Ryerson Good Food Center Application

Problem Elicitation, Implementation and Testing Report

K.V., Y.B., L.Z.

Table of Contents

Problem Description 3
- Delivery Man 3
- Food Center Volunteer 3
- Customer 3

Functional Requirements 4
- Use case name: DeliverFood 4
- Use case name: ViewServiceAlerts 4
- Use case name: OrganizeFood 5
- Use case name: GetFood 5

Design 6
- Factory Method 6

Testing 7

References 8

Problem Description

Our application has three potential users: the 1) Food Delivery Man, 2) Food Center Volunteer, 3) Customer. For brevity, we will denote them as DM, FCV and C, respectively.

The Ryerson Good Food Center is an actual location in Ryerson University, which, for at least three years, has operated without an end user application and is based off word of mouth or Ryerson's end user interface (a.k.a RAMMS). Our application will focus on the Food-Bank aspect of the Center, although there are other services available. More information about them can be found here: http://www.rsuonline.ca/good-food-centre

This current implementation of the 'Center' is somewhat limiting since users usually have no idea what food is available for the day and cannot prepare in advance. In the case of a driver emergency, customers and volunteers are usually notified with very little notice; this generally breaks schedules. Our implementation aims to solve this and more problems as per the design elicitation:

Our application will serve as an interface for *DM, FCV,* and *C,* which are implementations of *AbstractPerson*.

Delivery Man

DM has initial access to the *FoodArray* object. Before a delivery, only *DM* has access to *FoodArray,* and *FCV, C,* have no access to it beforehand. When delivering, *DM* passes *FoodArray* off to *FCV*. *DM* only knows the quantities, *names* and *countryOfOrigin* of these items and not their *pointValue*, *isRefrigerated* or other properties which are ultimately determined by *FCV*. At this stage, *C* still has no access to *FoodArray*. *DM* can open *ServiceTicket*s to let the *FCV, C* know of three emergencies: *LeftKeyInCar*, *VehicleAccident,* and *SmallOrder,* all which satisfy the *ServiceTicket* Interface.

Food Center Volunteer

FCV has access to the *DM*'s *FoodArray* object after it has been delivered but has to organize it. *FCV* **must** add the properties int *pointValue,* boolean *isRefrigerated* (There are other properties in the real-life implementation of FoodCenter, however, these serve the proof of concept). After sorting the *FoodArray*, *C* can access the array. They can view *ServiceTicket*s raised by *DM*. *FCV* has the added capability of adding more *C*s and *FCV*s.

Customer

C can query the app to see if there is a *ServiceTicket* raised by the *DM*. After delivery and sorting, *C* will have access to the *FoodArray.* They are then able to take items from the *FoodArray.* These items are subtracted from the *FoodArray* object. *C* is unable to take more items that surpass their ten-point limit.

Functional Requirements

Use case name: **DeliverFood**

Participating Actors:	Initiated by *DM* Communicates with *FCV*
Flow of Events:	1. The *DM* is packaging all the food to be delivered for the day. *DM* activates the *DeliverFood* function of the terminal. 2. FoodCenterApp responds by presenting a form to *DM*. 3. *DM* fills out the form by inputting the name of the food item, and the country of origin (if known). The *DM* continually does this for each individual item. 4. FoodCenterApp acknowledges the changes and informs *FCV*. 5. *DM* makes his delivery to the Food Center. 6. *FCV* receives the food.
Entry Condition:	- *DM* is logged into FoodCenterApp. - *DM* has at least one food item to deliver.
Exit Condition:	- *DM* receives acknowledgement of successful recording by FoodCenterApp.
Quality Requirements:	- *DM*'s input is acknowledged immediately. - The information from *DM*'s input is reported to the *FCV* within 10 seconds.

Use case name: **ViewServiceAlerts**

Participating Actors:	Initiated by either *DM, FCV,* or *C.*
Flow of Events:	1. The DM, FCV or C would like to view the available service alerts recorded on the system. 2. FoodCenterApp responds by showing the most recent *ServiceAlert*, or display 'No service alerts' (or similar) if none have been recorded by *DM*.
Entry Condition:	- The *DM, FCV* or *C* is logged into FoodCenterApp.
Exit Condition:	- *DM, FCV* or *C* receives the corresponding service alert.
Quality Requirements:	- *DM*'s, *FCV*'s or *C*'s query is responded immediately. - The information relayed accurately represents the latest *ServiceTicket*.

Use case name: **OrganizeFood**

Participating Actors: Initiated by *FCV*

Flow of Events:

1. The *FCV* receives the food delivery from *DM*.

2. *FCV* activates the *OrganizeFood* function on their terminal.

2. FoodCenterApp responds by presenting a form to *FCV*, listing the names of the food items as well as their *countryOfOrigin*.

3. *FCV* fills out the form by inputting the *pointValue* of the food item, as well as noting if the food item has to be refrigerated or not. The *FCV* continually does this for each individual item.

4. FoodCenterApp acknowledges the changes and sets all the corresponding values to each food item in the *FoodArray*.

Entry Condition:

- The *FCV* is logged into FoodCenterApp.
- FoodArray has at least one food item to organize.

Exit Condition:

- *FCV* receives acknowledgement of successful recording by FoodCenterApp.

Quality

- *FCV*'s input is acknowledged immediately.

Requirements:

- The information from *FCV*'s input is successfully applied to each corresponding food item.

Use case name: **GetFood**

Participating Actors: Initiated by *C*

Flow of Events:

1. *C* activates the *GetFood* function of their user agent.

2. FoodCenterApp responds by presenting a form to *C*. In this form each item is named, as well as the *countryOfOrigin*, and the *pointValue*, and asks if *C* would like the item.

3. *C* fills out the form by responding to each query by form of y/n (yes, no).

4. FoodCenterApp acknowledges the request. If yes, it removes that item from the *FoodArray* and deducts the corresponding point value from *C*'s points (if there are sufficient points. If there are insufficient points, *C* is unable to get that item. If no, the next item is displayed.

5. *C* obtains the food items they requested.

6. FoodCenterApp acknowledges the results and reorganizes *FoodArray*.

Entry Condition:

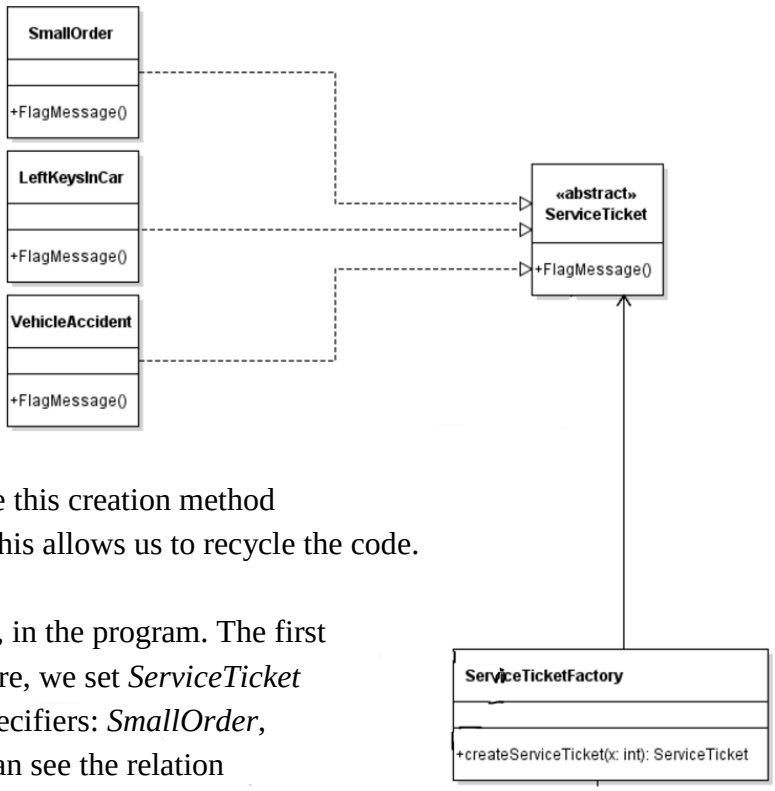
- *C* is logged into FoodCenterApp.

	- C has sufficient points to obtain at least one item.
Exit Condition:	- C receives acknowledgement of successful recording by FoodCenterApp and receives their food
Quality	- DM's input is acknowledged immediately.
Requirements:	- The information from DM's input is reported to the FCV within 10 seconds.

Design

In this project, we used the Factory Method design pattern.

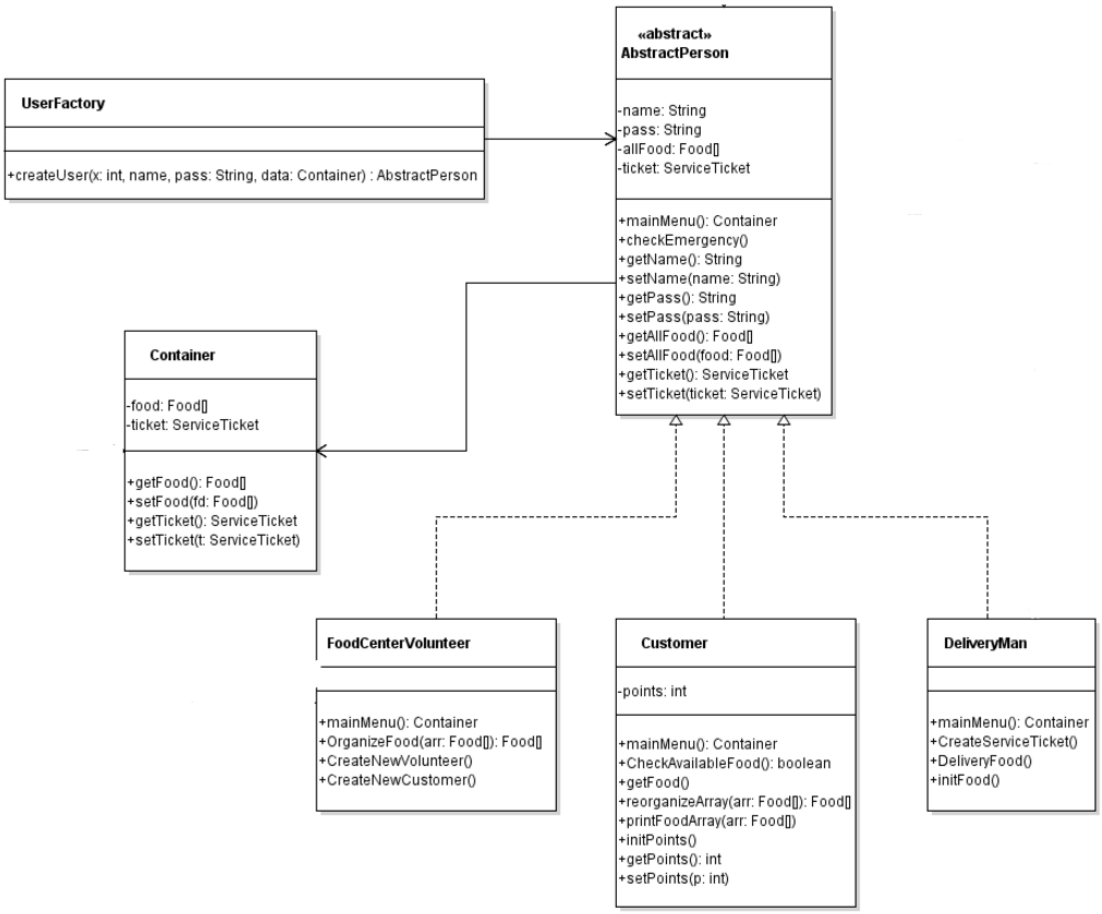
Factory Method



In the program, we extensively use Factory Method to create new *DM*, *FCV* or *C* objects as required by the login information. By defining a separate method for creating objects, we create a factory floor where subclasses can override this creation method to create the specific product as required. This allows us to recycle the code.

The factory method was used in two places, in the program. The first place was the service ticket constructor. Here, we set *ServiceTicket* as an abstract and depended on the three specifiers: *SmallOrder*, *LeftKeysInCar*, and *VehicleAccident*. We can see the relation to *ServiceTicketFactory* as clearly depending on these specifications. This is exemplified in the snippet of the diagram to the left. (Note, for the sake of format, the connection from *ServiceTicket* to *AbstractPerson* was removed, but is present in the attached UML diagram).

The second place where factory method is more intricately for the creation of *Users*. We see the same linear connection between *AbstractPerson* and *FCV, C,* and *DM*.
However, the *UserFactory* relies on more inputs from the user (these being the name, password. Else, we have program placed inputs such as *Container* and *AbstractPerson*). Here, the container is important in being able to pass on multiple parameters to user creation, and being able to expect them back (since functions can only return at most one primitive or object, we make it send an object composed of a *FoodArray* and the *ServiceTicket*). (For the sake of format, some connections were omitted, although they are present in the attached UML diagram, please view next page.)



Testing

For testing unit predefined constructors of several classes:

Test case ID	Test case Description	WhiteBox/ Blackbox
testContainerConstructor	Tests the constructor for the Container class. Verifies the methods used in the constructor by comparing with getters for the private instance variables.	Blackbox
testCustomerConstructor	Tests the constructor for the Customer class. Verifies the methods used in the constructor by comparing with getters for the private instance variables.	Blackbox
testDeliveryManConstructor	Tests the constructor for the DeliveryMan class. Verifies the methods used in the constructor by comparing with getters for the private instance variables.	Blackbox

testFoodVolunteerConstructor	Tests the constructor for the FoodVolunteer class. Verifies the methods used in the constructor by comparing with getters for the private instance variables.	Blackbox
testUserFactoryConstructor	Tests the constructor for the UserFactory class. Verifies the methods used in the constructor by comparing with getters for the private instance variables.	Blackbox
testfoodsConstructor	Tests the constructor for the Food class. Verifies the methods used in the constructor by comparing with getters for the private instance variables.	Blackbox

For testing Food Class:

testsetCountryOfOrigin	Tests the setCountryOfOrigin method in the Food class. Verifies the method by checking Instance Variable with a getter after setting the variable with Setter.	Whitebox
testsetFood	Tests the setFood method in the Food class. Verifies the method by checking Instance Variable with a getter after setting the variable with Setter.	Whitebox
testsetName	Tests the setName method in the Food class. Verifies the method by checking Instance Variable with a getter after setting the variable with Setter.	Whitebox

For testing the Ticket Class:

testsetTicket	Tests the setTicket method in the Ticket class. Verifies the method by checking Instance Variable with a getter after setting the variable with Setter.	Whitebox

References

Design patterns - Factory method. (n.d.). Retrieved November 13, 2017, from https://sourcemaking.com/design_patterns/factory_method

RSU - Good Food Center. (n.d.). Retrieved November 12, 2017, from http://www.rsuonline.ca/good-food-centre