Final documentation for MAS subject, s20512 Bohdan Miroshnychenko

Fantasy Shop – Not Just a Fantasy. Complete documentation for the project

# The Introduction

This project aims to reproduce a shop environment, implemented as a desktop application. Whether you are an RPG enthusiast, or just like Lord Of The Rings, this will seem familiar to you. A shop is structured around transactions, both user-to-user(in the case of workers and customers) as well as entity-to-user(in case an item is acquired in ways other than purchasing). The Fantasy Shop works to create a whole ecosystem of item acquisition and transfer between users and shops. It is very important, subsequentially, that items are detailed and intricate, instead of just being semi-blank objects. The shop itself is a large network, where actors can pose as customers who acquire items, management team who distributes them and edits accordingly, as well as the combination of both of them, allowing for fluent transactions which are as error-proof as they can be in the project of such scale. My aim is to describe implementation steps and provide all necessary diagrams for such project. The document is broken down into different parts, which you can easily access through the use of Table of Contents and Table of Figures, provided bellow.



Figure 1. What I try to emulate

Table of Contents

[I. The Introduction 1](#_Toc106114657)

[III. Table of Figures 3](#_Toc106114658)

[IV. User Requirements 4](#_Toc106114659)

[V. Use Case Diagram 5](#_Toc106114660)

[VI. Analytical Class Diagram 6](#_Toc106114661)

[VII. Design Class Diagram 7](#_Toc106114662)

[VIII. Selected Use Case 8](#_Toc106114663)

[A. Basic Flow: Client Buys An Item 8](#_Toc106114664)

[B. Alternative Flow: Client does not chose preferences 9](#_Toc106114665)

[C. Alternative Flow: User does not have enough funds 10](#_Toc106114666)

[IX. Activity Diagram for UC01 12](#_Toc106114667)

[X. State Diagram For UC01 13](#_Toc106114668)

[XI. Interaction Diagram For UC01 14](#_Toc106114669)

[XII. GUI Design 15](#_Toc106114670)

# Table of Figures

[Figure 1. What I try to emulate 2](#_Toc169000907)

[Figure 2. Use Case Diagram 5](#_Toc169000908)

[Figure 3. Analytical Class Diagram 6](#_Toc169000909)

[Figure 4. Design Class Diagram 7](#_Toc169000910)

[Figure 5. Activity Diagram 12](#_Toc169000911)

[Figure 6. State Diagram 13](#_Toc169000912)

[Figure 7. Interaction Diagram 14](#_Toc169000913)

[Figure 8. UI Design, Initial Scene 15](#_Toc169000914)

[Figure 9. Shop Items Scene 16](#_Toc169000915)

[Figure 10. Successful Purchase Result Screen 17](#_Toc169000916)

# User Requirements

User Requirements will be split inside a multi-leveled list, uniquely named for the purpose of future use and reference.

* UR010: The system should store all of the appropriate information about shops, people and other classes in a local database
* UR011: The system should allow for the creation of objects in the database, as long as the requirements for the creation, outlined in the objects themselves, are met
* UR012: The user should be able to log in without credentials, just by picking 0-\* preferences from a list on the first scene, after launching the program
* UR013: The user should be able to select a shop, available as a drop down list, and that selection should be remembered, until the app stops working or another selection is made
* UR014: Based on the selection from UR013, the user should see the list of items, available(related to) the selected shop
* UR015: The user should be able to select an item, for which they have enough funds and purchase it, severing the relationship between shop and item, and creating a new one, between item and customer
* UR016: The back-end of the application, not related to UC01 should be implemented properly, i.e. exemplar data should be present, instances created and associations established
* UR017: Many different instances of class Building should be allowed for creation, having different available fields and methods, depending on the type of building. All methods and fields should be available for the building type Shop, as it is the main concern of this project
* UR018: All choices, other than to buy an item should be reversible, so the implementation should allow for back-tracking

# Use Case Diagram

The following Use Case Diagram provides a simple, high-level understanding of the focus of implementation. Although the back-end of the system is implemented properly, the user is not able to access it, due to the limitation of the scope of the project. The Use Case titled “Request a Buy transaction” is UC01, which will be properly described in the following sections of the document

A picture containing text, diagram, screenshot, font

Description automatically generated

Figure 2. Use Case Diagram

# Analytical Class Diagram

The goal of this diagram – to prove the concept and show basic design decisions, without delving to much into details and implementation. It shows main classes, relationships with them and more.

Figure 3. Analytical Class Diagram

A diagram of a computer flowchart

Description automatically generated

# Design Decisions

In the Java language, not all UML constructs exist, so a few choices needed to be made about the implementations. The following list will describe each design challenge and a way of implementation. The next chapter will show on the Design Class Diagram, the implementation of discussed topics

* **Identifiers:** unique identifiers, which were not present on the analytical diagram are managed by a separate non-business class called IDManager, which assign unique ID’s
* **Class Extent:** every meaningful business class has a class extent attached to it. For all but one class, these extents are ArrayLists, which are protected from manipulation. The object is added to the extent on object creation, and the extent can only be accessed as an unmodifiable list. Class Dungeons uses a TreeSet for the extent for the purposes of automatic sorting using a comparator
* **Extent Persistence:** it is achieved by reading/writing the extents into separate files for each class
* **Complex Attribute:** each Customer should have a license, to go into dungeons. I decided to implement this as a separate class License, which cannot exist without a Customer – license constructor is private, and the license is created inside the constructor of Customer
* **Multi-Valued Attribute:** Workers have a set of skills, which is implemented using an Enum list
* **Class Attribute:** salary brackets in Workers specify howe much money each position can earn. It is implemented as a few attributes, which have static, class-wide values.
* **Derived Attribute:** all weapons have a derived attribute called getDamage, which performs a series of statistical calculations in order to arrive at an approximate damage number for the weapon specified. Upon further consideration, in the implementation, 2 variants were created, one for the average user stats with empty input params, and the second one, which can take custom user parameters as 3 int values
* **Class Method:** mostSuccessfullDDelve allows us to aggregate on the class extent and output a Dungeon Delve, which yielded the most expensive collection of Loot
* **Regular Association:** many classes are connected this way, and backwards referencing was kept in mind during implementation
* **Qualified Association:** Dungeon Delves can be uniquely found, because the association between Dungeon and Dungeon Delve is a qualified association. During implementation, 2 methods findDelveByID(Long id) and addDelve(DungeonDelve dd) were added to implement this construct, keeping the relations between an ID and a DDelve in a Map
* **Abstract Class with Polymorphic Methods:** getDamage, the method which governs the derived attribute “Damage” is also a polymorphic method, which changes depending on the class. It was implemented in the abstract class Weapon, and then in each weapon class separately, with different rules.
* **Overlapping Inheritance:** I need to implement a system, where buildings would have different purposes, which can be combined(most commonly “Shop” and “Storage”). For this, Overlapping Inheritance was chosen. It is implemented by flattening the hierarchy, leaving everything in one superclass, keeping info about building types in an EnumSet, and evoking special rules for certain operations, which require a certain type to be executed
* **Dynamic Inheritance:** valuables are important only because of their value, so the exact type of valuable is not really important. Because of this, I decided to implement dynamic inheritance between different types of valuables. It is implemented by having a secondary constructor for each valuable class, which accepts a different kind of valuable as a parameter, converting it to a new one, while keeping the value of it the same
* **Attribute Constraints:** a dynamic attribute constraint would be the wages in the Worker, which are controlled by a method checkWage, invoked automatically in the constructor and setter. It is implemented by storing multiple bracket values as class attributes, which can be accessed by this method to confirm that the wage was placed in a correct bracket. The static constraint would be implemented for the Axes by the method checkAdditionalCritDamage, also automatically invoked, to check against a predetermined number.
* **Unique Constraints:** dungeon names should be unique. This is implemented by checking uniqueness of created or changed dungeons against the class extent
* **Ordered Constraint:** as mentioned previously, the class extent for Dungeons is a bit different, using a TreeSet. This is because I wanted to impose an ordered constraint, which orders dungeons by level whenever their list is accessed, by adding a comparator to that TreeSet.
* **Bag Constraint:** my Transaction class uses the Bag constraint, as shown on the analytical diagram, but it is implemented as a class, which has correct references with bag references towards the classes it is connected to – Customer and Loot
* **Custom Business Constraint:** my shop sells weapons, and they cannot be unbalances, so whenever a weapon is created or changed, a method called checkBallance is called, imposing a constraint, which checks for average damage and assesses it based on the declared rarity. If the rarity does not match average damage – an exception is raised

# Design Class Diagram

The design Class Diagram aims to showcase the implementation process and guideline, as well as provide an in-depth look into the whole project structure. Underlined red are the static class extents.

A diagram of a computer program

Description automatically generated with medium confidence

Figure 4. Design Class Diagram

# Selected Use Case

|  |  |
| --- | --- |
| **Use Case UC01** | Client Buys An Item |
| **Actor** | Client |
| **Use Case Overview** | Our user, who is logged in as a client, is able to select a shop, pick an item from it and buy it. |
| **Subject Area** | Shop Enterprise |
| **Trigger** | User wants to buy an item |
| **Precondition 1** | All shops are associated with at least one item |
| **Postcondition 1** | The item is not associated with the shop, but is associated with the client |

## **Basic Flow:** Client Buys An Item

|  |  |
| --- | --- |
| **Description** | This scenario describes the situation where the user tries to buy an item |
| **1** | User selects preferences from a Dropdown List |
| **2** | User selects a shop they want to browse from a list |
| **3** | User observes a list of items, sorted based on the preferences, chosen at the start |
| **4** | User selects an item they want to buy |
| **5** | User presses “Buy” button |
| **6** | The item association is removed from the shop and funds are deducted from the user’s account |
| **7** | The item is added to the user’s inventory and new association is created |
| **Terminate Outcome** | A transaction is created and stored in the database. Item has new, proper association |

## Alternative Flow: Client does not chose preferences

|  |  |
| --- | --- |
| **Description** | This scenario describes the situation where the user does not pick any preferences |
| **1** | User selects a shop they want to browse from a list |
| **2** | User observes a list of items, not sorted |
| **3** | User selects an item they want to buy |
| **4** | User presses the “Buy” button |
| **5** | The item association is removed from the shop and funds are deducted from the user’s account |
| **6** | The item is added to the user’s inventory and new association is created |
| **Terminate Outcome** | A transaction is created and stored in the database. Item has new, proper association |

## Alternative Flow: User does not have enough funds

|  |  |
| --- | --- |
| **Description** | This scenario describes the situation where the user picks preferences, shop and item, but does not have enough money to buy it |
| **1** | User selects preferences from a Dropdown List |
| **2** | User selects a shop they want to browse from a list |
| **3** | User observes a list of items, sorted by preferences selected earlier |
| **4** | User selects an item they want to buy |
| **5** | User presses the “Buy” button |
| **6** | A pop-up window, displaying the message “You do not have enough funds” is displayed |
| **7** | The user is returned to the item selection screen |
| **Terminate Outcome** | A transaction is not created |

This use case was picked as a focus for this project, due to it being one of the broadest in reach and touching the most classes at the same time. UC01 interacts with most important systems and shows main functionality of a shop – creating and completing transactions. In the following parts this use case will be described by Activity, State and Interaction diagrams.

# Activity Diagram for UC01

The Activity Diagram describes the flow of actions between the user and the system. Although it is clear that more actions are done on the system side, it is still important to see the back-and-forth.

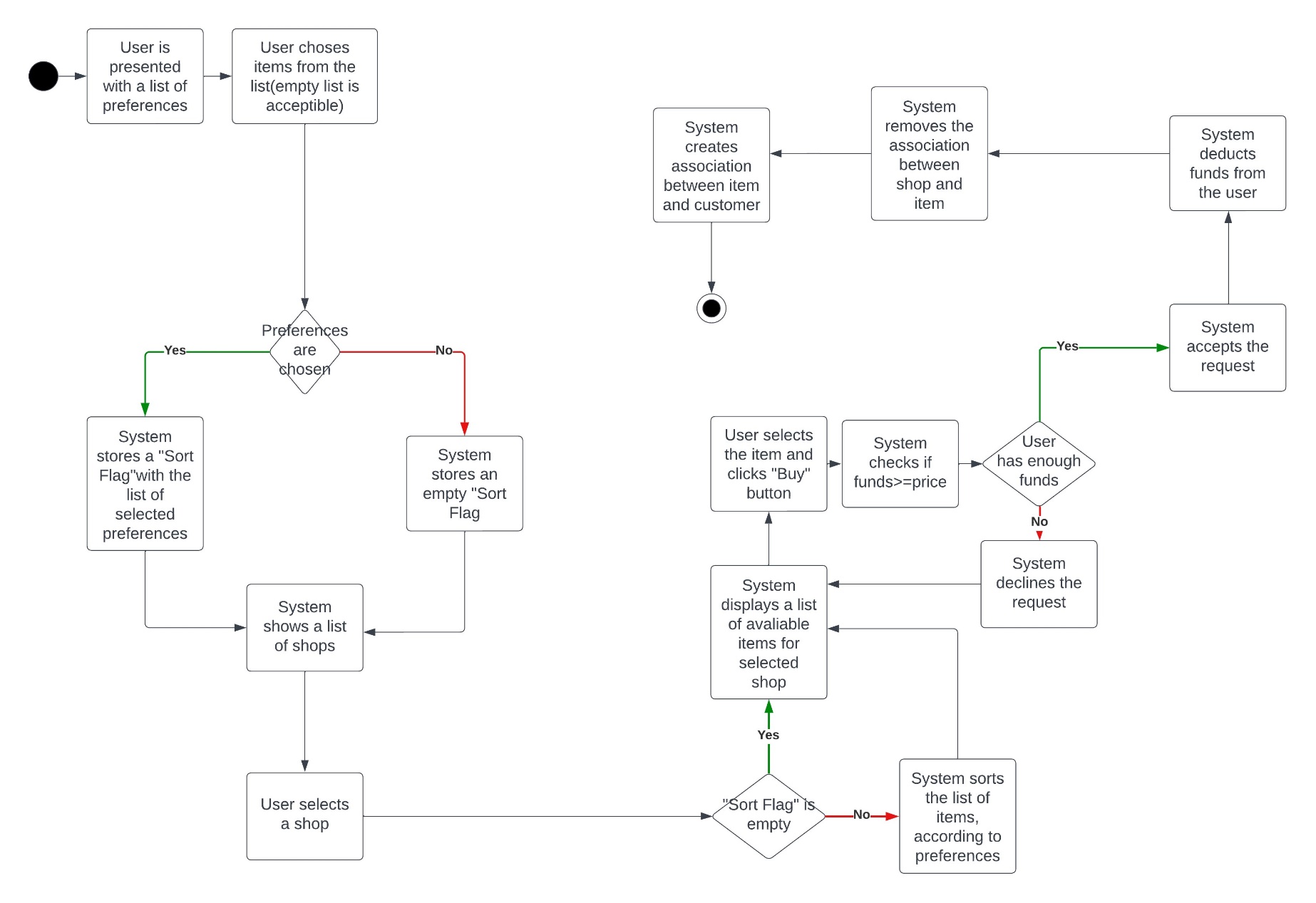


Figure 5. Activity Diagram

# State Diagram For UC01

The following State Diagram showcases the different states of the item in the shop.

A white screen with black text

Description automatically generated

Figure 6. State Diagram

# Interaction Diagram For UC01

The following Interaction Diagram provides an inside on the back-end of the system. It shows how the user action impact the system and what methods are called on high-level approach.

A diagram of a software project

Description automatically generated

Figure 7. Interaction Diagram

# GUI Design

The approach, chosen for this project, is very minimalistic and to-the-point. I have made a choice to focus on productivity and visibility, instead of fancy graphics. The individual scene designs is found below, while the logical progression between scenes is from top image, to the bottom image.

A screenshot of a computer

Description automatically generated

Figure 8. UI Design, Initial Scene

A screenshot of a computer

Description automatically generated

Figure 9. Shop Items Scene

A screenshot of a computer

Description automatically generated

Figure 10. Successful Purchase Result Screen