Table of Contents

[Table of Contents 2](#_Toc449652100)

[1) Introduction 3](#_Toc449652101)

[2) System Requirements and Assumptions 4](#_Toc449652102)

[2.1) Requirements to run 4](#_Toc449652103)

[2.2) Assumptions 4](#_Toc449652104)

[3) Class Diagram 5](#_Toc449652105)

[4) Design Decisions and Design Patterns 5](#_Toc449652106)

[4.1) Patterns Used 6](#_Toc449652107)

[4.1.i) Strategy Pattern 6](#_Toc449652108)

[4.1.ii) Iterator Pattern 6](#_Toc449652109)

[4.1.iii) Factory Pattern 7](#_Toc449652110)

[4.1.iv) Abstract Factory Pattern 7](#_Toc449652111)

[4.1.v) Singleton Pattern 7](#_Toc449652112)

[4.1.vi) Observer Pattern 8](#_Toc449652113)

[4.1.vii) Composite Pattern 8](#_Toc449652114)

[4.1.viii) Model View Controller Pattern 8](#_Toc449652115)

[5) How to use the program 8](#_Toc449652116)

[5.1) Before compiling the application 8](#_Toc449652117)

[5.2) After compiling and executing the program 8](#_Toc449652118)

[5.2.i) File 8](#_Toc449652119)

[5.2.ii) User 9](#_Toc449652120)

[5.2.iii) Zones 9](#_Toc449652121)

[5.2.iv) Cart 9](#_Toc449652122)

[5.2.v) Help 9](#_Toc449652123)

[6) Testing strategy and test results 10](#_Toc449652124)

[6.1) Test during development 10](#_Toc449652125)

[6.2) Post development testing strategy and outcomes 11](#_Toc449652126)

[7) Application evaluation, improvements, and limitations 11](#_Toc449652127)

[7.1) Application Evaluation 11](#_Toc449652128)

[7.1.i) Design requirements 11](#_Toc449652129)

[7.1.ii) Customer/User requirement 11](#_Toc449652130)

[7.1.iii) Design Patterns that must be used 12](#_Toc449652131)

[7.2) Improvements 12](#_Toc449652132)

[7.2.i) Core Functionality Improvements 12](#_Toc449652133)

[7.2.ii) GUI Improvements 12](#_Toc449652134)

[7.2.iii) Testing Improvements 13](#_Toc449652135)

[7.3) Limitations 13](#_Toc449652136)

[Title: FantasyLand Application 13](#_Toc449652137)

# Introduction

The assignment requires that a solution is created according to the specification in the Problem Statement (PS) outlined below:

*For this assignment you will be expected to take a given problem statement (this document) and convert it into a working implementation delivered in Java. You are expected to develop a good object-oriented design, apply design patterns where appropriate, and develop a comprehensive testing plan by which your finished code can be tested. As a minimum you must include a comprehensive class diagram that shows the full interaction of each class in your solution and also where particular kinds of design patterns have been applied.*

*There is no need to follow a full software development life-cycle for this. The requirements for this project relate primarily to ensuring that you have a software solution that implements everything required.*

*Problem Statement*

*The FantasyLand is to be implemented as a Java application or applet which has a set of different fun-zones, such as a movie zone, a game zone, a food zone etc. In each zone customers can pay to take part in different activities and/or to buy goods specific to that zone. For example in food zone customers can pay to buy food or/and learn how to make food. When customers enter the FantasyLand the software greets them and let them do the following:*

* *List the fun-zones available*
* *Get a shopping cart*
* *Enter a fun-zone*
* *Get a list of items (both activities and goods) available for sale at the zone*
* *Add items to the shopping cart*
* *Proceed to checkout and purchase the items*
* *Edit/remove items from shopping cart*

*Your implementation must have at least three customers, four fun-zones, five items (mix of goods and activities for sale) at each zone. The user interface of the software can be either textual or graphical, but extra marks will be awarded for the GUI implementation.*

*Your implementation must use the following design patterns:*

* *Iterator pattern to iterate through collection*
* *Abstract Factory with factory methods to create all fun-zones*
* *Singleton to allow only one instance of any factory type you create*

*Observer pattern to make all fun-zones observable objects so that a customer (when registered for notifications) can be notified whenever a new item has been added for sale at a fun-zone*

*You may also use any other patterns you feel appropriate. Your implementation code must be well documented and you need to clearly identify in the code the places where you have implemented the design patterns.*

*The above is only a brief outline of the required system and it is your task to analyse the system and interpret requirements before designing and implementing a suitable solution. If you feel that the description is ambiguous in any way then you must list any assumptions you make.”*

In short, a user of the application is to be given the option of purchasing a range of goods and services from a list of fun zones. The user should be able to list the fun zones and their contents, create a shopping cart, enter a fun-zone, and select items for purchase. The user should be able to edit/remove items from the cart. The code must illustrate good object oriented design and include the required programming patterns.

# System Requirements and Assumptions

## 2.1) Requirements to run

* Java runtime 7.0 or above including Java swing libraries
* Adequate computer resources (*e.g.* memory/CPU) are assumed and should be negligible by modern computing standards.

## 2.2) Assumptions

1. The PS requires multiple customers to be available. It was assumed this meant a single user interface with switching between the users. Therefore, no multi-user support nor user password protection was implemented.
2. Thread safety was assumed not to be a requirement. Thread safety has been implemented in elements of Cart.java to illustrate thread safety as an example. However, as the application does not support multi-user input and only one thread is run, this has not been implemented across the program.
3. Internationalisation is illustrated in the Help/About dialog in FantasySwing.java. This was purely to illustrate how the help/about dialog could be internationalised but was assumed not to be a requirement.
4. Financial transactions were assumed not to be secure, nor did the PS require multiple payment options. A strategy pattern could have been implemented for this but it was assumed to not be required.
5. It was assumed data was not required to be persistent, only that the logical requirements in the PS were to be implemented. Persistent storage, data verification, data corruption detection with a procedure for managing corrupted data, and data validation could be implemented but was assumed not to be required. In conjunction with this, the PS did not mention if the stock should be considered finite, nor if the stock should be considered unique or shared between zones. The implementation passes initial stock values into the zone factory upon initialisation and is unique to the parent zone. This could be expanded via a strategy pattern to incorporate increased functionality such as ordering more stock when levels are low. However, it was assumed to not be required for the PS. It was assumed that when stock items are placed into the cart they are to be removed from zone stock (analogous to a supermarket basket).
6. System memory and computing requirements were not factored into the design. It is assumed any computer would have adequate system resources to run this application – see Requirements
7. It was assumed all users have access to all zones and so switching user does not affect the selected zone. If users were differentiated by access rights (*e.g.* age or gender), the application program state maintained by the context object (Context.java) would have to be modified via a callback *e.g.* Observer pattern.
8. Users (observers) can [un]observe zones (observable) on demand but the default option is unobserved.

# Class Diagram

Although outlining the class diagram through words is possible, a UML diagram is more readable. As such, the supporting documents includes three images of the UML diagrams for the GUI, core functionality, and testing functionality (GUI\_UML, PACKAGE\_UML, and TEST\_UML respectively). Alternatively, the diagrams can be viewed inside the source files in eclipse if ObjectAid is installed.

GUI\_UML.png

PACKAGE\_UML.png

TEST\_UML.png

# Design Decisions and Design Patterns

Good object orientated design follows a series of design principles:

1. Identify areas of the application that vary from those that do not
2. Program to an interface rather than an implementation
3. Favour composition over inheritance
4. Classes should be open for extending but not modification
5. Classes should attempt to adhere to a single set of responsibilities

With these in mind, the following design decisions were made. Initially, the brief was assessed in order to group similar classes (nouns) into class hierarchies that encapsulated functionality so as to minimise code duplication (design principles 1, 3, and 5). Interfaces were then designed based on the functionality derived from the brief and base classes were constructed to implement the interfaces (design principle 2, 3, and 5). Class variables were designed only to be accessed via accessor methods to encapsulate object data (design principle 4). This decouples logical functionality from object inheritance thereby permitting objects from different inheritance hierarchies which implement the same interface to be used as equivalents (design principles 2, 3, 4). Furthermore, private methods do not check input parameters as it is assumed that the class only passes valid arguments within the class. However, public and protected methods do check input arguments as they cannot be guaranteed. The core logic should be robust against methods being called with bad arguments (*e.g.* null, empty, or whitespace-only strings being passed as the name of a user for user registration) and this will be verified using unit test regression suites.

Debugging is an important feature of program development. Therefore, the ability to debug was embedded within the application. A base class Debugger implements DebuggerInterface functionality. The Debugger class itself allows output to be annotated by name, an optional string qualifier, and a string. Errors are always output to stderr. Normal debug is sent to stdout when enabled. Furthermore, most interfaces extend DebuggerInterface and most base classes extend Debugger. During the design process, extending the Debugger class was not possible for all classes (*e.g.* FunZone which extends Observable), therefore, the DebuggerInterface implementation of this object implements the DebuggerInterface methods and passes the calls directly to an embedded Debugger instance. This is a way of simulating multiple inheritance. It could also be extended by levels of debugging severity and alternative output destinations *e.g.* network logging. Using stdout and stderr is a common programming method allowing other applications to redirect this application’s output and these were implemented here.

The overall implementation was designed to follow the Model View Controller (MVC) pattern as the pattern has been shown to be a very effective method of designing an application with a user interface. The directory FantasyPackage contains the logic of the application with a singleton FantasyCore object which exposes and encapsulates functionality by implementing a FantasyCoreInterface (implementing the Model and acting as the Controller). The directory FantasyGui contains a graphical user interface (GUI) using Java Swing and has been developed to be available both as an applet and as an application as specified in the brief (implementing the View). The directory FantasyTest was developed to contain an application unit test framework along with test examples for testing purposes. Additionally, it includes menu bindings for convenient GUI usage when enabled.

Lastly, the application was designed to be able to be compiled in production or development mode depending on the precise requirements of the user and the testing process. Additionally, customisable options allow further modification, including: default window dimensions; start-up user initialisation source; and an illustration of internationalisation support. Development mode provides the option of enabling the tests menu and enabling outputting of debug data.

## 4.1) Patterns Used

This section will cover the design patterns used, where they are used, and why they are used.

### 4.1.i) Strategy Pattern

This is used on start-up to populate the program with users. Only one concrete class is actually created but the class selected is configurable at compile time (Runtime.java) to get user lists from different sources *e.g.* a fixed list, a saved file, or from across the internet (verified as functional). This was primarily to illustrate the use of a strategy pattern and could be developed further to allow strategy selection at runtime by incorporating the strategies into a context class and selecting the desired strategy on load.

### 4.1.ii) Iterator Pattern

This is used extensively for iteration whilst encapsulating the underlying implementation of object collections. Either iterator objects are exported from classes that manage groups of objects (public Iterator<class> getIterator()) or the implied iterator is used in class methods (for (class object:list)) that use their own object lists. Creation of a custom iterator class and exporting an object of that class had been previously implemented but the interface based design does not easily allow this form of usage due to the additional class declaration. The decision was taken to note the possible implementation and indicate the reason for the decision not to implement it and keep to the interface based programming design cleaner.

Examples are cited in:

FantasyCore.java

Line 36 exports the Zones iterator and 113 the Users iterator, an example of a standard Java iterator as an iterator object

Lines 44, 86, and 119 have examples of the implicit iterator

Line 252 shows a “for loop” using an index to iterate over the array. This is not an iterator pattern as it uses knowledge of the underlying implementation (array) in the class but it may be faster at runtime. It is used to illustrate the difference between iteration and an iterator pattern

Cart.java

Lines 30, 83, 109, 146, and 174 are standard for loop iterator

Line 160 exports a cart entry iterator object

FunZone.java

Line 70 exports a FunZoneStock iterator

Lines 44, 80, and 140 show for (class object:list)

Line 160 shows iteration over a string array using an index. This is iteration but not using an iterator pattern as previously explained. It was thought to be a more efficient implementation and a good choice as the array was created locally

There are many more examples of this pattern. Further examples can be found through a “grep” of the source code.

### 4.1.iii) Factory Pattern

The factory pattern was partly used in fulfilment of the brief and also because it is not necessarily known at runtime how many, or what type, of fun item are to be instantiated (activities and products). Additionally, the factory pattern allows for the instantiation of new objects dynamically at runtime which allows for the addition of new fun items. This decouples and encapsulates the creation of objects.

Two Examples are:

FunActivityFactory and FunItemFactory subclasses of FunThingFactory instantiate objects of type ‘Activity’ and ‘Item’ which implement the FunThingInterface.

Zone\*.java (except ZoneProducer) extend FunZoneFactory and contain the subclasses of the Zones as well as their unique ZoneFactory. The ZoneFactory’s create objects that implement the FunZoneInterface.

### 4.1.iv) Abstract Factory Pattern

Similarly to the factory pattern, the abstract factory pattern was partly used in fulfilment of the brief but also to encapsulate the instantiation of factories without specifying the factory classes. An abstract factory pattern is used here to create factory subclasses of a factory baseclass. The problem statement required the implementation of an Abstract Factory Pattern and Zone creation appeared to be the most suitable fit. ZoneProducer is a factory producer which returns factory objects that extend the abstract FunZoneFactory base class which implements the FunZoneFactoryInterface functionality. The factory is selected based on parsing a string argument and sets the factory debugging state according to the second parameter. Factory objects are either singletons for well known zones or newly created if a novel Zone is being requested.

### 4.1.v) Singleton Pattern

The Singleton pattern was chosen in several places of this design. This was partly to fulfil the design brief for factory but also to make sure the underlying logic could only be instantiated once. As implemented, Singleton’s persist for the lifetime of the application once instantiated. Therefore, FantasyCore class is declared as a Singleton and holds the instance through a private variable. Creation time is delayed until first access by a public static accessor function which has parameters passed that are used in the instantiation of the first FantasyCore object. If the FantasyCore singleton is found to be null, the FantasyCore singleton is created passing in the runtime creation parameters. FantasyCore singleton is then returned.

As required by the brief, each factory created is set to be a singleton. The factories are declared within the ZoneProducer which is also declared as a Singleton so as to avoid multiple ZoneProducer instantiations creating their own factories.

### 4.1.vi) Observer Pattern

The observer pattern was partly used in fulfilment of the brief which specifies users must be able to observe and receive notifications when new items are added to the observed zone. The controller (FantasyCore) adds or removes an Observer (a User object implementing a UserInterface that extends the Observer interface) from an Observable (FunZone that extends Observable class).

### 4.1.vii) Composite Pattern

The composite pattern was chosen to implement development mode which was designed to affect multiple independent class hierarches in the same way. This was specifically used for debugging where collections of debuggable objects are treated uniformly as a debuggable object. Debuggable objects implement the DebuggerInterface and are linked together so that the differences between the objects can be ignored and treated as a whole, or as individuals.

### 4.1.viii) Model View Controller Pattern

The MVC was chosen to be implemented as the design brief required the implementation of a user interface. As mentioned previously, the MVC pattern has been shown to be a very effective method of designing an application with a user interface. To avoid replicating what was stated at the start of Section 4.1, the MVC will be briefly summarised here. The GUI (FantasySwing.java) is the View component of the MVC pattern and makes requests of FantasyCore which is acting externally as the Controller but internally implements the Model. Separation of these last two functions could be done but was thought unnecessary.

# How to use the program

## 5.1) Before compiling the application

* Select user strategy string in Runtime.java, keep in mind to comment out the strategies not required otherwise a compilation error may occur. Eclipse may still build the application but functionality could be compromised
* Set application height/width in Runtime.java
* Set developer mode on or off: currently allows debugging output and test menu in “on” mode. Errors are still reported by the Debugger class even if debugging is turned off.

## 5.2) After compiling and executing the program

*Note: It will be assumed the user is running in development mode.*

Upon execution of the program, the “help” display box is shown. This provides the user with some basic information on running the application and can be accessed later in the “Help” drop-down menu. Press OK or close the display window.

You are now looking at the main display of the GUI. The GUI displays:

* Current user selected
* Zone the user is in
* Notifications\*
* Cart goods and activities, credit, and left to pay

\*Displays notifications if the user observes a zone which introduces a new product

The next sections will be broken down by Menu with reference to the main display when required. Every menu item can be selected with hotkeys. The hotkey associated with the menu item is shown by an underline of the character to press, *e.g.* “­Contents”. The initial menu hotkey requires the “alt” modifier.

### 5.2.i) File

Exit – allows the user to exit the application. It could be developed to incorporate other file processes (*e.g.* saving to persistent storage)

### 5.2.ii) User

Choose User – Select the user you wish you use

Watch current zone – Sets the user to observe the current zone for new products/activities

Stop watching current zone – Stops the user from observing the current zone for new products/activities

Clear Notifications – Clears notifications sent from observed zones for the user

### 5.2.iii) Zones

Select Zone – Select the zone you wish the user to view

Contents – Displays the current contents of the current zone in a new panel

Purchase – This menu item has a submenu which allows the user to select either an *Activity* or a *Product*. If the zone has any of the requested types (*e.g.* products), a new box displays the possibilities in a drop-down menu. This can then be perused at the user’s discretion.

Add New Item – This adds a randomly named item/product (*e.g.* “Item28”) to the zones contents. The zone will inform all users registered as observers of the zone

Add New Activity – This adds a randomly named activity (*e.g.* “Activity28”) to the zones contents and notifies observes of the zone

### 5.2.iv) Cart

Cart Contents – Display the current contents of the current users cart in a new panel

Remove Item – Opens a display for the user to remove pre-selected items. Removed items are returned to the original zones stock if possible, i.e. the zone still exists. If the item being returned is new to the zone, it alerts observing users of the zone that the product is now in stock again through the notifications display part of the GUI.

Pay – User selects this option to pay the difference (balance) between the total cost of the items and the amount of credit. The user must pay the amount into cart credit in order to checkout.

Cart Checkout – User selects this option to checkout all items in the cart. If there is sufficient credit to cover the costs (To Pay displays 0 or less), checkout can continue. Otherwise, it will not.

Refund – This menu item has a submenu which allows the user to *Refund Cart Balance*, or *Refund Cart Credit*. Refunding the cart balance refunds the difference between credit and total, *e.g.*there is £11 credit and £10 total to pay, and it will refund £1. Refunding the cart credit will refund the entirety of the stored credit.

### 5.2.v) Help

Show Help – Displays the help menu to the user

About – Displays the information about the application to the user

Add Tests/Tests – *Add Tests* adds the *Tests* menu and replaces the *Add Tests* menu option. This is found only in developer mode.

**Tests submenu** – this menu demonstrates testing the underlying logic (Controller testing) through an example regression suite.

Add/Remove items from cart – adds and removes a random number of items to the cart

Add/Remove new zone – adds and removes a random number of zones

Add/Remove new users – adds and removes a random number of users

# Testing strategy and test results

## 6.1) Test during development

Testing of the application was subdivided into three different stages: logic testing (FantasyPackage), testing via GUI (FantasyGUI), and independent testing. Each stage of testing has the potential to expose a fault. Each fault is categorised by: description of the fault, how to replicate the fault, and the stack trace output by the debugging if possible. No stage can pass onto the next if a fault is detected. This is illustrated below:

Fault detected?

YES

NO

Fault detected?

NO

YES

Fault detected?

YES

NO

Logic testing of FantasyCore and the dependent classes (FantasyPackage) has associated unit tests that are exposed as a regression suite. This can be exposed in the GUI in development mode, but does not require a GUI to run. An example of this is given and further unit tests could be created for a more robust testing suite. Logic testing is facilitated with runtime debugging exposed through developer mode. The number of tests available is too small for production testing but illustrates the principle.

GUI testing was tested manually according to defined behaviours. As there is no input outside of controlled drop-down boxes, testing procedure could be simplified to the available menu options. During development, each menu option was tested in a random order within the same FantasyLand instance. Additionally, as the GUI is sufficiently intuitive and simple, a precise testing methodology for each menu option does not require verbose documentation. Increasing functionality (*e.g.*as outlined in section 7.2) may warrant a detailed testing structure depending on the nature of the added functionality. An illustration of the GUI testing is outlined in section 6.2.

Independent testing was briefly conducted once by a professional tester shortly prior to assignment completion. Application simplicity and robust testing procedures during development were considered enough to warrant minimal independent testing. The application was compiled in non-developer mode and the tester only had access to the compiled application: the tester had no access to code or input on program design.

## 6.2) Post development testing strategy and outcomes

There are endless permutations that can test the code in slightly different ways. Below is an illustration of tests run but it is not an exhaustive list, *e.g.* does not include a test “Adding 1,000 random items”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Testing Unit | Test Description | Expected Outcome | Outcome | Pass/Fail |
| GUI consistency | Does every menu item contain a uniquely accessible shortcut within its context? | Pass | Pass | Pass |
| Product/activity allocation | Add multiple products/activities to different users and check FunThing assignment. Add new items/activities to various zones and then add new products/activities. Check FunThing assignment. | Pass | Pass | Pass |
| Observer pattern | Select a user, select a zone, and set the user to observe the zone. Add a new item to zone and see if the notification appears. Stop watching the zone and see if the notifications stop. | Pass | Pass | Pass |
| User strategy string | Select each different strategy string and compile the program. Compare strategy string expectations with available users. E*.g.* check users against user list found at http://anthonybane.com/FantasyLand/users.txt , note this source will eventually be deleted and the error user “Dummyuser” should appear | Pass | Pass | Pass |
| Help/About | Select the Help and About menu options and see if the corresponding display box appears | Pass | Pass | Pass |

# Application evaluation, improvements, and limitations

## 7.1) Application Evaluation

The PS states certain requirements that have to be met. Each requirement is outlined below according to the requirement category, and whether the application meets the requirement:

### 7.1.i) Design requirements

|  |  |
| --- | --- |
| Requirement | Requirement Met? |
| Be a Java application or Java applet | Yes (both) |
| At least four different fun-zones | Yes |
| At least three customers | Yes |
| A minimum of 5 activities/products per zone | Yes |
| A user interface, either text or graphical | Yes |

### 7.1.ii) Customer/User requirement

|  |  |  |
| --- | --- | --- |
| Requirement | Req. met? | How was it met? |
| “List the fun-zones available” | Yes | User can select dropdown menu “Zones” – “Select Zone” to view available zones |
| “Get a shopping cart” | Yes | Each user upon instantiation is assigned a new Cart object |
| “Enter a fun-zone” | Yes | User can select zone to enter via the drop-down menu “Zones” – “Select Zone” |
| “Get a list of items (both activities and goods) available for sale at the zone” | Yes | Inside a zone, the user can select dropdown menu “Zones” and “Contents” where an updated list of available activities and products is stored. |
| “Add items to the shopping cart” | Yes | User can purchase a product or activity from a zone via the drop-down menu “Zones” – “Purchase” – “Activity”/“Product” |
| “Proceed to checkout and purchase the items” | Yes | The user can checkout via the drop-down menu “Cart” – “Checkout Cart”. The user must first pay the balance through the “Pay” option. This is so the user can add credit to the cart before checkout. |
| “Edit/remove items from the shopping cart” | Yes | User can remove items from the drop-down menu “Cart” – “Remove Item” |

### 7.1.iii) Design Patterns that must be used

|  |  |  |
| --- | --- | --- |
| Pattern | Has it been met? | Example of where |
| Iterator Pattern | Yes | Cart.java |
| Abstract Factory Pattern | Yes | ZoneProducer.java creates factories, *e.g.* clothes\_shop\_factory |
| Singleton Pattern | Yes | FantasyCore.java |
| Observer Patterns | Yes | FunZone.java extends observable and User.java observes |

*Note: A full list of patterns used and locations can be found in section 4.*

## 7.2) Improvements

### 7.2.i) Core Functionality Improvements

* Encapsulate all parameters in interface classes *e.g.* User constructor should use a UserIdentifierInterface rather than a String for better extendibility
* Singletons could be destroyed (for memory efficiency) once no longer needed by returning them to their originating object by implementing putObject() as well as getObject()
* Persistent storage of data with integrity checking and verification utilising serialisation
* Discounts/sale modifications for individual users (Strategy pattern)
* Restrict zones/items to certain users
* Using a decorator pattern for certain items (*e.g.* in ZoneFoodCourt for toppings to food)
* Multiple purchases/returns of an item
* Post purchase tracking of products/activities for future refund requests
* Different locations for user data upon start-up (see Runtime.java)
* Different locations for store data (*e.g.*file input)
* Add new items/activities with validation for input
* Different payment methods
* Thread-safety for the entire application, where relevant, to enable multiple overlapping Controller interactions *e.g.* network access from multiple sources
* Allow user to add more of the selected product directly in the cart
* Add additional Controller input sources *e.g.* network input
* Better separation of the Model, View, and Control elements of a MVC

### 7.2.ii) GUI Improvements

* Reflect modifications following core functionality improvements (*e.g.* new user)
* Increased GUI internationalisation
* Select user source (see Runtime.java)
* Could display products/activities on a panel with modified purchase options to allow for multiple requests at the same time
* Default runtime parameters are currently provided, but encapsulated, by Runtime.java. These could be overridden through a graphical interface or command line input and passed as a parameter to FantasyLand.java or a configuration file – possibly edited from the GUI.
* Display prices to two decimal places to reflect actual money denominations, not required in this implementation as prices are fixed at whole numbers
* Add alternative Viewers or other Controller accessors *e.g.* web interface or multi-threaded network access. Provide intermediate viewer output *e.g.* exporting HTML tagged data that can be displayed via a web browser or java swing windows. Another example would be XML-tagged data. Another example could be exporting a serialised java object that could be imported by a remote viewer

### 7.2.iii) Testing Improvements

* More regression tests for core functionality
* A robust testing strategy for new GUI or logic modifications

## 7.3) Limitations

* GUI displays prices to one decimal place
* Context.user is not sensitive to the removal of a user or zone that it is tracking