePanel) via the DisplayBasePanel, and is updated accordingly by those panels as an order is changed and updated.

OrderDisplayPanel

This panel is responsible for two separate tasks. The first is to display a list of all the items currently in an order (taken from the 'items array list inside the Order class) which is updated every time an item is added or removed. This is very important as it gives the till operator the ability to be able to view exactly what items have been added to an order, and remove, change any items that are incorrect or not required as needed.

The second task, which is exclusive to this panel, is to update the quantity of any items in the current order to a new value entered by the till operator. This is achieved by using the 'updateItemQuantity' method within the Order class. The till operator is able to 'double click' on any item in the list box, resulting in an input dialog appearing, which allows them to enter a new value. The panel then calculates the position of the selected item in the 'items' array list before running the 'updateItemQuantity' method which takes the selected item from the 'items' and the quantity value entered and updates the quantity of that item. Finally, the panel updates the TillDisplayPanel, and the CostDisplayPanel to reflect the change of quantity.

I feel this is a good approach to updating an item quantity, as it provides a quick, easy and intuitive way for the till operator to update the quantity of an item, which is very important when the system needs to be used quickly if there were many customers waiting to be served. It also is not very 'resource intensive' ensuring that the system performs quickly.

TillDisplayPanel

Also contained within the DisplayBasePanel, this panel is responsible for displaying the financial information for each item in the current order. This differs from the information OrderDisplayPanel displays, because instead of displaying the detailed information for each item (e.g. the type and size), it instead simply displays the item name, quantity and price.

This information is displayed in a *JScrollPane*, which encapsulates a *JList* object, that loops through the array list of items for the current order, and from that 'Item' object, pulls the description, quantity and price before adding it to the *JList* model and displaying it inside the *JScrollPane*.

I decided that having a separate panel to display price information for order items would be useful to the till operator, as it means that instead of them having to analyse and 'pull out' the price from one large list box with all item information, they are able to quickly look at the list box contained within this panel, and find the information a lot more quickly, improving serving times to a customer. I have encapsulated the list box in a scroll pane, so that if the number of items in the order exceed the viewing size of the list box, the till operator can scroll down and view these items, preventing the till operator thinking that the items had not been added, and so adding more items unnecessarily.

ControlPanel

This panel is different to all the other sub-panels of DisplayBasePanel as it does not display any information to the user. The purpose of the panel is to provide the 'system-wide' or 'generic' operations that are required to allow the user to use the application.

These operations include creating a new order, exporting receipts and saving the till system to file. I felt that ensuring the most generic operations (such as creating a new order) were easy to access in the GUI was a good idea, again to improve productivity and speed when using the system, and I feel this panel provides an intuitive, and 'easy-to-program' way to do this, and can be easily modified and expanded as required in the future.

The panel consists of *JButtons* that allow the user to perform these tasks when clicked. These buttons are linked to an *ActionListener* contained within the panel itself, which accesses the Till object (passed as a parameter from DisplayBasePanel) to perform the appropriate operations.

MenuListener

This class is used by the TillFrame to provide an *ActionListener* for the *JMenuBar* contained within the frame. The listener has access to the Till object created inside TillFrame, to allow it to use methods which can use and manipulate the data inside the system, such as creating a new order, or generating an 'end of day' report. It also has access to both the GUI base panels (DisplayBasePanel and InputBasePanel) which allow it to use and update information displayed in the GUI, and use data that is entered by the till operator via the GUI.

I decided to use this separate class for the menu bar listener, because I felt that it would make a more robust system to distribute the responsibility of listening for actions to a separate class instead of having the TillFrame (which is the "top-parent" class that everything else "spans from") trying to perform too much in one go, and so I felt it was best to try and reduce the work-load on the class as much as possible to improve reliability.

GUI Breakdown Analysis

The following diagrams and images provide a breakdown of the GUI to show how the design described above forms the user interface that is used by a till operator. ***Figure 1.3*** shows how the separate classes in the design "come together" to form the GUI that is displayed on screen. ***Figure 1.4*** shows a typical dialog that displays some kind of message to the user. ***Figure 1.5*** shows a typical input dialog used to specific data into the system, and finally ***Figure 1.6*** shows an example of the output generated for a report and receipt.

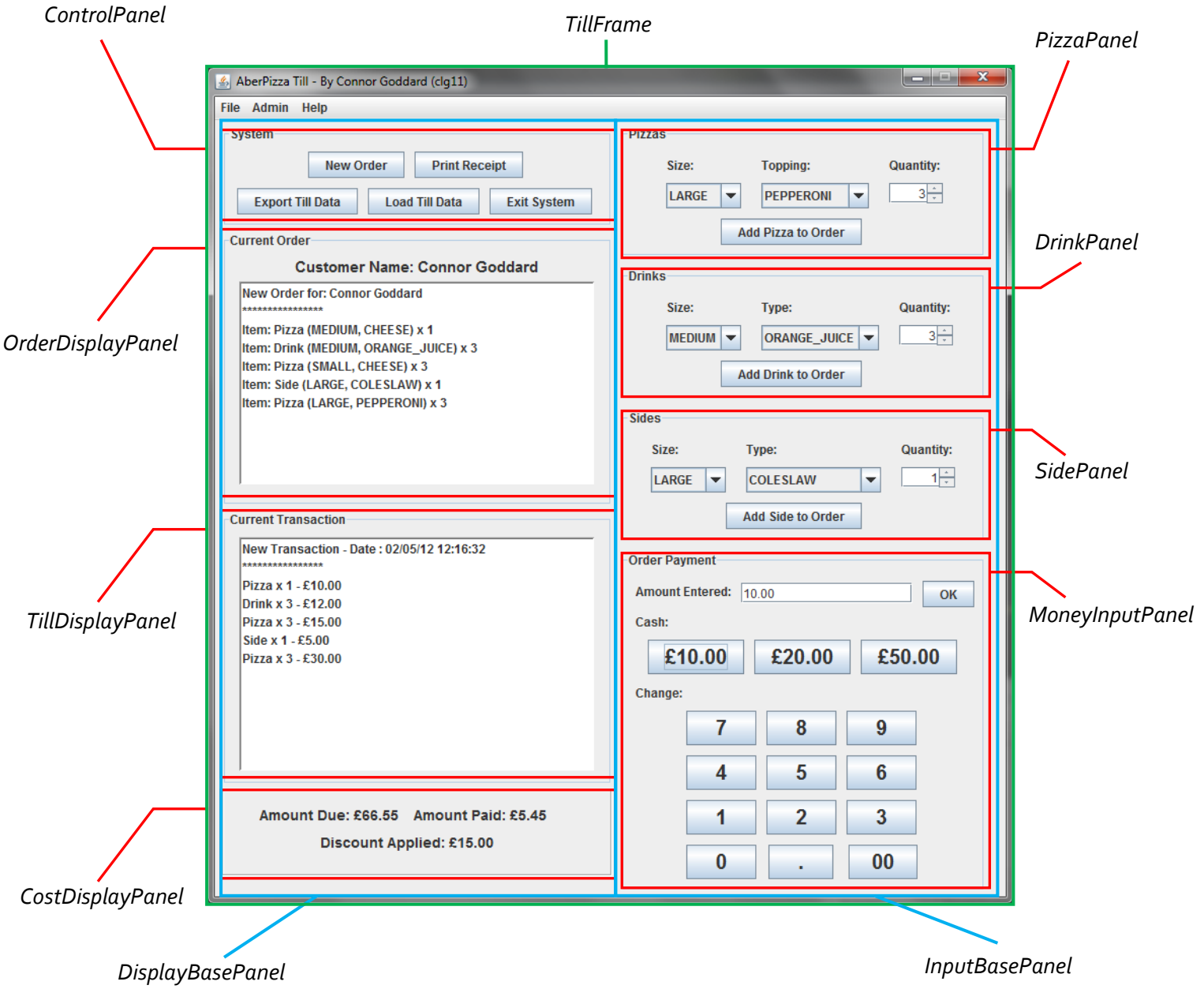


Figure 1.3: GUI Class Breakdown

There are no separate dialogs for specific tasks with the GUI. All system operations can be accessed using the panels in the diagram above, or by using the menu bar (shown at the top of the diagram). This is done to support a touch-screen user interface.

Figure 1.4: Message Box Dialog Examples

Specific dialog boxes are displayed to inform the user of any errors that have occurred, or any notifications that are useful to display to the user.

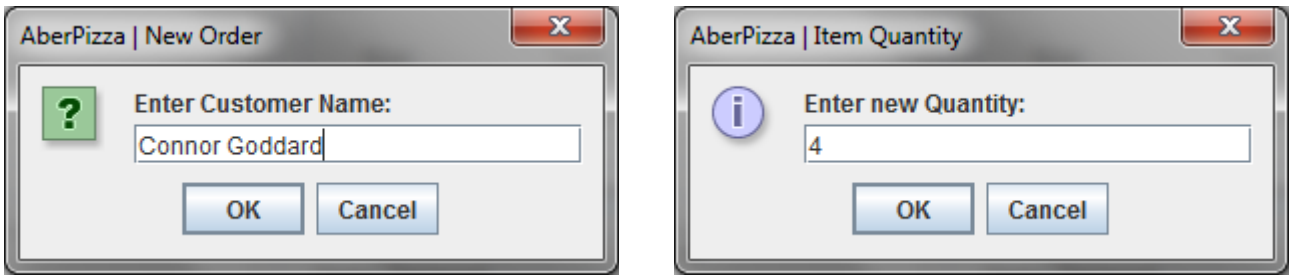
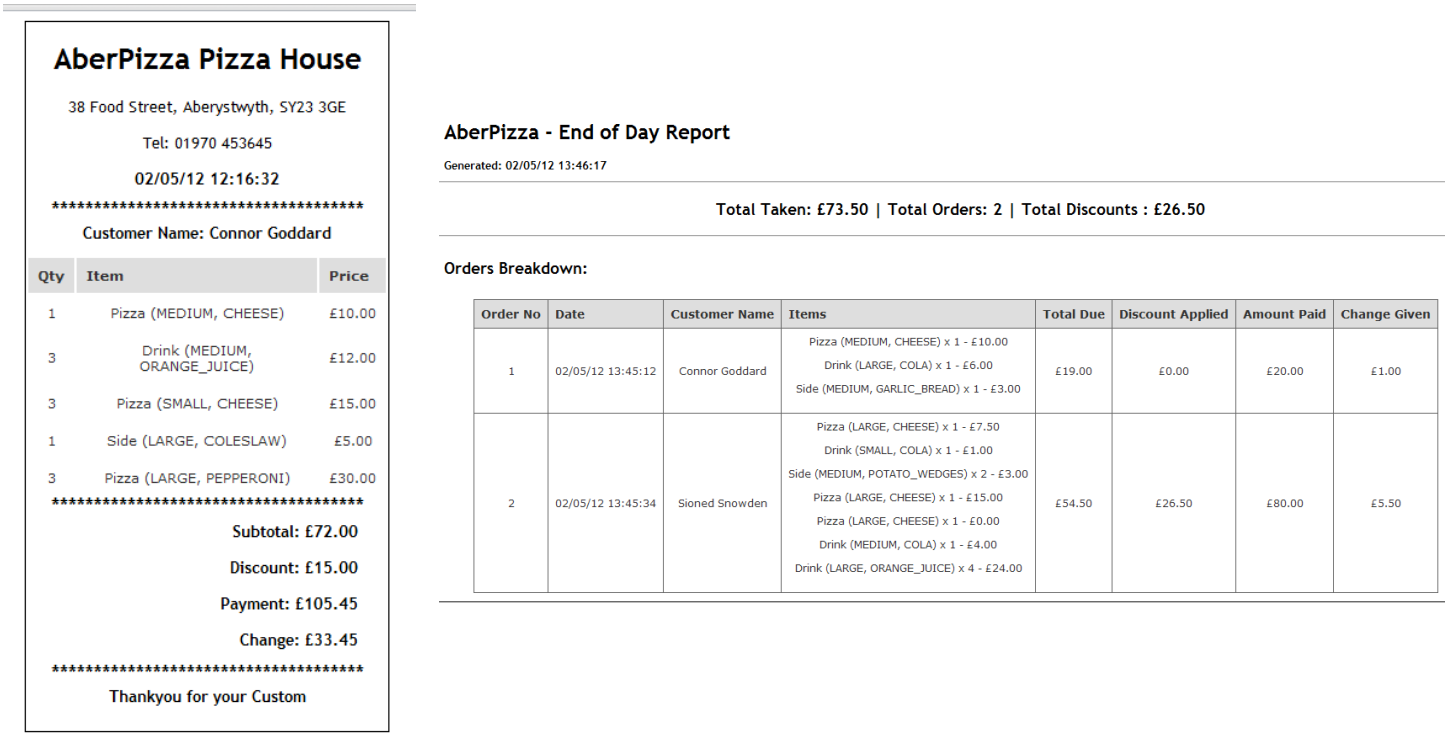


Figure 1.5: Input Dialog Examples

Input dialogs are used to allow the user to input specific values such as a customer name when creating a new order, or when changing the quantity of an order item.



AberPizza Pizza House

38 Food Street, Aberystwyth, SY23 3GE

Tel: 01970 453645

02/05/12 12:16:32

****

Customer Name: Connor Goddard

Qty	Item	Price
1	Pizza (MEDIUM, CHEESE)	£10.00
3	Drink (MEDIUM, ORANGE_JUICE)	£12.00
3	Pizza (SMALL, CHEESE)	£15.00
1	Side (LARGE, COLESLAW)	£5.00
3	Pizza (LARGE, PEPPERONI)	£30.00

****

Subtotal: £72.00

Discount: £15.00

Payment: £105.45

Change: £33.45

****

Thankyou for your Custom

AberPizza - End of Day Report

Generated: 02/05/12 13:46:17

Total Taken: £73.50 | Total Orders: 2 | Total Discounts : £26.50

Orders Breakdown:

Order No	Date	Customer Name	Items	Total Due	Discount Applied	Amount Paid	Change Given
1	02/05/12 13:45:12	Connor Goddard	Pizza (MEDIUM, CHEESE) x 1 - £10.00 Drink (LARGE, COLA) x 1 - £6.00 Side (MEDIUM, GARLIC_BREAD) x 1 - £3.00	£19.00	£0.00	£20.00	£1.00
2	02/05/12 13:45:34	Sioned Snowden	Pizza (LARGE, CHEESE) x 1 - £7.50 Drink (SMALL, COLA) x 1 - £1.00 Side (MEDIUM, POTATO_WEDGES) x 2 - £3.00 Pizza (LARGE, CHEESE) x 1 - £15.00 Pizza (LARGE, CHEESE) x 1 - £0.00 Drink (MEDIUM, COLA) x 1 - £4.00 Drink (LARGE, ORANGE_JUICE) x 4 - £24.00	£54.50	£26.50	£80.00	£5.50

Figure 1.6: Receipt/Report Examples

Bespoke receipts and reports are automatically generated and saved to disk as HTML files using a pre-defined HTML template

Testing

To ensure that the software I have designed and implemented is able to be used in the "real world" as a functioning application, I have performed both data model and GUI testing designed to test the software in many different situations to help highlight any errors in the code that can then be rectified accordingly.

Data Model Testing

To test the classes that handle the data in the application, I used the '**JUnit**' testing framework. This allowed me to design my own specific and complex tests that can automatically run the methods inside the data classes with specified test data and then check the results of those methods against criteria set by myself to test that they working correctly.

I created a JUnit test class for every (applicable) data class in my application:

- **TillTest** – Creates a new instance of a Till object and tests all the methods to ensure that orders are able to be created and that system-wide operations such as generating reports and saving/loading the till are functioning properly.
- **OrderTest** – Creates an isolated instance of an Order object and tests that the 'createNewPizza/Drink/Side' methods are able to take the correct parameters and can add the new item to the 'items' array list. The 'updateItemQuantity' method is also tested to ensure that an item stored in the 'items' can be successfully retrieved and its quantity updated. In addition, all "get and set" methods are also tested using test data to ensure that an order can successfully create and add items and return information about itself and those items.
- **OrderItemTest** – Tests the "get and set" methods for the Item object and the quantity value stored in an OrderItem to ensure that an item can be successfully set when adding an item to an order and can be retrieved when required, and that the quanityt of an item can be correctly changed when required. The constructor of the OrderItem is also tested as this is what is used by the Order class when adding a new item.
- **DiscountTest** – Discount class is tested in isolation with an Order object populated with test data to ensure that it can correctly analyse the items in an order, and accurately apply the appropriate discounts resulting in the correct amount of money being discounted from the order subtotal.
- **PizzaTest/DrinkTest/SideTest** – Three separate test classes test that their respective data classes are able to set and return their appropriate "type" enumerations (PizzaType,DrinkType or SideType) and that the constructors of those classes are able to create a new Pizza/Drink/Side object with a specific description, size, type and optional price as required.
- **ReceiptTest** – Tests that bespoke receipts can be successfully created and exported to '.html' files using test data entered into isolated Till and Order objects, by checking that files with filenames that contain test customer names and have the '.html' file extension exist.

This ensures that the Receipt class is successfully able to retrieve the necessary data from the Till and Order classes and insert it into the pre-defined template.

- **ReportTest** – Similar to ReceiptTest, this test class ensures that correct 'end of day' reports can be generated. It creates isolated Till and Order classes and populates them with test data. It then checks that the array lists containing the test data are successfully passed into an isolated Report constructor by testing sample values within the array lists, before running the 'exportReceipt' method and checking that a file containing today's date and an '.html' extension exist. It therefore ensures that order and item data is successfully retrieved and can then be manipulated into a pre-defined template, before it is exported as a file.

- **XMLTest** – Designed to test that the entire Till object (including all its order and item data) can be successfully exported to a '.xml' file, and can then create a new Till object, using that data. This is achieved by creating an isolated Till object and populating it with test data. This object is then passed to an isolated XML class that exports the Till to XML, before checking that this new XML file exists in the file system. It then attempts to create a new Till object using the data from the '.xml' file, and then "probes" the data in that new Till object to see if it matches the old data.

- **AllTests** – Test suite class that allows all the test classes to be run in one go for convenience.

GUI Testing

Unlike the data model testing, I had to manually test the GUI to ensure it was functioning as expected. To do this, I devised a testing table which details the structure and plan of the tests I wanted to run to ensure all the features of the GUI were working correctly. In the same table I then documented the results of the test and wrote additional comments accordingly.

Test ID	Description	Test Inputs	Excepted Outcome	Pass/ Fail	Comments
1	Create a new order – 'Name' input dialog appears.	Enter a customer name (Text)	Name is accepted – New order created and name displayed in "Customer Name:" display	P	
		Enter a customer name (Integer)	Name is accepted – New order created and name displayed in "Customer Name:" display	P	
		Enter nothing	System rejects name – dialog asking for a name to be entered appears	F	Dialog failed to appear and system created new order with name field empty - **Corrected**
2a	Add new Pizza to Order	Selected Size = SMALL Selected Topping = HAM_CHEESE Selected Quantity = 1	System accepts these values – new item added to 'Current Order' list box (with size, topping & quantity) and 'Current Transaction' list box (with quantity & price (price = item price x quantity). Subtotal display updated to reflect the addition of the item (Subtotal = Existing subtotal + (item price x quantity))	P	
2b		Selected Size = LARGE Selected Topping = CHEESE Selected Quantity = 36		P	
2c		Person tries to set quantity to < 1 using 'JSpinner' component on PizzaPanel	'JSpinner' does not allow value to become < 1 when user clicks 'down arrow'	P	

2d		Person tries to set quantity to > 100 using 'JSpinner' component on PizzaPanel	'JSpinner' does not allow value to become > 100 when user clicks 'up arrow'	P	
2e		User clicks 'Add Pizza to Order' button repeatedly	Expected outcome for tests 2a & 2b repeated for however many times user clicks button	P	
3a	Add new Drink to Order	Selected Size = MEDIUM Selected Type = COLA Selected Quantity = 1	System accepts these values – new item added to 'Current Order' list box (with size, topping & quantity) and 'Current Transaction' list box (with quantity & price (price = item price x quantity). Subtotal display updated to reflect the addition of the item (Subtotal = Existing subtotal + (item price x quantity))	P	
3b		Selected Size = SMALL Selected Type = ORANGE_JUICE Selected Quantity = 100		P	
3c		Person tries to set quantity to < 1 using 'JSpinner' component on DrinkPanel	'JSpinner' does not allow value to become < 1 when user clicks 'down arrow'	P	
3d		Person tries to set quantity to > 100 using 'JSpinner' component on DrinkPanel	'JSpinner' does not allow value to become > 100 when user clicks 'up arrow'	P	
3e		User clicks 'Add Drink to Order' button repeatedly	Expected outcome for tests 3a & 3b repeated for however many times user clicks button	P	
4a	Add new Side to Order	Selected Size = LARGE Selected Type = GARLIC_BREAD Selected Quantity = 32	System accepts these values – new item added to 'Current Order' list box (with size, topping & quantity) and 'Current Transaction' list box (with quantity & price (price = item price x quantity). Subtotal display updated to reflect the addition of the item	P	

4b		Selected Size = SMALL Selected Type = FRIES Selected Quantity = 74	(Subtotal = Existing subtotal + (item price x quantity))	P	
4c		Person tries to set quantity to < 1 using 'JSpinner' component on SidePanel	'JSpinner' does not allow value to become < 1 when user clicks 'down arrow'	P	
4d		Person tries to set quantity to > 100 using 'JSpinner' component on SidePanel	'JSpinner' does not allow value to become > 100 when user clicks 'up arrow'	P	
4e		User clicks 'Add Side to Order' button repeatedly	Expected outcome for tests 4a & 4b repeated for however many times user clicks button	P	
5a	Update an Item Quantity	User double clicks on an item listed in the "Current Order" list box	System retrieves selected item and displays new quantity value input dialog	P	
5b		User enters a quantity value < current value	System accepts the new value. Dialog closes and both display list boxes are updated to reflect the reduced quantity for the selected item. The discount and subtotal displays are also updated accordingly to reflect the decreased price.	P	
5c		User enters a quantity value > current value	System accepts the new value. Dialog closes and both display list boxes are updated to reflect the increased quantity for the selected item. The discount and subtotal displays are also updated accordingly to reflect the increased price.	P	
5d		User enters a quantity value of 0	System accepts the new value – Detects that the quantity of the item is now 0, and so removes the item from the order. This is reflected in the GUI by removing the item from the 'display' list boxes and decreasing the subtotal display value accordingly.	P	

5e		User enters a quantity value of > 100	System detects that the entered value is greater than 100, and displays another dialog informing the user to enter a value <= 100.	F	System accepted the quantity value and calculated the item price using this new value. - **Corrected**
6a	Applying Discounts to Order	Current Items in Order: 2 x Large Pizzas New LARGE pizza added to order	System detects 3 x LARGE pizzas in order and sets price of newly-added pizza to '£0.00'. Subtotal display value does not change and discount display value is increased by '£15.00'	P	
6b		Current Items in Order: 1 x Large Pizza, 1 x Medium Pizza New LARGE pizza added to order	No discount applied – Subtotal display value reflects new item added to order	P	
6c		Current Items in Order: 2 x Large Pizzas New SMALL pizza added to order		P	
6d		Current Items in Order: 1 x Large Pizza, 1 x Small Drink New MEDIUM side added to order	System detects 1 x LARGE pizza, 1 x small drink & 1 x medium side in order. Subtotal display value changed to (*Subtotal of ENTIRE ORDER* / 2). Discount display value changed to (*Subtotal of ENTIRE ORDER* / 2). 'Half price' discount applied.	P	
6e		Current Items in Order: 1 x Large Pizza, 3 x Medium Side New SMALL drink added to order		P	
6f		Current Items in Order: 1 x Medium Pizza, 1 x Large Side New SMALL drink added to order	No discount applied (Large pizza missing) – Subtotal display value reflects new item added to order	P	

6g		Current Items in Order: 1 x Small Drink, 1 x Medium Side New LARGE drink added to order		P	
6h		Current Items in Order: 1 x Large Pizza, 1 x Medium Side New LARGE Pizza added to order	No discount applied (Any drink missing) – Subtotal display value reflects new item added to order	P	
7a	Order Payment	User enters amount of money > Subtotal	Order payment completed. Dialog appears informing user of how much change is due (*Amount paid – Subtotal*). Dialog appears confirming receipt has been printed before order is closed and GUI resets.	P	
7b		User enters amount of money < Subtotal	Dialog appears informing user of how much money is still due. Subtotal display value is decreased accordingly and Amount due display is increased accordingly. User is then able to enter a new amount of money for payment (process repeats as required)	P	
7c		User enters amount of money = Subtotal	Order payment completed. Dialog appears confirming receipt has been printed before order is closed and GUI resets.	P	
7d		User enters incorrect format for money value	Dialog is displayed informing user of the incorrect format and requests them to try again.	P	
7e		User attempts to enter payment with no open order	Dialog is displayed informing user that an order must first be created.	F	System attempted to run payment method – application crashed - **Corrected**
8a	Print Receipts	Payment is completed on open order & order closed.	Separate receipts for all orders stored in Till (including the new order just added) are exported automatically. Dialog is displayed confirming this.	P	

8b		User clicks 'Print Receipt' button in control panel – with orders stored in till	Separate receipts for all orders stored in Till are exported. Dialog is displayed confirming this.	P	
8c		User clicks 'Print Receipt' button in control panel – with no orders stored in till.	System detects that there are no orders saved in the till, and displays dialog box informing the user of this – Exporting cancelled.	P	
8d		User clicks 'Print Receipt' button in control panel – with no items added to order	System detects that there are no items in the current order, and displays dialog box informing the user of this – Exporting cancelled.	F	System attempts to print receipt anyway – no file exported by success dialog still appears falsely. - **Corrected**
9a	Cancel Order	User clicks "Cancel Order" menu item while new order is in progress	Dialog appears asking if user wants to cancel. If 'Yes', order is cancelled, dialog appears confirming this, and GUI is reset. If no, user is returned to open order.	P	
9b		User clicks "Cancel Order" menu item while no order is open	System detects no order is in progress. Dialog appears informing user there is no order to cancel.	P	
10a	Export Till to File	User clicks "Export Till Data" button with order in progress	System exports all entire order and item information (excluding open order) to '.xml' file. Dialog is displayed on screen confirming the successful export.	P	
10b		User clicks "Export Till Data" button with no order open	System exports all entire order and item information '.xml' file. Dialog is displayed on screen confirming the successful export.	P	
11a	Load Till from File	User clicks "Load Till Data" button and opens acceptable '.xml' file	'Open File' dialog is displayed. User selects the acceptable '.xml' file using dialog. System then loads in this file. Dialog displayed confirming a successful load.	F	'Open file' dialog fails to display. - **Corrected**
11b		User clicks "Load Till Data" button and opens un-acceptable '.xml' file	'Open File' dialog is displayed. User selects an unacceptable file using dialog. System detects the file is not acceptable and displays	F	

			dialog informing user. Load aborted.		
12a	Generate 'End of Day' Report	User clicks 'Generate Till Report' menu item with orders saved in till.	System automatically exports report with all order data contained. Dialog is displayed confirming export was successful upon completion.	P	
12b		User clicks 'Generate Till Report' menu item with till order array empty.	System detects that there are no orders saved in the till and displays dialog informing user that there are no orders to export. – Report generation cancelled.	F	System continues regardless and exports an empty report - **Corrected**
13a	'About' Dialog	User clicks 'About' menu item	System displays "About" dialog	P	
14a	Exit System	User clicks "Exit System" button	Dialog displayed asking user if they want to exit. If 'Yes', the system exports till data automatically and then exits. If 'No', user is returned to till GUI.	P	
14b		User clicks 'Exit' menu item		P	

Self-Evaluation

As an overall opinion, I am very happy with my solution to the problem. I feel that it meets all the specified use cases and objectives set, and has utilised all but few of the new programming tools and skills that have been demonstrated to me throughout this module. However there are some points I have identified that could have been done better, and I seek to improve if I was to do a similar project in the future.

Data Model

In general, I found designing and implementing the data model for the software easier than I initially expected. I quickly realised how robust and useful abstract classes and interfaces can be when trying to implement similar classes and data types in a system (such as the Pizza, Drink and Side classes. I also found it very useful that by having the interface, I was able to refer to any particular "product" (Pizza, Drink or Side) as a type of 'Item', and then simply add a cast to that 'Item' object as required. This allowed me to store all three different types of product in one array list which was of type 'Item', instead of having three separate array lists, one for each product type.

Although overall I found the design and implementation of the data model relatively issue-free, I did run into some problems when it came to the XML serialisation. It took me a while to fully understand that in order for serialisation to work properly, "getters and setters" are required for all data objects as well as default constructors for every class that are to be exported. I realised this after many failed attempts exporting to XML, and every time having an '*InstantiationException*' thrown by the encoder. Adding the default constructors fixed the issue and I will be sure to bear this in mind if I have to perform XML serialisation again.

GUI

I am particularly pleased with the final GUI design. My aim was to design a user interface that was:

a) Intuitive enough to be used quickly - Important when ensuring the time taken to process and order is kept to a minimum

b) Easy to learn and use – Allows new till operators to learn how to use the software quickly and helps to reduce errors while using the system (again reducing order processing time)

c) Compatible with touch screen EPOS systems – Research I conducted shows that the majority of till systems nowadays are run on touch screen systems

I believe my design addresses these objectives and provides a simple, yet sophisticated user interface that is easy to learn and quick to use when time is a critical factor. In particular I believe the 'number input panel' used for entering customer payments into the system works very well and provides the extra 'ease-of-use' needed when trying to serve a customer as quickly as possible.

Overall I was very happy with designing and implementing the GUI, and was comfortable with using the 'Swing' framework to build the frames, panels and components that make up the GUI. I never hit any major problems that I was not able to fix with some additional research, and found that it

was relatively easy to create rather complex interface layouts using the '*SpringLayout*' layout manager.

On reflection however, I do now realise that I could have improved the GUI code by having one "main" listener to respond to the interactive components across the whole GUI, as opposed to having lots of separate, local listeners for each panel. The advantage of having just one separate listener class, is that it organises the code better by keeping all component action methods in one place, which in turn means that would be the only class that would require access to the data model, as opposed to the current structure which has separate listeners for each panel, and so as a result requires many references to the data model, involving more code than is necessary.

Testing

The task of testing my system was broken down using two separate approaches. The data model was tested using the JUnit testing framework, which allowed me to create and run automated tests using specified criteria on the methods contained in the data classes.

This project was my first attempt at using the JUnit framework, which I found an extremely useful and powerful tool when ensuring that the data classes in the system were working as required, and were producing the results that I was expecting. The methods available in JUnit were very easy to implement, and I have used a variety of these methods in the test code as required to thoroughly test many different outputs from the methods in the data classes.

The application GUI was tested using traditional testing tables. I felt this was an appropriate method of testing the GUI as it allowed me to see for myself how a user would interact with the program which is always useful to know, and also ensured that all the main functionality of the GUI was tested for intuitiveness and aesthetics, as well as whether it performed it's intended task which is important when considering the large variety of people that will potentially use the system.

Feedback Form

Year: 2011-2012

Module: Software Development (CS12420)

Assignment Number: 2

Assignment Description: AberPizza Till System

Worth 20% of final mark for this module

How many hours (approx.) did you spend on this assignment? 50

Expected Letter Grade: B (2:1) (76/100)

And why?

1. 'Quality of the code & Javadoc Commenting' (Worth 5/100 marks):

   I believe in the project I have addressed the criterion well. The source code throughout the project has been correctly formatted (using appropriate indentation where necessary) and heavily commented using both Javadoc, and normal comments to ensure that the code is easily readable by anyone who may want to view the code at a later date. I therefore propose an award mark of **4/5**.

2. 'Design & Implementation of the Data Classes' (Worth 25/100 marks):

   After reviewing the initial project class diagram, I believe the final design and implementation of the data classes fulfils this criterion to a high level. My design incorporates the original design (apart from the moving of the 'getReceipt() method') and also makes use of the new techniques I have been taught throughout the module (including the use of abstract classes and threads). As a result, the data classes enable the system to fulfil all the use case objectives set out in the project brief.

   I do however acknowledge that my design does deviate at times from the original design (with regards to the 'getReceipt()' method) and unfortunately (due to time constraints) does not allow new products (e.g. a new type of pizza) to be added to the system catalogue without the code being edited. As a result I propose a mark of **20/25**.

3. 'Design & Implementation of GUI – Linking to Data Classes' (Worth 20/100 marks):

   I am happy that the GUI I designed for the application meets this criterion to a high standard. I believe my design provides an intuitive user interface that is easy to learn and operate under pressure, while at the same time ensuring that the user is able to run all the operations needed to meet all the use case objectives.

   The underlying structure to the GUI I feel is sensible, and links well with the data classes. However (as discussed in the self-evaluation) I feel this linking could be very much improved by using one main action listener, as opposed to many individual listeners, all of which

require a link to the data classes. Bearing this in mind, I propose a mark of **16/20** for this criterion.

4. 'Testing – Including JUnit & Test Tables' (Worth 20/100 marks):

   Overall, I believe I have performed thorough testing on the core features of the application. Every data class in the application (that is able to be automatically tested) has an associated JUnit test case class that performs a number of tests designed to ensure that the classes are subject to a large variety of common, and less-common scenarios, helping to remove as many errors as possible before the system released.

   The GUI testing tables I feel provides thorough testing of practically all aspects of the GUI in many different scenarios again to highlight any errors that can then be fixed. I feel these test tables are also well documented and provide detailed information into any tests that failed, and why that was the case.

   I do feel however that (due to time constraints) I did not collect any screenshots of the GUI being tested which would have been useful to support my test tables, by showing evidence of the testing taking place. I therefore propose a mark of **16/20** for this criterion.

5. 'WOW Marks' (Worth 10/100 marks):

   After implementing the core functionality of the application, I attempted include some additional features, as well as improving existing features, and experimenting with some of the very latest skills I had learnt on the module. I am very pleased with the final design for the HTML receipts and reports.

   I feel they look professional, and provide a clear representation of the data they contain. In addition, both the report and receipt HTML templates are XHTML 1.0 valid. The application also includes the added functionality of being able to send the exported reports and receipts to a printer and is run using a dedicated thread, which was the latest topic covered in this module prior to the submission of this assignment. I propose a mark of **6/10**.

6. 'Documentation' (Worth 20/100 marks):

   I believe overall I have produced detailed, well presented documentation that provides a real insight into every aspect of how my final application was designed, implemented and tested and my approach to these. I feel the documentation provides enough detail for someone who has never seen the project before to be able to gain a good idea as to the nature of the project and my application.

   I do however appreciate that the length of the documentation might exceed what was expected, and even after attempting to remove some un-necessary information it possibly still goes into too much un-necessary detail. I am also unsure that incorporating the justification of the design into each of the class descriptions was a good idea, and next time would probably not do this, as to not confuse the reader. I therefore propose a mark of **14/20** for this criterion.

What did you Learn?

I have learned a tremendous amount in this project. The most valuable skill I have learned I feel is how to use interfaces and abstract classes in a Java program. Both provide amazing scope for designing complex object-orientated programs relatively easily, I now appreciate them highly and will endeavour to use again in future projects where possible.

Another key skill that I have been able to practice in this project is JUnit testing. I now have an appreciation of how powerful unit testing can be when it comes to data classes, and found that using JUnit tests in my application allowed for a much broader variety of scenarios to be tested on the data classes, helping to identify addition errors that may have been overseen with traditional testing. I have also attempted to use some threads, allowing me to experience the benefits of concurrent programming.

Finally, I have started to realise that I perhaps need to learn how to reduce the size of my documentation by becoming more concise when writing. Even though I may not have learned how to do this yet, this project has brought this to light, and I will work on this ready for any future projects.